Hardware Algorithm Implementation
For
Mission Specific Processing

Thesis

Jason W. Shirley, Captain, USAF

AFIT/GE/ENG/08-27

Department of the Air Force
Air University

Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio

Approved for public release; distribution unlimited.
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.
Hardware Algorithm Implementation
For
Mission Specific Processing

Thesis

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Electrical Engineering

Jason W. Shirley, B.S.E.E.
Captain, USAF

March 2008

Approved for public release; distribution unlimited.
Hardware Algorithm Implementation
For
Mission Specific Processing

Jason W. Shirley, B.S.E.E.
Captain, USAF

Approved:

/signed/  27 Feb 2008
______________________________
Yong C. Kim, PhD (Chairman)  date

/signed/  27 Feb 2008
______________________________
LaVern A. Starman, PhD, Maj (Member)  date

/signed/  27 Feb 2008
______________________________
Guna S. Seetharaman, PhD (Member)  date

/signed/  27 Feb 2008
______________________________
Juan R. Vasquez, PhD (Member)  date
Abstract

There is a need to expedite the process of designing military hardware to stay ahead of the adversary. The core of this project was to build reusable, synthesizeable libraries to make this a possibility. In order to build these libraries, Matlab® commands and functions, such as Conv2, Round, Floor, Pinv, etc., had to be converted into reusable VHDL modules. These modules make up reusable libraries for the Mission Specific Process (MSP) which will support AFRL/RY.

The MSP allows the VLSI design process to be completed in a mere matter of days or months using an FPGA or ASIC design, as opposed to the current way of developing a system which can take 1-2 years to complete. By having the libraries built, the components can be implemented in an FPGA or ASIC design over and over again. The libraries make it possible to make upgrades to weapons systems to meet the ever-changing needs the War Fighter faces. MSP makes it possible to develop various algorithms, including algorithms implemented in Matlab®. The MSP libraries were built and tested using TSMC 250-nm® technology library from the Taiwan Semiconductor Manufacturing Company. They were also synthesized for an FPGA. The modules were all synthesized using the CAD tools from Cadence® and Mentor Graphics®. Power, area, and delay results for each module were presented.
Acknowledgements

First and foremost, I would like to thank my Lord and Savior Jesus Christ for giving me the wisdom and knowledge to complete my thesis. I also owe a large debt of gratitude to my wife and daughter for their understanding and patience during my research at the Air Force Institute of Technology (AFIT). I would like to thank Dr. Yong Kim for his wisdom, knowledge, and helpfulness during my research as he guided and encouraged me throughout my research process. I would like to thank Dr. Juan Vasquez and Douglas Summers-Stay for sponsoring me, and I hope that the research that I have gathered is useful to the Air Force Research Lab (AFRL) AFRL/RY. Finally, I’d like to thank my fellow students for all of their encouragement and knowledge during my research efforts: Sam Stone, Benjamin Hancey, Eric Lam, James Crouch, and Gene Hockenberry.

Jason W. Shirley
# Table of Contents

Abstract ......................................................... iv  
Acknowledgements .............................................. v  
List of Figures .................................................. ix  
List of Tables ................................................... xi  
List of Abbreviations .......................................... xii  

I. Introduction .................................................. 1  
   1.1 Specific Issue ............................................ 4  
   1.2 Problem Statement ........................................ 4  
   1.3 Scope and Assumptions ................................... 5  
   1.4 Thesis Organization ....................................... 5  

II. Background ................................................... 7  
   2.1 Overview of VLSI Design Process ......................... 7  
   2.2 VLSI Design Process ...................................... 8  
      2.2.1 Specification ......................................... 9  
      2.2.2 Architecture ......................................... 9  
      2.2.3 RTL Coding ......................................... 11  
      2.2.4 RTL Verification ..................................... 11  
      2.2.5 Synthesis/Manual Layout ............................. 11  
      2.2.6 Implementation/Fabrication ......................... 12  
   2.3 FPGA vs. ASIC Risk ....................................... 13  
   2.4 MSP Design Reusability ................................... 14  
   2.5 Optical Flow ............................................. 17  
   2.6 Chapter Summary ......................................... 19  

III. Methodology .................................................. 20  
   3.1 Overall Design ........................................... 21  
   3.2 Two Dimensional Convolution ............................. 22  
      3.2.1 Memory .............................................. 23  
      3.2.2 Multiplier ........................................... 23  
      3.2.3 Control ............................................. 24  
      3.2.4 Multiplexer, Adder, Register ....................... 24  
   3.3 Matrix Transpose ......................................... 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4 Compute Derivatives</td>
<td>25</td>
</tr>
<tr>
<td>3.5 Reduce Matrix Function</td>
<td>27</td>
</tr>
<tr>
<td>3.6 Round/Floor</td>
<td>28</td>
</tr>
<tr>
<td>3.7 Pseudoinverse (Pinv)</td>
<td>28</td>
</tr>
<tr>
<td>3.8 Synthesis/Timing</td>
<td>29</td>
</tr>
<tr>
<td>3.9 Testing Procedure</td>
<td>30</td>
</tr>
<tr>
<td>3.10 Chapter Summary</td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV. Analysis and Results</td>
<td>32</td>
</tr>
<tr>
<td>4.1 Overview of MSP Modules</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Error Analysis</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Power, Area, and Delay</td>
<td>40</td>
</tr>
<tr>
<td>4.4 Synthesis</td>
<td>44</td>
</tr>
<tr>
<td>4.5 Chapter Summary</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>V. Conclusions</td>
<td>51</td>
</tr>
<tr>
<td>5.1 Summary of the Project</td>
<td>51</td>
</tr>
<tr>
<td>5.2 Future Work</td>
<td>51</td>
</tr>
<tr>
<td>5.2.1 Improve Module Designs</td>
<td>52</td>
</tr>
<tr>
<td>5.2.2 Power, Area, and Delay</td>
<td>52</td>
</tr>
<tr>
<td>5.2.3 Complex Numbers</td>
<td>52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix A. Reducing Power Consumption</td>
<td>53</td>
</tr>
<tr>
<td>A.1 Clock Speed/Clock Gating</td>
<td>53</td>
</tr>
<tr>
<td>A.2 Turning Off The Circuit</td>
<td>54</td>
</tr>
<tr>
<td>A.3 Recycle Power</td>
<td>55</td>
</tr>
<tr>
<td>A.4 Redesign Logic Of The Circuit</td>
<td>56</td>
</tr>
<tr>
<td>A.5 Higher Level of Integration</td>
<td>56</td>
</tr>
<tr>
<td>A.6 Dynamic Power Management</td>
<td>56</td>
</tr>
<tr>
<td>A.6.1 Clock Gating</td>
<td>57</td>
</tr>
<tr>
<td>A.6.2 Qualified System Latches</td>
<td>58</td>
</tr>
<tr>
<td>A.6.3 Guarded Evaluation</td>
<td>58</td>
</tr>
<tr>
<td>A.6.4 Bus Deactivation</td>
<td>59</td>
</tr>
<tr>
<td>A.6.5 Self-timed Techniques</td>
<td>59</td>
</tr>
<tr>
<td>A.7 Synthesize</td>
<td>59</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Power, Area, and Delay Triangle</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>A Sample of Target Tracking Imagery</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Three Main Steps for VLSI Design Process</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Six Steps for VLSI Design Process</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Nand Gate Layout</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Standard Design Time vs. MSP Design Time</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Optical Flow Vectors for a $250 \times 400$ Image</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Reduce Matrix Function Top Level Design 1</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Conv2 Sub-Modules Design</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Behavior Multiplier Module Design</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Booth Multiplier Module Design</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Reduce Matrix Function Top Level Design 2</td>
<td>34</td>
</tr>
<tr>
<td>4.6</td>
<td>Multiplier State Machine Design</td>
<td>35</td>
</tr>
<tr>
<td>4.7</td>
<td>Booth Multiplier State Machine Design</td>
<td>35</td>
</tr>
<tr>
<td>4.8</td>
<td>Compute Derivatives Top-Level Design</td>
<td>36</td>
</tr>
<tr>
<td>4.9</td>
<td>Matrix Transpose Top-Level Design</td>
<td>36</td>
</tr>
<tr>
<td>4.10</td>
<td>Pseudoinverse Top-Level Design</td>
<td>37</td>
</tr>
<tr>
<td>4.11</td>
<td>Reduce Matrix Function Matlab® vs. VHDL Percent Error</td>
<td>38</td>
</tr>
<tr>
<td>4.12</td>
<td>Reduce Matrix Function Module Net Power(mW) vs. Clock Speed</td>
<td>40</td>
</tr>
<tr>
<td>4.13</td>
<td>Reduce Matrix Function Module Cell Area vs. Clock Speed</td>
<td>41</td>
</tr>
<tr>
<td>4.14</td>
<td>Compute Derivatives Module Net Power(mW) vs. Clock Speed</td>
<td>41</td>
</tr>
<tr>
<td>4.15</td>
<td>Compute Derivatives Module Cell Area vs. Clock Speed</td>
<td>42</td>
</tr>
<tr>
<td>4.16</td>
<td>Matrix Transpose Module Net Power(mW) vs. Clock Speed</td>
<td>42</td>
</tr>
<tr>
<td>4.17</td>
<td>Matrix Transpose Module Cell Area vs. Clock Speed</td>
<td>43</td>
</tr>
<tr>
<td>4.18</td>
<td>Pseudoinverse Module Net Power(mW) vs. Clock Speed</td>
<td>43</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.19.</td>
<td>Pseudoinverse Module Cell Area vs. Clock Speed</td>
<td>44</td>
</tr>
<tr>
<td>A.1.</td>
<td>Energy Recovery Clocked Flip-Flops with Clock Gating</td>
<td>54</td>
</tr>
<tr>
<td>A.2.</td>
<td>Clock Recovery Circuit</td>
<td>55</td>
</tr>
<tr>
<td>A.3.</td>
<td>Illustration of Clock Gating</td>
<td>57</td>
</tr>
<tr>
<td>A.4.</td>
<td>Guarded Evaluation</td>
<td>58</td>
</tr>
<tr>
<td>A.5.</td>
<td>Leakage Power Optimization Flow</td>
<td>60</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.</td>
<td>Summary of Maximum Clock Frequency for Modules</td>
<td>16</td>
</tr>
<tr>
<td>3.1.</td>
<td>Configurable parameters</td>
<td>21</td>
</tr>
<tr>
<td>4.1.</td>
<td>Comparison between $\text{Matlab}^\circledR$ and VHDL Gaussian Distribution</td>
<td>38</td>
</tr>
<tr>
<td>4.2.</td>
<td>Comparison between $\text{Matlab}^\circledR$ and VHDL Pseudoinverse Module</td>
<td>39</td>
</tr>
<tr>
<td>4.3.</td>
<td>FPGA Synthesize Results for Top-Level 1, Behavior Multiplier</td>
<td>45</td>
</tr>
<tr>
<td>4.4.</td>
<td>FPGA Synthesize Results for Top-Level 2, Behavior Multiplier</td>
<td>45</td>
</tr>
<tr>
<td>4.5.</td>
<td>FPGA Synthesize Results for Top-Level 1, Booth Multiplier</td>
<td>46</td>
</tr>
<tr>
<td>4.6.</td>
<td>FPGA Synthesize Results for Top-Level 2, Booth Multiplier</td>
<td>46</td>
</tr>
<tr>
<td>4.7.</td>
<td>FPGA Synthesize Results for Compute Derivative Fx Module</td>
<td>47</td>
</tr>
<tr>
<td>4.8.</td>
<td>FPGA Synthesize Results for Compute Derivative Fy Module</td>
<td>47</td>
</tr>
<tr>
<td>4.9.</td>
<td>FPGA Synthesize Results for Compute Derivative Ft Module</td>
<td>47</td>
</tr>
<tr>
<td>4.10.</td>
<td>FPGA Synthesize Results for Matrix Transpose Module</td>
<td>48</td>
</tr>
<tr>
<td>4.11.</td>
<td>FPGA Synthesize Results for Floor Module</td>
<td>48</td>
</tr>
<tr>
<td>4.12.</td>
<td>FPGA Synthesize Results for Round Module</td>
<td>49</td>
</tr>
<tr>
<td>4.13.</td>
<td>FPGA Synthesize Results for Pseudoinverse Module</td>
<td>49</td>
</tr>
<tr>
<td>4.14.</td>
<td>Summary of Maximum Clock Speed for Modules</td>
<td>49</td>
</tr>
<tr>
<td>4.15.</td>
<td>Summary of Net Power Watts for Modules</td>
<td>50</td>
</tr>
</tbody>
</table>
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFIT</td>
<td>Air Force Institute of Technology</td>
<td>v</td>
</tr>
<tr>
<td>AFRL</td>
<td>Air Force Research Lab</td>
<td>v</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large-Scale Integration</td>
<td>1</td>
</tr>
<tr>
<td>MSP</td>
<td>Mission Specific Process</td>
<td>1</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
<td>1</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
<td>3</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuits</td>
<td>3</td>
</tr>
<tr>
<td>TSMC</td>
<td>Taiwan Semiconductor Manufacturing Company</td>
<td>4</td>
</tr>
<tr>
<td>Conv2</td>
<td>Two Dimensional Convolution</td>
<td>4</td>
</tr>
<tr>
<td>Pinv</td>
<td>Pseudoinverse</td>
<td>4</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
<td>5</td>
</tr>
<tr>
<td>DoD</td>
<td>Department of Defense</td>
<td>11</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
<td>13</td>
</tr>
<tr>
<td>CLB</td>
<td>Configurable Logic Block</td>
<td>44</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Process</td>
<td>52</td>
</tr>
<tr>
<td>DPM</td>
<td>Dynamic Power Management</td>
<td>54</td>
</tr>
<tr>
<td>DC</td>
<td>Direct Current</td>
<td>55</td>
</tr>
<tr>
<td>AC</td>
<td>Alternating Current</td>
<td>55</td>
</tr>
<tr>
<td>Vp</td>
<td>Peak Voltage</td>
<td>55</td>
</tr>
<tr>
<td>Vth</td>
<td>Threshold Voltage</td>
<td>56</td>
</tr>
<tr>
<td>P</td>
<td>Power</td>
<td>56</td>
</tr>
<tr>
<td>V</td>
<td>Voltage</td>
<td>56</td>
</tr>
<tr>
<td>I</td>
<td>Current</td>
<td>56</td>
</tr>
</tbody>
</table>
I. Introduction

In today’s advancing technologies, devices are shrinking and densities of integrations are increasing. With these trends come many new challenges in designing integrated electronic circuits and systems. To achieve high performance (Power, Speed, Dynamic Range, etc.) in new integrated circuits for next-generation systems; new methodologies must be created, adopted, and executed.

The objective of this research is to examine and investigate current difficulties/problems associated with modeling and fabricating Very Large-Scale Integration (VLSI) circuits, then provide reusable library cells for AFRL/RY directorate’s Mission Specific Process (MSP) in Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). These reusable libraries can be broken into three variations which are as follows: optimizing power, minimizing area, and minimizing delay. By having these libraries built and ready to go, system requirements can be upgraded and changed in a matter of days instead of months or even years. These reusable libraries make it possible to meet the changing requirements of the operational environment. The MSP makes it flexible for system changes to be implemented quickly into a system. The variations between power, area, and delay can be generalized as the points on an equilateral triangle as seen in Figure 1.1. The points of the triangle represent the priority optimal design for the given circuit. For example, if you want to optimize the power, the other two parameters will not be the priority. This will cause the design to have minimal power consumption at the expense of the final de-
The idea behind AFRL/RY directorate’s MSP is a threefold approach to designing circuits. First, it takes time for the design to be developed and fabricated before it can get into the hands of the customer. The design to market can take up to 1-2 years depending on the technology and the level of difficulty of the circuit design. In addition to the amount of time it takes to develop the product, it can be obsolete in only a few years. This is what makes MSP so unique; it is made up of pre-built synthesizable reusable libraries so the designer doesn’t have to start from scratch. The designer can quickly use the pre-built libraries and make new ones when specifications change and add them to the library.

Second, if modifications are required for additional features to be added to the product, then you have to wait for the redesign to take place. You have to pay the vendor again for the changes they make and wait for the new design to be delivered. This is where MSP comes in to reduce the cost and schedule of redesigning a circuit just to make a few changes for additional specifications. The use of the reusable libraries makes designing a circuit much easier than current methodologies. In a
sense, it is like putting a Lego® set together. By having all the pre-built parts it’s just a matter of integrating them together for your personal application.

Thirdly, a Field Programmable Gate Array (FPGA) can replace an Application Specific Integrated Circuit (ASIC) design, unless you want absolute performance. The use of MSP makes it flexible for the designer to develop prototypes faster and cheaper for an FPGA as opposed to an ASIC design. Having MSP in your tool box can save time and money for the next generation of a circuit design. Therefore, the War Fighter will have new equipment in the field in a matter of days as opposed to the old way of doing business, which could take 1-2 years for a weapons system to be developed.

This project will support AFRL/RY in their development of a target tracking project, where the circuits and digital circuits are to be implemented in one chip. The goals are to provide reliable building blocks as portable synthesizable reusable libraries. These libraries will enable the War Fighter to get an upgradable weapon system in a matter of a few days or hours and to keep up-to-date with the ever-changing Global War on Terrorism.

Building these reusable libraries and making them usable for FPGA or ASIC designs is beneficial in many ways. For instance, programming the design on an FPGA makes it possible to protect the design against enemy hands. Anti-tamper methods on the FPGA board protects the design from being discovered by the enemy. Additionally, placing the designs on an FPGA allows the weapons system to be implemented quicker than sending the design to a foundry for fabrication as an ASIC design, which can take months due to long lead times that the foundries put in their schedule. Once the design is created on an ASIC chip it has to be thoroughly tested to find faults in the fabrication process. All these design steps take time, especially if there are problems with the completed design.
1.1 Specific Issue

The War on Terrorism has made the military soldier depend on today’s technology of global positioning systems, radar systems, and different communication devices. These devices, being mobile or not, are required for them to operate in the field and communicate with Command and Control. For this reason it is essential that these libraries be built and perfected.

With any circuit design there are three key parameters that designers face when designing circuits, and they are as follows: power, area, and delay. The designer has to make tradeoffs between these three parameters to meet their specific design constraints. Every commercial or military application has its own specifications for power, area, and delay.

The VLSI technology continues to place more and more transistors on a single chip. This allows the chips to become more powerful in computing power as the area of the chips remain small. The chips constantly require electrical power to keep them operational which makes it difficult for the War Fighter to do their mission without wondering if their batteries are going to sustain throughout their mission. There is a need to be able to run longer missions and have longer lasting equipment that doesn’t require battery change-outs in the middle of a critical mission. This low power optimization will be using a 250-nm technology library [15] from the Taiwan Semiconductor Manufacturing Company (TSMC). This library will provide a starting point to develop circuits that have lower power consumption for the future.

1.2 Problem Statement

The problem is to take the AFRL/RY Optical Flow Dense Algorithm written in Matlab® and convert the commands into synthesizable reusable library modules written in VHDL. We will be laying the foundation with these synthesizable reusable libraries for other weapons systems that require Matlab® commands such as, Round, Floor, Two Dimensional Convolution (Conv2), and Pseudoinverse (Pinv), etc. The
Optical Flow Dense Algorithm can provide various libraries to handle Unmanned Aerial Vehicles for image processing. An example of target tracking imagery can be seen in [1] Figure 1.2. The small rectangles show targets that can potentially be tracked using Optical Flow.

1.3 Scope and Assumptions

It is assumed that the reader has knowledge about the VLSI technology and understands VHDL, scripting, and intergrating the script in Cadence® software program or Modelsim®. The main software programs that will be used for this research are Modelsim®, Mentor Graphics®, and Cadence® Companies’ software tools. The simulations will be run on Modelsim® to verify the Register Transfer Level(RTL) coding. Cadence® and Mentor Graphics® software tools will be used to verify that the modules are synthesizable.

1.4 Thesis Organization

Chapter 2 of this thesis will give background information required to understand the technology options that are available for use to reduce power consumption. Each option will be briefly explained and the main focus of the research project option will be expounded into further details and discussions. Also, background information will
be given to support the design decisions used in Chapter 3. Chapter 3 will discuss the theory and methods that were used for this thesis project. Chapter 4 will look at the results that were gathered throughout this research and will be analyzed and discussed. Finally, Chapter 5 will discuss future work and topics. The ModelSim® VHDL code will be located in Appendix C.
II. Background

This chapter will give an overview of the background information used throughout this research. The MSP idea makes it flexible enough to accommodate the ever-changing Air Force missions. This section will present an overview of the VLSI Design Process as well as a more in-depth view of the process. We will also discuss FPGA vs. ASIC Risk, MSP Design Reusability, and Optical Flow.

2.1 Overview of VLSI Design Process

The VLSI Design Process can be summed up in the following three main steps: architecture, verification, and implementation and can be seen in Figure 2.1. The architecture is made up of three methods to design a circuit which are power, area, and delay (speed). Each architecture has its pros and cons for design implementation depending on what you are trying to achieve. The use of the circuit will drive what architecture you should use when designing it.

It is important to verify that your expected results match your simulated results. Once you have determined your design is working properly through simulations it is time to implement your design on an FPGA or fabricated circuit such as an ASIC.

![Figure 2.1: Three Main Steps for VLSI Design Process](image-url)
The fabrication process for a circuit can take anywhere from 3 to 6 months at the foundry. In the next section, the VLSI Design Process will be broken down for a more magnified look at the steps involved to build an operational design.

2.2 VLSI Design Process

The typical VLSI Design Process can be broken into the following six steps and can be seen in Figure 2.2:

1. Specification
2. Architecture
3. RTL Coding
4. RTL Verification
5. Synthesis
6. Implementation/Fabrication

Each of these steps will be discussed further for an understanding of what is required before a design can be implemented into the field as an operational weapons system.
2.2.1 Specification. The specification for any design comes from the customer who has a specific need for a project they want built and implemented. There are three different design strategies for circuits - custom, ASIC, and FPGA. The customer lists the criteria they want for their system, which could be a specific size (area) of the circuit, a certain power usage, or delay (speed) of the design, or even that they want the design written in VHDL. The vendor (designer) will discuss with the customer the possibilities based on the technology available at the time the original specification was created.

2.2.2 Architecture. There are three different architectures that can be implemented in any design. The three architectures are power, area, and delay (speed).

2.2.2.1 Power. Clock control plays a major role when designing a circuit to reduce power consumption. Reducing the speed of the clock for a circuit will reduce the switching activity. This results in power savings by limiting the amount of switching activity that takes place. Besides reducing the speed of the clock for the circuit, designers have also proposed “clock gating [13] by modifying the design of the existing energy recovery clocked flip-flops to incorporate a power saving feature that eliminates any energy loss on the internal clock and other nodes of the flip-flops.” According, to Steve Kilts, in his book, Advanced FPGA Design, he suggests the following:

The most effective and widely used technique for lowering the dynamic power dissipation in synchronous digital circuits is to dynamically disable the clock in specific regions that do not need to be active at particular stages in the data flow [9].

It would be ideal to have the circuit temporarily turn the clock off when a particular section of the circuit is not required to reduce power consumption. These are a few techniques that are available for use to reduce power consumption. Additional power reduction techniques can be found in Appendix A.
2.2.2.2 Area. Another architecture that can be used when developing a circuit design is area. Ways to implement reduction in area for a circuit depends on picking the correct topology. Topology that focus on reducing the area size of a circuit can be attained by reusing [9] “the logic resources to the greatest extent possible, often at the expense of throughput(speed).” When you want to increase the delay for a design you need to pipeline your design. However, to reduce the area you need to do the opposite of creating pipelines, you need to roll them up to be able to use the available resources. Also, it is a good idea, according to Steve Kilts, to [9] “share logic resources between different functional operations.”

2.2.2.3 Delay (Speed). The third architecture that can be used when designing a circuit is speed. There are three ways to describe speed in regards to circuit design - they are throughput, latency, and timing. In the book Advanced FPGA Design [9], Kilts gives a description for each one of these.

A high-throughput design is one that is concerned with the steady-state data rate but less concerned about the time any specific piece of data requires to propagate through the design (latency). A low-latency design is one that passes the data from the input to the output as quickly as possible by minimizing processing delays. Timing refers to the clock speed of a design. The maximum delay between any two sequential elements in a design will determine the max clock speed [9].

A key factor that should be considered when designing a circuit for speed is to use pipelining wherever possible to increase your throughput. The idea of pipelining is very similar to how an assembly line works. Each member is continually performing their specific task; they finish one, pass it on to the next station, and get another. The result is that a completed product is continuously produced. Pipelining [9] is a way to increase throughput for a process. Another [9] technique is to have the system run things in parallel to speed up the process. This technique is most useful when doing math calculations in a design.
2.2.3 RTL Coding. The third step in the VLSI Design Process is RTL Coding. RTL Coding is nothing more than what language you will code your design in. There are different Hardware Description Languages (HDL) that can be used to make an RTL module. The most commonly used RTL modeling languages are the following:

1. VHDL
2. Verilog
3. System Verilog

VHDL is the adapted hardware description languages for the Department of Defense (DoD). For this thesis we will code our design in VHDL.

2.2.4 RTL Verification. Step four in the VLSI Design Process is RTL Verification. RTL Verification is when you verify that your VHDL code simulates correctly. The simulation waveform generated in Modelsim® gives you a waveform to show the circuit is functioning. If this waveform gives you the expected results then your code and RTL are working properly.

2.2.5 Synthesis/Manual Layout. A flexible synthesis tool, such as Leonardo Spectrum by Mentor Graphics® Tool, allows a synthesizable HDL design to be synthesized for both FPGA and ASIC. The software tool uses optimization algorithms to determine the best floor design, place, and route of the design. The synthesis step has made it possible for an FPGA to only need the design to be download to the FPGA board to verify whether the simulation results match the synthesis results and meet the specifications that were stated at the beginning of the VLSI Design Process. Unlike the FPGA design, after the ASIC design is synthesized, it still needs to be sent to a foundry to be fabricated. The use of FPGA eliminates the fabrication step, it is economically inexpensive to own and can be reconfigured for a new design in a matter of seconds, depending on the size of the design. Therefore, if there was a problem with the timing of the design and it didn’t meet the specifications of the design, the
designer would be aware of the problem and begin to correct the timing issue. However, the ASIC design wouldn’t find this problem until the design was tested after it was received from the foundry.

When developing a custom design, a manual layout could be used. This would require you to hand place every piece of the circuit in a location that is determined by you. Then you would have to route all the cell wires together. This is a very tedious and time consuming process. The designer has to worry about design rule violations such as placing the cells too close together or too far apart causing timing delays. An example of a handmade design of a Nand Gate is seen in Figure 2.3. This Nand Gate is only one gate. If you have a complicated design, in the range of hundreds of millions of gates, you can see how this task can be extremely time consuming. Once the gates have been layed out, then you have to determine if they are in the optimal position for the place and route step. This design lacks scalability when new technologies are created, the old layout may not be usable at all in ASICs. However, ASIC designs will have a higher absolute performance than FPGA designs.

2.2.6 Implementation/Fabrication. Implementation/Fabrication is the final step in the VLSI Design Process. Implementation/Fabrication consists of the design being synthesized for an ASIC design, otherwise it would be a manual layout design. The design is finalized and the circuit is sent on as a computer file to the foundry for
the fabrication process. At the foundry, several different metal layers and polysilicon layers will be used to create the design. The design will also need a set of test vectors to be sent with it to the foundry. This is required to determine if the design is working correctly. If the design comes back correct, then this ASIC or custom design can only be used for this given application. On the other hand, if the design comes back not working properly, then the designer will have to analyze the circuit and determine where in the six step process the design failed. This basically leaves the designer at square one. All of this time has been invested into a product that does not work. Even if the change is found within a few days it will still be 3 to 6 months from the time they send it to the foundry before they will see it again. Therefore, making an ASIC design is very costly and time consuming for the design process. Unlike the ASIC, the FPGA can be designed and implemented in a shorter time span and is an economical, reusable, and reconfigurable product.

2.3 FPGA vs. ASIC Risk

Now that the VLSI design cycle has been thoroughly discussed, let us examine the risk that is involved with building a circuit to meet an FPGA or ASIC design. The FPGA is an inexpensive design option that will only cost a few thousand dollars for a board. The Static Random Access Memory (SRAM) FPGA is reconfigurable in that once you put one design on the board you can take the program off the board and reprogram the board for a completely new design with no new costs involved. Once the design is synthesizable and thoroughly tested, it requires less time to become operational. There is also a higher chance that your FPGA design will synthesize and work the first time, unlike the ASIC design, saving a company time and money. On the other hand, an ASIC design is an expensive investment and will require substantially more time and money to be implemented. For example, let us say that a company is implementing a new design and they choose to take the route of an ASIC design. They could invest a million dollars into the program before it is even fabricated due to the labor intensive effort required. If the design fails once, it comes back from fabrication
to engineers who will thoroughly look at the design to understand why it failed. Once the failure has been determined, it may cost an additional $100,000 from the time the engineers determine the problem to the time it is refabricated and works. Also, this design can only be used for this single application. If this application was for a satellite that was in space and it failed, the satellite would lose that functionality in the system. However, if the design was on an FPGA, the design could be reprogrammed in a matter of minutes to keep the system functioning. As you can see, there are several risks that need to be considered when deciding what design solution to use.

In this thesis, we will create a set of synthesizable VHDL libraries targeting FPGAs and ASICs. However, we are putting emphasis on the FPGAs due to extra time and cost required for us to fabricate ASIC solutions for validation purposes.

2.4 MSP Design Reusability

Now you have seen the design steps required for a VLSI circuit design and how there are two different approaches to reach a solution. MSP offers design reusability by creating multiple modules of the same functioning task that focus on optimizing power, area, and delay modules. A typical design takes an average of 24 months to complete. The MSP design can take 3 months on average to complete, due to the synthesizable reusable libraries. Each design is unique to itself with different variables that depend on the length of time required to complete a design. Some of the variables that need to be considered are as follows: complexity factor, can you reuse anything from a previous design, is the task fully defined, experience of the designer, etc. A typical design timeline can be seen in Figure 2.4. The MSP libraries include the Architecture, RTL Coding, RTL Verification, and Synthesis. Using these MSP libraries can save time, money, and resources when designing a circuit. Chang and Aguan [3] state how important it is to have reusable VHDL modules in their journal article “Design-for-reusability” in VHDL.
The reuse of electronic components can improve productivity in system design. However, without careful planning, components are rarely designed for reuse.

There is also more than one option when designing a multiplier, adder, or controller. You can build a behavior combinational multiplier, structurally combinational multiplier, behavior sequential multiplier, structurally sequential multiplier, or even a Booth multiplier just to name a few. They all have strengths and weaknesses for a design. The MSP idea is to build reusable libraries to meet the needs of the designers. A multiplier is a good example of this. The designer can build three variations of a multiplier to have one optimized for power, one for area, and one for delay (speed). When a new requirement is generated and it requires the use of a multiplier, the reusable libraries of power, area, and delay will be available. Since these modules have already been optimized, there is no additional work for the designer. This concept goes back to the the power, area, and delay triangle that was shown in Figure 1.1. The pre-built modules make it possible for the designer to only have to choose which module will work best for the specifications they were given. This can be seen in Table 2.1 where there are four types of multiplier designs showing their results for power, area, and delay. These results came from the software tool Cadence®. The
Table 2.1: Summary of Maximum Clock Frequency for Modules

<table>
<thead>
<tr>
<th>Type of Multiplier</th>
<th>Power mW</th>
<th>Area mm$^2$</th>
<th>Max. Freq. MHz</th>
<th>Constrained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booth</td>
<td>37.17</td>
<td>0.1537</td>
<td>53.99</td>
<td>Yes</td>
</tr>
<tr>
<td>Combinational Behavior</td>
<td>7.35</td>
<td>0.2043</td>
<td>53.09</td>
<td>No</td>
</tr>
<tr>
<td>Combinational Structural</td>
<td>9.13</td>
<td>0.2422</td>
<td>28.23</td>
<td>No</td>
</tr>
<tr>
<td>Sequential Behavior</td>
<td>64.87</td>
<td>0.2151</td>
<td>37.31</td>
<td>Yes</td>
</tr>
</tbody>
</table>

constrained column in the table gives a clear picture of what power, area, and maximum clock frequency can be achieved. When combinational designs are synthesized in Cadence® there is no clock used to determine how fast the design will run accurately. Therefore, Cadence® gives a best guess estimate for the power, area, and delay. As you can see, the Booth Multiplier has the fastest delay but the third largest amount of power consumption. One multiplier design may have the smallest area, but the slowest delay. It is up to the designer to pick which module will work best for their application.

If you take this a step further and expand to other modules like adders, subtractors, and controllers, you will have generated an arsenal of modules that have been optimized for power, area, and delay. This will be like picking out a new car. What options do you want for a new car - power windows, cd player, full size spare tire, etc.? Having these libraries built, reconfigurable, and reusable makes designing a weapon system simpler and saves time, money, and resources. Also, using an FPGA makes it easier to upgrade the system at a lower cost while applying less time. There will be several different variations of the same module focusing on optimized power, area, and delay. For example, if a design calls for fast multiplications to be performed, the designer has the option to choose the multiplier that has been optimized for speed from the reusable libraries that are already built.

In this thesis we will be implementing a Booth multiplier. The Booth multiplier is based on adding, subtracting, and shifting the binary values several times. Andrew Booth [5] noticed this can be achieved by having a lookup table to determine if the binary value needs to be added, subtracted, or do no operation and only shift the
binary values one place. The Booth algorithm is summarized in points 1-3 below [5].

By using a lookup table, it is possible to quickly determine what operation needs to be performed. The best thing about the Booth multiplier is the option it has to perform no operation if there is a run of “00” or “11” in the number being multiplied. This do nothing option reduces the number of operations, saving clock cycles required to compute the multiplication. For example, if the bit width is 32 bits, it will only require 16 clock cycles to complete the multiplication. For this thesis we will implement a 2 bit shift Booth multiplier to cut the clock cycles in half from the given number of input bits.

Booth Multiplier Algorithm Summary:

1. Examine each pair of digits in the multiplier, creating the first pair by appending a dummy ‘0’ at the least significant end.

   If the pair is 01, add the multiplicand.

   If the pair is 10, subtract the multiplicand.

   Otherwise, do nothing.

2. Shift both partial product and multiplier one place to the right, allowing the next pair of digits to be examined.

3. Repeat as many times as there are digits in the multiplier [5].

2.5 Optical Flow

The Optical Flow algorithm compares two images together to see what is different between the two images. Some applications where optical flow are used are change detection, computer vision, pattern recognition, tracking targets, and image processing. Optical Flow is best described by Horn and Schunck [7] in their paper, “Determining Optical Flow.”

Optical flow is the distribution of apparent velocities of movement of brightness patterns in an image. Optical flow can arise from relative mo-
tion of objects and the viewer. Consequently, optical flow can give important information about the spatial arrangement of the objects viewed and the rate of change of this arrangement.

There are two [7] [10] well regarded methods for calculating optical flow - are Horn-Schunck and Lucas-Kanade methods. The Horn-Schunck [7] method looks at the difference between the brightness and contrasts between the two images to estimate what changes have occurred. The changes are represented in a vector field to show the direction of motion the image is moving compared to the first image. The Lucas-Kanade method [10] makes use of the “spatial intensity gradient of the images.” Using the Horn-Schunck or the Lucas-Kanade method requires large amounts of calculations to determine if there is a change between the first image and the second image. In this project, we will look at the Lucas-Kanade method to determine an optical flow solution between two images. An example of a vector field that was produced between two images using the Lucas-Kanade method is shown in Figure 2.5. These results were attained from Matlab® comparing two images that are $250 \times 400$ pixels.
2.6 Chapter Summary

The background information for Chapter 2 covered the following: VLSI Design Process, FPGA vs. ASIC Risk, MSP Design Reusability, and Optical Flow. Several low power, area, and delay (speed) implementations were also looked at and discussed in the architecture section of the VLSI Design Flow section. MSP has shown the importance of creating many different modules for optimized power, area, and delay that will perform the same task. Two different methods to perform Optical Flow have been discussed, which were the Lucas-Kanade and Horn-Schunck Optical Flow methods. We will build reusable libraries in VHDL for the Lucas-Kanade method so the libraries can later build a complete Lucas-Kanade Optical Flow system. This image processing can be used in part of the target tracking project. These ideas will be discussed further in Chapter 3 to develop this thesis project.
III. Methodology

The methodology used to convert the Matlab® commands that generated the Dense Optical Flow will be discussed in this chapter. The research goal is to convert the Matlab® commands, located in Appendix B, used in the Dense Optical Flow to build reusable VHDL libraries such as, Conv2, Matrix Transpose, Round, Floor, and Pinv, etc. The goal is to demonstrate functioning MSP modules that are reusable for image processing such as Optical Flow, DSP, computer vision, pattern recognition, tracking targets, change detection, etc. These MSP libraries will be the key for the foundation of these applications.

The Dense Optical Flow Matlab® command functions will be created and demonstrated using smaller modules. These modules will make up the parts for the Matlab® command that can later be used to build an Optical Flow system using MSP modules. An example of some of the Matlab® commands that are used in Optical Flow code can be seen in Listing III.1.

Listing III.1:

```
1  % Example of using the conv2 command
2  hResult = conv2(im, mask);
3
4  % Example of using the matrix transpose command
5  curFx = curFx';
6
7  % Example of using the Pinv command
8  U = pinv(A'*A)*A'*curFt;
9
10 % Example of using the round command
11 uIn = round(uIn);
12
13 % Example of using the floor command
14 halfWindow = floor(windowSize/2);
```

To create these commands, smaller modules were designed around the Matlab® Dense Optical Flow written by Sohaib Khan [8] using the Lucas-Kanade algorithm [10].
The reason for creating individual modules and making them configurable components is to be able to use a generic module and shape it to meet the needs of a specific design. We will assume that the image sizes will not change while the data is being recorded with a camera and the modules will be set to receive grayscale images. This parameter is configurable to meet future needs. The configurability parameters of many of the individual modules can be seen in Table 3.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address_width</td>
<td>Bit width of matrix size (row, column)</td>
</tr>
<tr>
<td>data_width</td>
<td>Bit width of pixel depth values</td>
</tr>
</tbody>
</table>

### 3.1 Overall Design

The overall design of the Dense Optical Flow was broken into smaller parts called modules. These modules were designed to take on the same characteristics as the Matlab® commands. There are two different designs for the Reduce Matrix Function that will be developed in VHDL. These two designs will look at large area and small power consumption between the designs. These modules were built using VHDL and tested with test benches to prove that they functioned properly. Other Matlab® commands that will be created in VHDL are the following: Conv2, Round, Floor, Matrix Transpose, Compute Derivatives, Reduce Matrix Function, Adder, Multiplier, Divider, Subtractor, and Pinv. Also, the goal is to implement the Matlab® functions as VHDL modules. For instance, the 2×2 matrix called A has the following parameters while the 2×2 matrix called B has the following parameters:

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\quad \begin{bmatrix}
5 & 6 \\
7 & 8
\end{bmatrix}
\]

The following Matlab® code calculates the two-dimensional convolution of Matrices A and B as seen in Equation 3.1. The resulting answer is stored in the 3×3 Matrix C.
Matrix C produces the following result:

\[
\begin{pmatrix}
5 & 16 & 12 \\
22 & 60 & 40 \\
21 & 52 & 32 \\
\end{pmatrix}
\]

These results match the results from the VHDL convolution module with 0% error. The other \texttt{Matlab®} commands or functions were converted to VHDL to build the reusable module libraries to lay the foundation for Dense Optical Flow algorithm. The \texttt{Matlab®} commands that are only specific to \texttt{Matlab®} were developed and function properly include: Conv2, Matrix Transpose, Round, Floor, and Pinv.

### 3.2 Two Dimensional Convolution

The two dimensional convolution \texttt{Matlab®} command Conv2 can be seen in Equation 3.1. The Conv2 to \texttt{Matlab®} command is the backbone for the image processing that has to take place in an application such as Optical Flow. Other \texttt{Matlab®} functions that were developed by Sohaib Khan, such as a Reduce Function and Computing Derivatives, can not be developed without the use of the Conv2 command. Therefore, the Conv2 command was the first module that was created.

Matrix C is created from the size of Matrix A and B. For example, Matrix C is \([D+F-1, E+G-1]\). These parameters come from the size of Matrix A\([D,E]\) and Matrix B\([F,G]\). The algorithm for the Conv2 can be seen in Equation 3.2.

\[
C(n_1, n_2) = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} a(k_1, k_2)b(n_1 - k_1, n_2 - k_2) 
\]

From the \texttt{Matlab®} Conv2 algorithm, VHDL modules were written and implemented having the same characteristics as the \texttt{Matlab®} command using a state
machine to take the place of the nested for-loops. The Conv2 Matlab® command requires the following modules to be built in VHDL in order for it to function:

1. Memory Module
2. Multiplier Module
3. Control Module
4. Multiplexer Module
5. Adder Module
6. Register Module

Once these modules are built, they will be connected together, like connecting Legos® together, to build the Conv2 Module for VHDL. Simulations will be run to test the VHDL Conv2 and the results will be compared to the Matlab® Conv2 command.

3.2.1 Memory. Picture images are made up of grayscale values between 0-255 integer values that makeup the brightness of the image. These values are used to discern between adjacent pixel colors of the images. A picture image is nothing more than a two-dimensional matrix that has rows and columns and is loaded with integer values to represent the brightness of the pixels. The memory method that will be used is to initially generate a matrix larger than the image that is going to have the Conv2 command performed on it. This is required when the Conv2 command is finished computing, the resultant matrix is larger than the initial image matrix size. Therefore, to take into account the matrix being larger than the initial image size, the memory module will initially be filled with all zero’s. This oversized matrix eliminated the problem of the Conv2 command from stepping out of bounds when computing the results for the Conv2 command. The matrix can then be filled with the required grayscale values that represent the image.

3.2.2 Multiplier. The initial Multiplier Module that will be developed will only be a basic behavior model. The second Multiplier Module that will be developed
will be a 2-bit Booth Multiplier. This multiplier will be able to cut the clock cycle in half for calculating the results. For instance, the data input for the pixel value for the image is 8 bits, but we need to take into account the fractional parts of numbers that are multiplied together. The Matlab® Optical Flow uses a $1 \times 5$ mask matrix that is made up of a Gaussian distribution. The Mask $1 \times 5$ matrix has the following values:

$$\text{Mask Matrix} \begin{pmatrix} .05 & .25 & .4 & .25 & .05 \end{pmatrix}$$

When these calculations are performed, the results give a fractional part to the resultant. Therefore, we will use fixed-point notation to represent the fractional part of the number. In doing so, we will allocate 16 bits for the fractional part to represent the Gaussian distribution trying to minimize the error that will be caused since the values .05 or .4 will be only a close approximation of these values. This leaves us with 16 bits for the fraction part and 8 bits for the grayscale images yielding a total of 24 bits required to represent the images. We will initially set the image data width to 24 bits. However, this parameter is configurable to a larger bit width, if required.

3.2.3 Control. The Control Module could be implemented using a for-loop but these loops cannot typically be used for algorithmic iterations in synthesizable code [9]. Therefore, a state machine will be built to unroll the iterative loop that will increase throughput for the design.

3.2.4 Multiplexer, Adder, Register. The Multiplexer Module is used to switch between the loading of images and the Control Module, and is connected to the Memory Module. Once the image is loaded into the Memory, the Multiplexer will switch from a load mode to only communicate with the control logic. The Adder Module is used to add the values together once the multiplication is completed for a set of values. The Conv2 Matlab® commands require the sum of matrix positions to be multiplied and added together. The Register Module locks in the values that have been added together into its register. Once all the calculations have been completed,
the Control Module will send a signal to Matrix C so the data in the Register Module can be written into Matrix C at location (0,0).

### 3.3 Matrix Transpose

Part of the Matlab® code transposes the input image. The following modules are required to build the Matrix Transpose Module:

1. Memory Module
2. Control Module
3. Multiplexer Module

Since the goal of the MSP is to create reusable libraries, creating the Matrix Transpose Module has been simplified because the Memory Module was already developed while creating the Conv2 command. Therefore, the only new module required to be built is a state machine Control Module to transpose a matrix.

### 3.4 Compute Derivatives

Another Matlab® function module that needs to be developed is one that is able to compute derivatives. The Fx and Fy derivatives are used to determine the edge detection of an object that has moved between Image1 and Image2. The Ft derivative is the summation of Image1 and Image2. You can picture it as placing Image2 on top of Image1, which forms the Ft derivative. The Compute Derivatives Module is built around the Matlab® Conv2 command. Again, the MSP reusable libraries make it possible for the creation of the Compute Derivative function by being able to recycle the Memory and Conv2 modules. The equation seen in Equation 3.3 is from the Compute Derivatives function that was written in Matlab®. This equation requires the Conv2 command to be used twice to multiply Image1 by a fixed 2 × 2 matrix and added together with the result from Image2 being Conv2 with a fixed 2 × 2 matrix.

\[
Fx = Conv2(Image1, 0.25[-11; -11]) + Conv2(Image2, 0.25[-11; -11]); \quad (3.3)
\]
The following modules are required to build the Compute Derivative Module:

1. Memory Module
2. Control Module
3. Conv2 Module

The Matlab® Compute Derivatives Function code can be seen in Listing III.2.

Listing III.2:

```matlab
[fx, fy, ft] = ComputeDerivatives(im1, im2);

function [fx, fy, ft] = ComputeDerivatives(im1, im2);
%
function ComputeDerivatives Compute horizontal, vertical and time derivative between two gray-level images.
%
if (size(im1,1) ~= size(im2,1)) | (size(im1,2) ~= size(im2,2))
    error('input images are not the same size');
end;

if (size(im1,3)~=1) | (size(im2,3)~=1)
    error('method only works for gray-level images');
end;

fx = conv2(im1,0.25*[-1 1; -1 1]) + conv2(im2, 0.25*[-1 1; -1 1]);
fy = conv2(im1,0.25*[-1 -1; 1 1]) + conv2(im2, 0.25*[-1 -1; 1 1]);
ft = conv2(im1,0.25*ones(2)) + conv2(im2, -0.25*ones(2));
%
% make same size as input
fx=fx(1:size(fx,1)-1, 1:size(fx,2)-1);
fy=fy(1:size(fy,1)-1, 1:size(fy,2)-1);
ft=ft(1:size(ft,1)-1, 1:size(ft,2)-1);
```
3.5 **Reduce Matrix Function**

When dealing with image processing, the images need to be down sampled in size. This down sampling is part of the Optical Flow. We will use the terminology Reduce Matrix Function instead of down sampling. The Reduce Matrix Function is another *Matlab*® function that needs to be created in VHDL. The Reduce Matrix Function Module takes the initial image and reduces it in size by one half. For example, an initial image size of $250 \times 400$ will be $125 \times 200$ after performing the Reduce Matrix Function. Here is another example of how the MSP design thought process uses the created reusable libraries to develop another module. The key module in image processing is the Conv2 Module; again, it is used to build another module. The following modules are required to build the Reduce Matrix Function Module:

1. Conv2 Module
2. Control Module
3. Reduce Control Horizontal Module
4. Reduce Control Vertical Module
5. Memory Module
6. Multiplexer Module
7. Adder Module
8. Register Module

The *Matlab*® Reduce Matrix Function code can been seen in Listing III.3.

Listing III.3:

```matlab
function smallIm = Reduce(im)
%
REDUCE Compute smaller layer of Gaussian Pyramid

% Sohaib Khan, Feb 16, 2000
%Algo
```
The Round command is available in VHDL, but the Floor command is not. The Round command takes an input and rounds it up or down depending on the fractional part of the number. For example, if the number is 4.5, the Round command will round the number to 5 and if the number is 4.49 it will round the number to 4. On the other hand, the Floor command will take the floor of a number. For example, if the value is 5.9 it will floor the value to 5.

3.7 Pseudoinverse (Pinv)

The Pseudoinverse, also known as Pinv, is used to calculate the inverse of a matrix. Not all matrices have an inverse, therefore, the Pseudoinverse is used to find a close matrix inverse for a matrix. The Pseudoinverse [12] Equation 3.4 is the equation used to calculate the Pseudoinverse for Matrix A and the answer is stored in Y. The Matlab® code uses the equation seen in Equation 3.5 to do this.
calculation. The method used to build the Pseudoinverse Module will be to use the Matrix Transpose module and multiply it by the original Matrix A. While that is being done, we will multiply Matrix Transpose by $B_{Vector}$, where the $B_{Vector}$ is the Ft derivative. Doing these two multiplications in parallel, we will perform the matrix inverse on the resultants using the LU-Factorization method [6] to find the inverse. The Pseudoinverse Module will include the following modules for its development:

$$Y = (A^T A)^{-1} A^T$$  \hspace{1cm} (3.4)

$$Y = (A^T A)^{-1}(A^T B_{Vector})$$  \hspace{1cm} (3.5)

1. Memory Module
2. Control Module
3. Multiplier Module
4. Divider Module
5. Adder Module
6. Matrix Transpose Module
7. Register Module

### 3.8 Synthesis/Timing

The Xilinx® Virtex-4 SX ML402 FPGA will be the target FPGA to attain the power, area, and delay of the circuits. The Virtex-4 is a generalized moderate cost FPGA that is a practical standard FPGA for DSP applications. The Virtex-4 ML402 is a good evaluation board with a wide range of applications such as DSP and low power. The Precision RTL® RTL 2007a.8 and Xilinx® ISE 9.2 are the two synthesis tools that will be used.

In one method, the two dimensional convolution module is built using sequential and combinational modules that make up the Conv2. The Conv2 module does many
multiplications and additions to compute the result for Matrix C. These sub-modules are built using combinational logic. Therefore, if the clock runs too fast, the multiplications and additions will not be completed before the next clock cycle, causing the wrong result to be stored in Matrix C. One of the ways to eliminate this problem is to lower the clock speed.

A second method would be to place a pipeline register between the multiplier and adder to increase throughput by increasing the clock speed. Timing is important for Dense Optical Flow due to the number of calculations required to complete two simple $2 \times 2$ matrices to create a $3 \times 3$ matrix. Timing simulations will show how fast the system can run to increase throughput. Timing simulations can also determine if it is required to add additional pipeline registers to increase throughput. The timing simulations can also determine how slow the clock can run without including additional registers or hardware. By slowing the clock speed to a minimal speed, power can be saved in the design. All these things rely on the timing simulations. The Mentor Graphics® tools will also play a role in attaining the results for Chapter 4 of this thesis project.

3.9 Testing Procedure

There are several ways to perform testing on these modules that will be developed. First, Modelsim® version 6.3c will be used to develop the VHDL code. The Modelsim® software will also be used to generate test benches to show the functionality of modules. Once the VHDL modules are functioning, they will be run through the Precision RTL® 2007a.8 and Xilinx® ISE 9.2 software programs for synthesis to look at power, area, and delay of the circuits. This will give an estimate of how to reduce high power draws that occur due to the high amounts of calculations required by Dense Optical Flow. This may require that the circuit clock speeds be slowed down to reduce power, which in turn can minimize circuit size and slow the speeds of the system being tested.
3.10 Chapter Summary

The idea of breaking the Matlab® commands and functions into synthesizable reusable library modules that are used in image processing were discussed. These modules will support AFRL/RY later when they want to create the Optical Flow as a system by taking the synthesizable reusable libraries and connecting them together to build an Optical Flow that can be fully designed in VHDL. The synthesis, timing, and testing procedures were introduced and discussed. The software that will be used to carry out this methodology was discussed and how it will be used. Each submodule was thoroughly tested to ensure there will be no errors when final assembly of the sub-modules are connected to create an MSP module such as a Conv2, Matrix Transpose, Round, Floor, and Pinv, etc.
IV. Analysis and Results

This chapter will discuss the overview design of the MSP modules that were discussed in the methodology section. We will also look at the power, area, and delay for the MSP modules. We will also look at the error analysis calculated between the Matlab® code and the VHDL modules.

4.1 Overview of MSP Modules

The overarching module that needed to be created was the Conv2 Module. This module was used in the development of the Reduce Matrix Function and Computing Derivatives. The Top-Level Design 1 for the Reduce Matrix Function can be seen in Figure 4.1. The Reduce Matrix Function requires the use of Conv2 Module to perform the Conv2 Matlab® command. When using the Conv2 command in Matlab® the matrix size grows larger due to the Conv2 algorithm. The hControl/vControl Reduce Module are used to reduce the oversized matrix that is created when Conv2 is performed. The Reduce Matrix Function is used to reduce a matrix size in half. For example, an image the size of $6 \times 8$ reduces to a $3 \times 4$. The Conv2 Module will generate a $6 \times 12$. The hControl Reduce Module then reduces the $6 \times 12$ matrix down to $6 \times 4$. The Conv2 Module will generate a $10 \times 4$ which will be reduced to $3 \times 4$ using the vControl Reduce Module. The final image is stored in the Memory Module as a $3 \times 4$.

Figure 4.1: Reduce Matrix Function Top Level Design 1
The modules that were used in the Top-Level Design 1 for the Conv2 Module can be seen in Figure 4.2. These modules make up the Conv2 Module. The first Conv2 Module uses a $1 \times 5$ Mask matrix opposed to the $5 \times 1$ Mask matrix that is used in the second Conv2 Module. The Conv2 Module design uses Multiplexer, Memory, Multiplier and Controller Modules. They are connected together to achieve the result for Conv2 Matlab\textsuperscript{®} command.

The Behavior Multiplier can be seen in Figure 4.3. The three modules, Behavior Multiplier, Adder, and Register are required in the development of the Conv2 command.

The modules that are required to build a 2-bit Booth Multiplier Module can be seen in Figure 4.4. The Booth Multiplier Module requires the input values to be loaded into the Multiplicand and Product Module. The Control Module is used to communicate between the other modules.
The Top-Level Design 2 that was used to develop the Reduce Matrix Function can be seen in Figure 4.5. This design approach eliminates the use of a second Multiplier Module reducing power consumption and minimizing area compared to Top-Level Design 1. Further analysis will be discussed later in Chapter 4 to see the differences between the two design methods. Every large scale design requires a Control Module to communicate between the other modules. The Control Module is basically the “brains” for the module.
The Control Module that was used for the Behavior Multiplier Module uses seven states for its state machine. The Control Module is used to communicate between the Multiplier Module and the other modules. The seven states that the Control Module uses to communicate with the Behavior Multiplier Module can be seen in Figure 4.6. The Booth Multiplier, on the other hand, requires that the Control Module be built using 24 states for its state machine if using a 32 bit input. The Booth Multiplier state machine can be seen in Figure 4.7. The Booth Multiplier is performing its shifts, adds, and subtraction operations during States S4-S20 for a 32 bit number. If the input number was to be reduced to 24 bits, it would require 4 less states for the Controller to use.

The Compute Derivatives Top-Level Design can be seen in Figure 4.8. Again, you can see how the Conv2 Module was used to build the Compute Derivatives Module. This function requires the use of the Conv2, Controller, Adder, and a Memory.
Figure 4.8: Compute Derivatives Top-Level Design

to build the derivative module in VHDL. The MSP Design effort payed off in the creation of the MSP Compute Derivatives Modules by being able to reuse previously designed modules.

The Matrix Transpose Module requires the use of a Multiplexer, Controller, and Memory. The Matrix Transpose Module receives data through the Multiplexer which loads the Memory. The Controller is used to transpose the values from the Memory to Memory Transpose. The Top-Level design of the Matrix Transpose Module can be seen in Figure 4.9.

Figure 4.9: Matrix Transpose Top-Level Design
The Pseudoinverse (Pinv) Module can be broken into 3 Modules. The \((A^T A)^{-1}\) is one module, the \(A^T B_{\text{Vector}}\) the other module, and finally the module that will perform the LU-Factorization. The Pinv Module Top-Level Design can be seen Figure 4.10.

4.2 Error Analysis

When creating the VHDL modules to have the same functionality as the Matlab® commands and functions, error is introduced. Part of this error is introduced and can be seen from the Mask Matrix \(1 \times 5\) or its transpose \(5 \times 1\) that have the Gaussian Distributions values in the Memory Matrix. The values \(0.05\) and \(0.40\) can not exactly be represented in binary numbers when only using a limited amount of bits to represent these values. We used 16 bits to represent the fractional part of the Mask Matrix values. Therefore, when the Conv2 Module multiplies the image by the mask, the resultant will have error. This is due to the fact that the Mask Values were not being exactly compared to the Matlab® values. The close approximation values that were used for the Mask Matrix and Matlab® values can be seen in Table 4.1. The error analysis equation that will be used is \(\%\text{Error} = \frac{\text{VHDL}-\text{MATLAB}}{\text{MATLAB}} \times 100\).

The error analysis that was shown for the Reduce Matrix Function for the VHDL compared to the Matlab® can be seen in Figure 4.11. The largest amount of error
that occurred was .0059% for the sample $6 \times 8$ data set that we used. The $6 \times 8$ data set that was used can be seen in the Input Data Matrix. This data set was arbitrarily chosen. The Reduce Matrix Function reduces the $6 \times 8$ matrix down to $3 \times 4$, but not before the $6 \times 8$ has the Conv2 Module performed on it. The x-axis represents the positions in the $3 \times 4$ matrix. The y-axis shows the percent error between the Matlab® output and the VHDL output.

![Percent Difference Between Matlab Output & VHDL Output](image)

**Table 4.1: Comparison between Matlab® and VHDL Gaussian Distribution**

<table>
<thead>
<tr>
<th>Matlab® Gaussian Distribution</th>
<th>0.05</th>
<th>0.25</th>
<th>0.4</th>
<th>0.25</th>
<th>0.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>VHDL Gaussian Distribution</td>
<td>0.049987793</td>
<td>0.25</td>
<td>0.399993896</td>
<td>0.25</td>
<td>0.049987793</td>
</tr>
<tr>
<td>% Error Difference</td>
<td>0.024414</td>
<td>0</td>
<td>0.001526</td>
<td>0</td>
<td>0.024414</td>
</tr>
</tbody>
</table>

The Compute Derivatives for Fx, Fy, and Ft had zero percent error compared to the Matlab® code. This is due to the fact that the Compute Derivative equation
used a positive or negative $2 \times 2$ matrix with the values .25 which can be represented in binary with zero percent error. Also, the Matrix Transpose Module had zero error because it took the original Matrix $A$ and transposed it to become Matrix $A^T$. There was no loss of data precision when transposing Matrix $A$.

The sample test data set that was used to test the Pseudoinverse Module was Matrix $A$ $9 \times 2$ and Matrix $B_{vector}$ $9 \times 1$. The comparison between Matlab® and VHDL results can be seen in Table 4.2. The results are stored in a $2 \times 1$ matrix. The largest percent error from the sample test data was 2.65%. A small amount of error occurs from the divider that is used to calculate the Pseudoinverse. This small error gets magnified when the value is multiplied. This intermediate result is divided and multiplied again as part of the LU-Factorization in order to find the Pseudoinverse.

$$
\begin{pmatrix}
6 & 8 \\
1 & 3 \\
10 & 12 \\
5 & 7 \\
15 & 14 \\
4 & 3 \\
12 & 13 \\
0 & 11 \\
2 & 17
\end{pmatrix}
$$

$$
\begin{pmatrix}
1 \\
5 \\
7 \\
9 \\
10 \\
15 \\
20 \\
2 \\
8
\end{pmatrix}
$$

Table 4.2: Comparison between Matlab® and VHDL Pseudoinverse Module

<table>
<thead>
<tr>
<th></th>
<th>Matrix Position (1, 1)</th>
<th>Matrix Position (2, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matlab® Results</td>
<td>0.731775</td>
<td>0.305488</td>
</tr>
<tr>
<td>VHDL Results</td>
<td>0.716705</td>
<td>0.313827</td>
</tr>
<tr>
<td>% Error Difference</td>
<td>2.10</td>
<td>2.65</td>
</tr>
</tbody>
</table>
4.3 Power, Area, and Delay

In the power, area, and delay section we will look at the results that were attained from the MSP modules implemented using the Cadence® synthesis tool. The Cadence® software tool did not specifically target an FPGA or an ASIC for the results that were attained.

First, we will look at the results that were attained from the Reduce Matrix Function, seen in Figure 4.12. From the excel chart, it is clear that the Top-Level Design 2 approach requires less net power to be used for the Reduce Matrix Function Module. The power savings that occurs for Top-Level Design 2 over Top-Level Design 1 is at least 2.8 times smaller. In Figure 4.12 the Cadence® software also stated that the fastest delay for Top-Level Design 1 using a behavior Multiplier Module ran at 52.02 MHz. The other designs ran at 53.86 MHz.

In Figure 4.13 it is clear that Top Level Design 2 uses a smaller area in the design, up to 54.6% smaller, compared to the Behavior Multiplier Top-Level Design 1 and Booth Multiplier Top-Level Design 2.

The results for Fx, Fy, and Ft can be seen in Figure 4.14 for Power vs. Clock Speed using the Cadence® software. As expected, as you increase the clock speed, the net power increases. The clock speeds and area that were attained for the excel charts
Figure 4.13: Reduce Matrix Function Module Cell Area vs. Clock Speed

Figure 4.14: Compute Derivatives Module Net Power (mW) vs. Clock Speed
are only estimations generated from the Cadence® software. Every software package uses different algorithms to attain the power, area, and delay.

The area for the Compute Derivatives was unchanged as the speed increases as shown in Figure 4.15. For example, you are able to achieve the same cell area for 10 MHz design or a 52 MHz design.

The Matrix Transpose Module achieved the results for Power vs. Clock Speed in Figure 4.16. The net power increased as the speed increased which is what is shown in Figure 4.16.
The cell area was unchanged when the clock speed was increased. This can be seen in Figure 4.17.

The Pseudoinverse Module achieved the results for Power vs. Clock Speed in Figure 4.18. The net power increased as the clock speed increased which is what is expected. The cell area was unchanged when the clock speed was increased. This can be seen in Figure 4.19.
4.4 Synthesis

In the synthesis section, all the modules were targeted for the Xilinx® Virtex-4 SX ML402 FPGA that uses 90nm technology. The device used was 4V SX35FF668, and a Speed Grade of −10 was used to attain area and speed for the synthesis. The Xilinx® Virtex-4 board isn’t a state-of-the-art board and therefore is very cost efficient. Without a doubt, high-end FPGA systems will work much faster (2-3 times the speed) than the current one with Configurable Logic Blocks (CLB) to spare. Like any synthesis software tool, each one uses different algorithms for optimal power, area, and delay for designs. That is why the results in this section will not match the results from the Cadence® software since it uses 250 nm technology to attain the area and delay for the modules. The Reduce Matrix Function Module was synthesized using the Precision RTL® software, which is a Mentor Graphics® product.

The Reduce Matrix Function Module for the Behavior Multiplier Top-Level Design 1 results can be seen in Table 4.3. The constraint for the frequency used was 76 MHz. After synthesis, the maximum clock speed/frequency was 76.482 MHz. Therefore, the synthesis had no timing violations. The Reduce Matrix Function used only 10.24% of the FPGA area. There are a total of 15360 CLB slices available for the targeted FPGA.
Table 4.3: FPGA Synthesize Results for Top-Level 1, Behavior Multiplier

<table>
<thead>
<tr>
<th>Resources</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs &amp; Outputs</td>
<td>264</td>
<td>448</td>
<td>59.93</td>
</tr>
<tr>
<td>Global Buffers</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
<tr>
<td>Function Generators</td>
<td>3145</td>
<td>30720</td>
<td>10.24</td>
</tr>
<tr>
<td>CLB Slices</td>
<td>1573</td>
<td>15360</td>
<td>10.24</td>
</tr>
<tr>
<td>Dffs or Latches</td>
<td>2027</td>
<td>31616</td>
<td>6.41</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>5</td>
<td>192</td>
<td>2.60</td>
</tr>
<tr>
<td>DSP48s</td>
<td>10</td>
<td>192</td>
<td>5.21</td>
</tr>
</tbody>
</table>

Table 4.4: FPGA Synthesize Results for Top-Level 2, Behavior Multiplier

<table>
<thead>
<tr>
<th>Resources</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs &amp; Outputs</td>
<td>166</td>
<td>448</td>
<td>37.05</td>
</tr>
<tr>
<td>Global Buffers</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
<tr>
<td>Function Generators</td>
<td>2725</td>
<td>30720</td>
<td>8.87</td>
</tr>
<tr>
<td>CLB Slices</td>
<td>1362</td>
<td>15360</td>
<td>8.87</td>
</tr>
<tr>
<td>Dffs or Latches</td>
<td>1670</td>
<td>31616</td>
<td>5.28</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>4</td>
<td>192</td>
<td>2.08</td>
</tr>
<tr>
<td>DSP48s</td>
<td>8</td>
<td>192</td>
<td>4.17</td>
</tr>
</tbody>
</table>

The Reduce Matrix Function Module for the Behavior Multiplier Top-Level Design 2 results can be seen in Table 4.4. The constraint for the frequency used was 73 MHz. After synthesis, the maximum clock speed/frequency was 73.115 MHz. Therefore, the synthesis had no timing violations. Design 2 only requires 8.87% of the CLB slices saving 15.49% more area than Design 1.

The Reduce Matrix Function Module for the Booth Multiplier Top-Level Design 1 results can be seen in Table 4.5. The constraint for the frequency used was 100 MHz. After synthesis, the maximum clock speed/frequency was 101.75 MHz. Therefore, the synthesis had no timing violations. Design 1 only requires 11.32% of the CLB slices. This results in a 10% increase in area over the behavior design. The Booth design has a 39.1% increase in maximum clock frequency over the Behavior Design 1.

The Reduce Matrix Function Module for the Booth Multiplier Top-Level Design 2 results can be seen in Table 4.6. The constraint for the frequency used was 100 MHz. After synthesis, the maximum clock speed/frequency was 100.422 MHz. Therefore,
Table 4.5: FPGA Synthesize Results for Top-Level 1, Booth Multiplier

<table>
<thead>
<tr>
<th>Resources</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs &amp; Outputs</td>
<td>268</td>
<td>448</td>
<td>59.82</td>
</tr>
<tr>
<td>Global Buffers</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
<tr>
<td>Function Generators</td>
<td>3475</td>
<td>30720</td>
<td>11.31</td>
</tr>
<tr>
<td>CLB Slices</td>
<td>1738</td>
<td>15360</td>
<td>11.32</td>
</tr>
<tr>
<td>Dffs or Latches</td>
<td>2387</td>
<td>31616</td>
<td>7.55</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>5</td>
<td>192</td>
<td>2.60</td>
</tr>
<tr>
<td>DSP48s</td>
<td>6</td>
<td>192</td>
<td>3.13</td>
</tr>
</tbody>
</table>

Table 4.6: FPGA Synthesize Results for Top-Level 2, Booth Multiplier

<table>
<thead>
<tr>
<th>Resources</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs &amp; Outputs</td>
<td>173</td>
<td>448</td>
<td>38.62</td>
</tr>
<tr>
<td>Global Buffers</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
<tr>
<td>Function Generators</td>
<td>2887</td>
<td>30720</td>
<td>9.40</td>
</tr>
<tr>
<td>CLB Slices</td>
<td>1444</td>
<td>15360</td>
<td>9.40</td>
</tr>
<tr>
<td>Dffs or Latches</td>
<td>1867</td>
<td>31616</td>
<td>5.91</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>4</td>
<td>192</td>
<td>2.08</td>
</tr>
<tr>
<td>DSP48s</td>
<td>6</td>
<td>192</td>
<td>3.13</td>
</tr>
</tbody>
</table>

The synthesis had no timing violations. Design 2 only requires 9.40% of the CLB slices. This results in a 6% increase in area over the Behavior Design. Although, the Booth design has a 37.34% increase in maximum clock frequency over the Behavior Design 2. Design 2 for the Booth is 20.36% smaller than Design 1 using a Booth multiplier.

The Compute Derivatives Fx, Fy, and Ft Modules synthesis results can be seen in Table 4.7, Table 4.8, and Table 4.9 respectively. The constraint for Compute Derivative Fx Module had a frequency of 105 MHz. After synthesis, the maximum clock speed/frequency was 105.876 MHz. Therefore, the synthesis had no timing violations.

The constraint for Compute Derivative Fy Module in Table 4.8 had a frequency of 105 MHz. After synthesis, the maximum clock speed/frequency was 105.876 MHz. Therefore, the synthesis had no timing violations.
Table 4.7: FPGA Synthesize Results for Compute Derivative Fx Module

<table>
<thead>
<tr>
<th>Resources</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs &amp; Outputs</td>
<td>106</td>
<td>448</td>
<td>23.66</td>
</tr>
<tr>
<td>Global Buffers</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
<tr>
<td>Function Generators</td>
<td>3439</td>
<td>30720</td>
<td>11.19</td>
</tr>
<tr>
<td>CLB Slices</td>
<td>1720</td>
<td>15360</td>
<td>11.20</td>
</tr>
<tr>
<td>Dffs or Latches</td>
<td>2187</td>
<td>31616</td>
<td>6.92</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>5</td>
<td>192</td>
<td>2.60</td>
</tr>
<tr>
<td>DSP48s</td>
<td>4</td>
<td>192</td>
<td>2.08</td>
</tr>
</tbody>
</table>

Table 4.8: FPGA Synthesize Results for Compute Derivative Fy Module

<table>
<thead>
<tr>
<th>Resources</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs &amp; Outputs</td>
<td>106</td>
<td>448</td>
<td>23.66</td>
</tr>
<tr>
<td>Global Buffers</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
<tr>
<td>Function Generators</td>
<td>3439</td>
<td>30720</td>
<td>11.19</td>
</tr>
<tr>
<td>CLB Slices</td>
<td>1720</td>
<td>15360</td>
<td>11.20</td>
</tr>
<tr>
<td>Dffs or Latches</td>
<td>2187</td>
<td>31616</td>
<td>6.92</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>5</td>
<td>192</td>
<td>2.60</td>
</tr>
<tr>
<td>DSP48s</td>
<td>4</td>
<td>192</td>
<td>2.08</td>
</tr>
</tbody>
</table>

The constraint for Compute Derivative Ft Module in Table 4.9 had a frequency of 101 MHz. After synthesis, the maximum clock speed/frequency was 101.75 MHz. Therefore, the synthesis had no timing violations.

Table 4.9: FPGA Synthesize Results for Compute Derivative Ft Module

<table>
<thead>
<tr>
<th>Resources</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs &amp; Outputs</td>
<td>106</td>
<td>448</td>
<td>23.66</td>
</tr>
<tr>
<td>Global Buffers</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
<tr>
<td>Function Generators</td>
<td>3368</td>
<td>30720</td>
<td>10.96</td>
</tr>
<tr>
<td>CLB Slices</td>
<td>1684</td>
<td>15360</td>
<td>10.96</td>
</tr>
<tr>
<td>Dffs or Latches</td>
<td>2183</td>
<td>31616</td>
<td>6.90</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>5</td>
<td>192</td>
<td>2.60</td>
</tr>
<tr>
<td>DSP48s</td>
<td>4</td>
<td>192</td>
<td>2.08</td>
</tr>
</tbody>
</table>

The constraint for Matrix Transpose Module in Table 4.10 had a frequency of 158 MHz. After synthesis, the maximum clock speed/frequency was 158.856 MHz. Therefore, the synthesis had no timing violations. The Matrix Transpose Module only required 3.22% CLB slices to be used.
Table 4.10: FPGA Synthesize Results for Matrix Transpose Module

<table>
<thead>
<tr>
<th>Resources</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs &amp; Outputs</td>
<td>68</td>
<td>448</td>
<td>15.18</td>
</tr>
<tr>
<td>Global Buffers</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
<tr>
<td>Function Generators</td>
<td>990</td>
<td>30720</td>
<td>3.22</td>
</tr>
<tr>
<td>CLB Slices</td>
<td>495</td>
<td>15360</td>
<td>3.22</td>
</tr>
<tr>
<td>Dffs or Latches</td>
<td>588</td>
<td>31616</td>
<td>1.86</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>2</td>
<td>192</td>
<td>1.04</td>
</tr>
<tr>
<td>DSP48s</td>
<td>2</td>
<td>192</td>
<td>1.04</td>
</tr>
</tbody>
</table>

The constraint for the Floor Module had a frequency of 471 MHz. After synthesis, the maximum clock speed/frequency was 471.032 MHz. Therefore, the synthesis had no timing violations and the results can be seen in Table 4.11. The Floor module only uses .01% of the CLB slices.

Table 4.11: FPGA Synthesize Results for Floor Module

<table>
<thead>
<tr>
<th>Resources</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs &amp; Outputs</td>
<td>9</td>
<td>448</td>
<td>2.01</td>
</tr>
<tr>
<td>Global Buffers</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
<tr>
<td>Function Generators</td>
<td>4</td>
<td>30720</td>
<td>0.01</td>
</tr>
<tr>
<td>CLB Slices</td>
<td>2</td>
<td>15360</td>
<td>0.01</td>
</tr>
<tr>
<td>Dffs or Latches</td>
<td>4</td>
<td>31616</td>
<td>0.01</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>0</td>
<td>192</td>
<td>0.00</td>
</tr>
<tr>
<td>DSP48s</td>
<td>0</td>
<td>192</td>
<td>0.00</td>
</tr>
</tbody>
</table>

The constraint for the Round Module had a frequency of 148 MHz. After synthesis, the maximum clock speed/frequency was 148.236 MHz. Therefore, the synthesis had no timing violations and the results can be seen in Table 4.12. The Round Module uses 3.67% of the CLB slices.

The constraint for the Pseudoinverse Module had a frequency of 33 MHz. After synthesis, the maximum clock speed/frequency was 34.58 MHz. Therefore, the synthesis had no timing violations and the results can be seen in Table 4.13. The Pseudoinverse Module uses 28.27% of the CLB slices.

The summary of maximum clock speed can be seen in Table 4.14. The Precision RTL® software tool showed similar results as the Cadence® software tool. The Reduce
Matrix Function Booth Multiplier Design outperforms the Behavior Multiplier by 39.1%.

The summary of the net power in Watts can be seen in Table 4.15. The **Precision RTL**® software tool showed similar results as the **Cadence**® software tool.
Table 4.15: Summary of Net Power Watts for Modules

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Net Power Watts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce Matrix Function Top-Level 1 Behavior Multiplier</td>
<td>0.672</td>
</tr>
<tr>
<td>Reduce Matrix Function Top-Level 2 Behavior Multiplier</td>
<td>0.662</td>
</tr>
<tr>
<td>Reduce Matrix Function Top-Level 1 Booth Multiplier</td>
<td>0.703</td>
</tr>
<tr>
<td>Reduce Matrix Function Top-Level 2 Booth Multiplier</td>
<td>0.687</td>
</tr>
<tr>
<td>Compute Derivative Fx</td>
<td>0.700</td>
</tr>
<tr>
<td>Compute Derivative Fy</td>
<td>0.708</td>
</tr>
<tr>
<td>Compute Derivative Ft</td>
<td>0.700</td>
</tr>
<tr>
<td>Matrix Transpose</td>
<td>0.668</td>
</tr>
<tr>
<td>Floor</td>
<td>0.636</td>
</tr>
<tr>
<td>Round</td>
<td>0.660</td>
</tr>
<tr>
<td>Pseudoinverse</td>
<td>0.646</td>
</tr>
</tbody>
</table>

4.5 Chapter Summary

In this chapter we discussed the overview of the designs that were used to build the MSP synthesizable reusable libraries. Top-Level Design schematics were used to show how the modules connected together. Error analysis was discussed to see what error was introduced by using the Gaussian Distribution for the Mask Matrix Modules. These modules used values that were close approximations to the Matlab® values. The error results showed only a maximum of .0059% error for our arbitrary data set. The power, area, and delay section used the Cadence® software to attain the results for each module. It was noticed that as the delay increased the power consumption increased. In some cases, the smallest cell area occurred when the delay was at maximum. The synthesis section used the Precision RTL® software to target the Xilinx® Virtex-4 SX ML402 FPGA to attain area and speed for the different modules.
V. Conclusions

5.1 Summary of the Project

The problem was to convert Matlab® commands and functions into VHDL components. These commands are used in image processing such as Optical Flow. To fully implement Lucas-Kanade Optical Flow using these libraries may take up to six months. The foundation has been laid through the synthesizable reusable libraries. The development of these MSP modules have shortened the development time for an FPGA or ASIC design. The MSP modules were implemented, validated, and are fully synthesizable and functional reusable libraries. Chapter 2 discussed the VLSI Design Process and how long it takes to get the design completed. Architecture techniques were used in the development of the MSP libraries. The risks were discussed and developed for why FPGAs are the right solution for getting a design field quickly. Chapter 3 discussed the methodology and design structure that was used to aid in the creation of the MSP libraries. Chapter 4 discussed the analysis and results that were achieved from the design methodology. These libraries make it possible for the designer to save time, money, and resources in the VLSI Design Process. The modules that were created are reconfigurable to meet the needs of the ever-changing Air Force. These modules can be used for future Air Force projects that require change detection, computer vision, pattern recognition, tracking targets, and image processing.

5.2 Future Work

The designs of the MSP Modules were shown to work successfully through simulations and were synthesized through the RTL software Cadence® and Precision RTL® software tools. The Cadence® software focused results on using the 250-nm technology library from the Taiwan Semiconductor Manufacturing Company. The Precision RTL® software tools targeted designs for the Xilinx® Virtex-4 SX ML402 FPGA. This section will discuss three future research topics.
5.2.1 Improve Module Designs. The modules designed are just the beginning. The great thing about this project is that it provided baseline modules that will be able to be enhanced for many projects to come. For example, the Reduce Matrix Function Module could have pipelines and parallelization added to it to improve the delay of the Module. This would require that the Conv2 Module also have additional multipliers, controllers, pipelining, and parallelization. The Memory Module could have enhanced features to accept multiple reads and writes during the same clock cycle. By breaking the Control Module into smaller control units it will assist in the parallelization of how quickly the multiplications can be achieved. This will result in a faster delay, and the area of the design will grow larger due to the additional hardware. These are just a few examples; the possibilities are endless.

5.2.2 Power, Area, and Delay. Optimized power, area, and delay modules can be developed for each Matlab® command or function that makes up the Lucas-Kanade method. These additional modules will give the designer more flexibility and options to choose from for their next design. For example, their arsenal can have three different versions of a multiplier: Multiplier Power Module, Multiplier Area Module, and Multiplier Delay Module. The designer will have the option to mix and match these modules together for future needs of the Air Force.

5.2.3 Complex Numbers. Currently, the modules are based on using fixed-point notation. Enhancements and variations can be made to handle floating point and complex numbers. A floating point number will give more range than using fixed-point. However, complex numbers can open the doors for potential uses such as Digital Signal Process (DSP), image processing, tracking targets, etc. When using DSP applications, complex numbers are used. DSP requires complex numbers to be able to work in the time and frequency domain using Laplace transforms.
Appendix A. Reducing Power Consumption

This section will discuss other ways to reduce power consumption for designing an architecture that requires low power.

A.1 Clock Speed/Clock Gating

One of the six different options used to reduce the amount of power in a circuit is to reduce the speed of the clock. Reducing the speed of the clock for a circuit will reduce the switching activity. This results in power savings by limiting the amount of switching activity that has to take place. Besides reducing the speed of the clock for the circuit, designers have also proposed.

Clock gating by modifying the design of the existing energy recovery clocked flip-flops to incorporate a power saving feature that eliminates any energy loss on the internal clock and other nodes of the flip-flops. Applying this approach of clock gating for a system that has 1000 flip-flops with 50% percent data switching eliminates the total power of the circuit by 47% and also found that during sleep mode the flip-flops reduce their power by (1000x) and has negligible amounts of overhead when the flip-flops are in active mode [13].

The proposed method that achieved these results [13] “inserted the gating feature inside the flip-flops themselves.” This design can be seen in Figure A.1. Figure A.1a is Single-Ended Conditional Capturing Energy Recovery (SCCER), Figure A.1b is Static Differential Energy Recovery (SDER), and Figure A.1c is Differential Conditional Capturing Energy Recovery (DCCER) flip flop.

Another option to reducing the power of the clock is to have an asynchronous circuit design, also known as a clockless design [2]. This design eliminates the use of having a clock saving power. Arsalan and Shams state the following about asynchronous designs [2]:

The asynchronous circuit design has been long regarded as a way for solving the problem of synchronous circuit design such as clock skew, worst case delay, and heavy global clock loading. When designed carefully, asynchronous circuits can be more power efficient as compared to their synchronous counterparts.
A.2 Turning Off The Circuit

A second option to reduce power consumption is to set up a circuit to have processes idle or turned off when not in use. This approach to reducing power consumption can be broken into three parts: timeout-based, predictive, and stochastic [4]. The timeout-based is self explanatory. When a task is not required to be used over a given time period it will timeout until it is required again. The predictive scheme can use an algorithm to predict when the circuit should turn different portions on and off, depending on it’s requirements. The predictive scheme, or partially shutting down the system when it is not in use, will reduce the power consumption. Partial shutdown will put a section of the system into idle or sleep mode depending on the operation that is being performed. If a process is set into idle mode it will wake up without waiting to reboot. If a process is set into sleep mode it will be turned off when not in use. The downsides are that the idle mode will require more power to keep the system readily available and the sleep mode will take longer for the section of the circuit to transition to run mode. The third approach to reducing power consumption is stochastic. This “approach formulates policy optimization of Dynamic Power Management (DPM) as an optimization problem under uncertainty rather than trying to eliminate uncertainty by prediction” [4].
Thirdly, recycling the power in the circuit is another possible solution to reducing power consumption. The switching activity for a circuit is the major cause for power consumption by having the circuit constantly charging and discharging across the load and parasitic capacitance [16]. If you were to recycle the power of the circuit back into it’s voltage supply you could save up to 80% power consumption [16], instead of sending the power discharging back into ground where it will be lost forever. Most circuits use Direct Current (DC) power supplies to turn them on. If you use Alternating Current (AC) power supplies instead you can recycle the power for your circuit. This approach can be seen in Figure A.2 [16] using two diodes and P channel and N channel transistors to keep the load from discharging. As the power source supplies a sinusoidal waveform the signal is 0 volts and the P channel transistor is turned on and begins charging the load capacitor. This causes the diode connected to the P channel to be just below the Peak Voltage (Vp) by one diode. During all this, the N channel transistor is turned off. The diode that is connected to the P channel transistor holds the load voltage to Vp. During the Vp of the sinusoidal waveform the N channel transistor is turned on and the P channel transistor is turned off. The P and N channel transistors are turned off during the negative cycle of the sinusoidal signal [16].
A.4 Redesign Logic Of The Circuit

Another option is to reduce the amount of floating point operations in the circuit by replacing them with fixpoint representation. It’s even possible to have a trade off between parallel and serial logic design for the circuit. Power can be saved by using parallization, pipelining, or a systolic architecture approach. Parallization gives you the option to be able to slow down the clock speed to still compute data through the circuit. The slowing down of the clock speed reduces the clock switching resulting in power savings. For example, a sequential design requires a clock that will result in continuous switching activity that will cause the power consumption to increase over a design that are combinational. In a sequential circuit, adders and multipliers are a good example of how additional switching activity will increase power consumption. Reducing the power consumption for a sequential circuit could reduce the clock speed of the circuit.

A.5 Higher Level of Integration

Changing the way the circuit is integrated can help reduce power. Performing a System on Chip approach, non-cmos, or even reducing the lamda size of the circuit from 250 nm to 90 nm, for example, can reduce the power. Reducing the technology size of design reduces the Threshold Voltage (Vth) required to turn on the transistors. As seen in the Equation A.1, reducing the voltage or current will cause the power usage to go down. P represents power (P), V represents voltage (V), and I stands for Current (I). Lowering the Vth decreases power, resulting in a power savings.

\[ P = VI \] (A.1)

A.6 Dynamic Power Management

Finally, dynamic power management is another approach to reduce power consumption in a circuit design. It has been stated [14] that “Dynamic power management is one of the most popular and successful low power design techniques in
commercial integrated circuits”. This approach to saving power is built around the idea of conserving power in the circuit for activities that don’t need to be operating. The approach is also easy to build into a circuit design or even older design structures. One of the best features of using DPM is that in most cases the area of the system will not increase much in size and the performance will not be reduced. The mechanisms for DPM are a collection of techniques using the following areas in the circuit to reduce the power consumption: Clock Gating, Qualified System Latches, Guarded Evaluation, Bus Deactivation, and Self-timed Techniques. Now let’s take a look at each one of these power saving strategies in further detail.

A.6.1 Clock Gating. Clock Gating was touched on early but it’s still important to note that when dealing with circuits or microprocessors that high performance is typically trying to be achieved. Performance in a design requires that the clock of the system operate as quickly as possible. This in turn causes the most power consumption for the system [14] by the amount of switching activity that is taking place. Also, the clock tree for a system puts an additional load on the system causing increased power consumption. The idea is to eliminate unnecessary activity on segments of the clock signal by gating these with special qualifying signals. This can been seen in Figure A.3.
A.6.2 Qualified System Latches. According to Tiwari [14] “regular system latches have a data input and a control input that is usually connected to the system clock. A qualified latch has an additional input, the “enable” input, which determines if the inputs to the latch will be read or not when the clock is asserted.” The use of these latches can have a better power savings than regular latches. The reasoning behind this is that the regular latches use power every time the system clock is switching low to high, even when the data is not changing across the latch.

A.6.3 Guarded Evaluation. From the previous two subsections you saw how the clock on the system requires large amounts of power to operate. Guarded evaluation comes into play when you are unable to slow the clock or can not make any further corrections to reduce the amount of power the clock is drawing. For instance, [14] “if the combinational block at the output of a system latch is known to be doing no useful work, power can be canned by disabling transitions from entering the block.” It can also be beneficial if you have a set of [14] “modules that share a set of inputs”. Being able to turn on only the module that is used will save power from going to a module that is not being used. This is seen in Figure A.4.

Figure A.4: Guarded evaluation for a dual function ALU: (a) Original design, (b) Guarded design
A.6.4 Bus Deactivation. According to Tiwari [14] “the idea is to restrict the enabling condition on tristate buffers driving a bus, such that the bus is not driven on cycles when it is known that its results will not be used. Such unnecessary activity on heavily loaded buses can lead to significant power wastage.”

A.6.5 Self-timed Techniques. Self-timed techniques are useful with on-chip memories to reduce power. When dealing with microprocessors the memory can cause large power consumption as data is being transferred back and forth to the cache with on-chip memories systems. “A low-power optimization is to pre-charge the bit lines and activate the sense-amps using short pulses. These pulses are generated only when changes are detected on the address lines and a read or write is going to be initiated [14].”

A.7 Synthesize

The synthesis approach in this research will be using Cadence Software CAD tools. The paper [11] “Leakage Power Optimization Flow”, written by Sirsi, discusses three different Power Flow models. The Power Flow model that worked the best for reducing power consumption for clock speed of 200Mhz, technology 130 nm, and cell instances 120K was his Power Flow 3 which uses a RTL Compiler and System on Chip Encounter. Siri’s power flow model is illustrated in Figure A.5. In general, the flow taken from Cadence’s support documentation has the block diagrams broken into simpler blocks. The most important thing for a good design is the design flow used for the system. Writing a good script to address these challenges plays a large role with using the Cadence Tools. The first step is to set up your environment settings along with what technology library that will be used. Load the VHDL file and perform elaboration on the design. Apply constraints such as timing, design rule constraints, defining input and output delay, and etc. Once these have been set it’s a good idea to add optimization setting to the design. Now you’re ready for the final steps of the script to synthesize, perform analysis, and export the design as a RTL file.
Figure A.5: Leakage Power Optimization Flow
Appendix B. Matlab® Code

This section has the Matlab® code that was written by Sohai Khan [8] for Optical Flow using the Lucas-Kanade method.

B.1 HierarchicalLK

Listing B.1: The HierarchicalLK.m Matlab® file.

```matlab
function [u,v, cert] = HierarchicalLK(im1, im2, numLevels, ...
    windowSize, iterations, display)
% HIERARCHICALLK Hierarchical Lucas Kanade
% [u,v]=HierarchicalLK(im1, im2, numLevels, windowSize, iterations...
%    , display)
% Tested for pyramids of height 1, 2, 3 only...
% operation with pyramids of height 4 might be unreliable
% Use quiver(u, -v, 0) to view the results
% NUMLEVELS Pyramid Levels (typical value 3)
% WINDOWSIZE Size of smoothing window (typical value 1-4)
% ITERATIONS number of iterations (typical value 1-5)
% DISPLAY 1 to display flow fields (1 or 0)
% Uses: Reduce, Expand
% Sohaib Khan
% edited 05-15-03 (Yaser)
% yaser@cs.ucf.edu
% technique, with an Application to Stero Vision," Int'l Joint
% Conference Artificial Intelligence, pp. 121-130, 1981.

if (size(im1,1) ~= size(im2,1)) | (size(im1,2) ~= size(im2,2))
    error('images are not same size');
end;

if (size(im1,3) ~= 1) | (size(im2, 3) ~= 1)
    error('input should be gray level images');
end;

% check image sizes and crop if not divisible
if (rem(size(im1,1), 2^(numLevels - 1)) ~= 0)
    warning('image will be cropped in height, size of output will ...
    be smaller than input!');
    im1 = im1(1:(size(im1,1) - rem(size(im1,1), 2^(numLevels - 1)))...)
    , :);
    im2 = im2(1:(size(im1,1) - rem(size(im1,1), 2^(numLevels - 1)))...)
    , :);
```

if (rem(size(im1,2), 2^((numLevels - 1))) ~= 0)
    warning('image will be cropped in width, size of output will ... be smaller than input!');
    im1 = im1(1, 1:size(im1,2) - rem(size(im1,2), 2^((numLevels - ... 1))));
    im2 = im2(1, 1:size(im1,2) - rem(size(im1,2), 2^((numLevels - ... 1))));
end;

%Build Pyramids
pyramid1 = im1;
pyramid2 = im2;

for i = 2: numLevels
    im1 = reduce(im1);
    im2 = reduce(im2);
    pyramid1(1:size(im1,1), 1:size(im1,2), i) = im1;
    pyramid2(1:size(im2,1), 1:size(im2,2), i) = im2;
end;

% base level computation
disp(' Computing Level 1');
baseIm1 = pyramid1(1, 1:size(pyramid1,1)/(2^(numLevels -1)), numLevels);
baseIm2 = pyramid2(1, 1:size(pyramid2,1)/(2^(numLevels -1)), numLevels);
[u,v] = LucasKanade(baseIm1, baseIm2, windowSize);

for r = 1: iterations
    [u, v] = LucasKanadeRefined(u, v, baseIm1, baseIm2);
end

%propagating flow 2 higher levels
for i = 2: numLevels
    disp([' Computing Level ', num2str(i)]);
    uEx = 2 * imresize(u, size(u)*2); % use appropriate expand function (gaussian, bilinear, cubic, etc).
    vEx = 2 * imresize(v, size(v)*2);
    curIm1 = pyramid1(1, 1:size(pyramid1,1)/(2^(numLevels - i)), (numLevels - i)... +1);
    curIm2 = pyramid2(1, 1:size(pyramid2,1)/(2^(numLevels - i)), (numLevels - i)... +1);
    [u, v] = LucasKanadeRefined(uEx, vEx, curIm1, curIm2);
end

[u, v, cert] = LucasKanadeRefined(u, v, curIm1, curIm2);
end
end;
if (display==1)
    figure, quiver(reduce((reduce(medfilt2(flipud(u),[5 5])))), -...
    reduce((reduce(medfilt2(flipud(v),[5 5])))), 0), axis equal
end

B.2 Reduce

Listing B.2: The Reduce.m Matlab® file.(appendix2/Reduce.m)

function smallIm = Reduce(im)
% REDUCE Compute smaller layer of Gaussian Pyramid
% Sohaib Khan, Feb 16, 2000
% Algo
% Gaussian mask = [0.05 0.25 0.4 0.25 0.05]
% Apply 1d mask to alternate pixels along each row of image
% apply 1d mask to each pixel along alternate columns of
% resulting image

mask = [0.05 0.25 0.4 0.25 0.05];

hResult = conv2(im, mask);
hResult = hResult(:,3:size(hResult,2)-2);
hResult = hResult(:, 1:2:size(hResult,2));

vResult = conv2(hResult, mask');
vResult = vResult(3:size(vResult,1)-2, :);
vResult = vResult(1:2:size(vResult,1),:);

smallIm = vResult;

B.3 LucasKanade

Listing B.3: The LucasKanade.m Matlab® file.(appendix2/LucasKanade.m)

function [u, v] = LucasKanade(im1, im2, windowSize);
% LucasKanade lucas kanade algorithm, without pyramids
% (only 1 level);
% REVISION: NaN vals are replaced by zeros

[fx, fy, ft] = ComputeDerivatives(im1, im2);
u = zeros(size(im1));
v = zeros(size(im2));
halfWindow = floor(windowSize/2);
for i = halfWindow+1:size(fx,1)-halfWindow
    for j = halfWindow+1:size(fx,2)-halfWindow
        curFx = fx(i-halfWindow:i+halfWindow, j-halfWindow:j+halfWindow);
        curFy = fy(i-halfWindow:i+halfWindow, j-halfWindow:j+halfWindow);
        curFt = ft(i-halfWindow:i+halfWindow, j-halfWindow:j+halfWindow);
        curFx = curFx';
        curFy = curFy';
        curFt = curFt';
        curFx = curFx(:,);
        curFy = curFy(:,);
        curFt = -curFt(:,);
        A = [curFx curFy];
        U = pinv(A'*A)*A'*curFt;
        u(i,j)=U(1);
        v(i,j)=U(2);
    end;
end;

u(isnan(u))=0;
v(isnan(v))=0;

function [fx, fy, ft] = ComputeDerivatives(im1, im2);
%ComputeDerivatives Compute horizontal, vertical and time
derivative between two gray-level images.
if (size(im1,1) ~= size(im2,1)) | (size(im1,2) ~= size(im2,2))
    error('input images are not the same size');
end;
if (size(im1,3)~=1) | (size(im2,3)~=1)
    error('method only works for gray-level images');
end;
fx = conv2(im1, 0.25*[-1 1; -1 1]) + conv2(im2, 0.25*[-1 1; -1 1])...
function [u,v,cert] = LucasKanadeRefined(uIn, vIn, im1, im2);

uIn = round(uIn);

vIn = round(vIn);

uIn = uIn(2:size(uIn,1), 2:size(uIn, 2)-1);

vIn = vIn(2:size(vIn,1), 2:size(vIn, 2)-1);

u = zeros(size(im1));

v = zeros(size(im2));

% to compute derivatives, use a 5x5 block...

% the resulting derivative will be 5x5...

% take the middle 3x3 block as derivative

for i = 3:size(im1,1)-2
    for j = 3:size(im2,2)-2
        % if uIn(i,j)~=0
        % disp('ha');
        % end;
        curIm1 = im1(i-2:i+2, j-2:j+2);
        lowRindex = i-2+vIn(i,j);
        highRindex = i+2+vIn(i,j);
        lowCindex = j-2+uIn(i,j);
        highCindex = j+2+uIn(i,j);
        if (lowRindex < 1)
            lowRindex = 1;
            highRindex = 5;
        end;

ft = ft(1:size(ft,1)-1, 1:size(ft,2)-1);

fx=fx(1:size(fx,1)-1, 1:size(fx,2)-1);
fy=fy(1:size(fy,1)-1, 1:size(fy,2)-1);

% make same size as input

fx=fx(1:size(fx,1)-1, 1:size(fx,2)-1);
fy=fy(1:size(fy,1)-1, 1:size(fy,2)-1);

ft=ft(1:size(ft,1)-1, 1:size(ft,2)-1);

% LucasKanadeRefined

Listing B.4: The LucasKanadeRefined.m Matlab® file.
(appendix2/LucasKanadeRefined.m)
if (highRindex > size(im1,1))
    lowRindex = size(im1,1)-4;
    highRindex = size(im1,1);
end;

if (lowCindex < 1)
    lowCindex = 1;
    highCindex = 5;
end;

if (highCindex > size(im1,2))
    lowCindex = size(im1,2)-4;
    highCindex = size(im1,2);
end;

if isnan(lowRindex)
    lowRindex = i-2;
    highRindex = i+2;
end;

if isnan(lowCindex)
    lowCindex = j-2;
    highCindex = j+2;
end;

curIm2 = im2(lowRindex:highRindex, lowCindex:highCindex);
[curFx, curFy, curFt]=ComputeDerivatives(curIm1, curIm2);
curFx = curFx(2:4, 2:4);
curFy = curFy(2:4, 2:4);
curFt = curFt(2:4, 2:4);
curFx = curFx';
curFy = curFy';
curFt = curFt';
curFx = curFx(:);
curFy = curFy(:);
curFt = -curFt(:);
A = [curFx curFy];
U = pinv(A'*A)*A'*curFt;
u(i,j)=U(1);
v(i,j)=U(2);
cert(i,j) = rcond(A'*A);
% derivative between two gray-level images.

if (size(im1,1) ~= size(im2,1)) | (size(im1,2) ~= size(im2,2))
    error('input images are not the same size');
end;

if (size(im1,3) ~=1) | (size(im2,3) ~=1)
    error('method only works for gray-level images');
end;

fx = conv2(im1, 0.25*[-1 1; -1 1]) + conv2(im2, 0.25*[-1 1; -1 1])...
    ;
fy = conv2(im1, 0.25*[-1 -1; 1 1]) + conv2(im2, 0.25*[-1 -1; 1 1])...
    ;
ft = conv2(im1, 0.25*ones(2)) + conv2(im2, -0.25*ones(2));

% make same size as input
fx = fx(1:size(fx,1)-1, 1:size(fx,2)-1);
fy = fy(1:size(fy,1)-1, 1:size(fy,2)-1);
ft = ft(1:size(ft,1)-1, 1:size(ft,2)-1);

B.5 Expand

Listing B.5: The Expand.m Matlab® file.(appendix2/Expand.m)

function largeIm = Expand(im);

% EXPAND Compute large layer of Gaussian pyramid

% Schaib Khan, Feb 16, 2000

% Algo
% Gaussian mask = [0.05 0.25 0.4 0.25 0.05]
% Insert zeros in every alternate row position and conv with mask
% insert zeros in every alternate clmn position in result
% and conv with mask’

mask = 2*[0.05 0.25 0.4 0.25 0.05];

% factor of 2 is there because
% each pixel gets contribution either
% from 0.05, 0.4, 0.05 or from 0.25, 0.25
% insert zeros in every alternate position in each row
rowZeros = [im; zeros(size(im))];
rowZeros = reshape(rowZeros, size(im,1), 2*size(im,2));

% conv with horiz mask
newIm = conv2(rowZeros, mask);
newIm = newIm(:,3:size(newIm,2)-2);

% insert zeros in every alternate position in each col
colZeros = newIm';
colZeros = [colZeros; zeros(size(colZeros))];
colZeros = reshape(colZeros, size(colZeros,1)/2, 2*size(colZeros,...
,2));
colZeros = colZeros';

largeIm=conv2(colZeros, mask');
largeIm=largeIm(3:size(largeIm,1)-2,:);
Appendix C. VHDL Code

This section has the Top-Level Designs that were coded in VHDL to convert the Matlab® commands and functions.

C.1 Reduce Matrix Function Top-Level 1 Behavior Multiplier

Listing C.1: The ReduceConv2Module.vhd VHDL file.
(appendix3/ReduceConv2Module.vhd)

```vhdl
-- Capt. Jason Shirley
-- This is the Reduce Matrix Function Top-Level Design 1 using the
-- Behavior Multiplier. These Modules are structurally connected.

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

entity reduceconv2_module is
  generic (address_width : integer := 8;
            data_width : integer := 24);
  port (clk : in std_logic;
         reset : in std_logic;
         enable_matrixA : in std_logic;
         enable_conv2_module : in std_logic;
         read_address_matrixA : in std_logic_vector (address_width - 1 downto 0);
         read_address_matrixReduce : in std_logic_vector (address_width - 1 downto 0);
         input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
         output_data_matrixC : out std_logic_vector (data_width - 1 downto 0);
         reduce_module_finished : out std_logic;
         input_to_matrixC_hresult : out std_logic_vector (data_width - 1 downto 0);
         writeEnable_matrixC_hresult : out std_logic;
         address_matrixA_hresult : out std_logic_vector (address_width - 1 downto 0);
         address_matrixB_hresult : out std_logic_vector (address_width - 1 downto 0);
         address_matrixC_hresult : out std_logic_vector (address_width - 1 downto 0);
         matrixA_output_hresult : out std_logic_vector (data_width - 1 downto 0);
```

69
matrixB_output_hresult : out std_logic_vector(...
  data_width - 1 downto 0);
control_adder_signal_hresult : out std_logic;
  input_to_matrixC_vresult : out std_logic_vector(...
  data_width - 1 downto 0);
writeEnable_matrixC_vresult : out std_logic;
  address_matrixA_vresult : out std_logic_vector(...
  address_width - 1 downto 0);
  address_matrixB_vresult : out std_logic_vector(...
  address_width - 1 downto 0);
  address_matrixC_vresult : out std_logic_vector(...
  address_width - 1 downto 0);
matrixA_output_vresult : out std_logic_vector(...
  data_width - 1 downto 0);
matrixB_output_vresult : out std_logic_vector(...
  data_width - 1 downto 0);
control_adder_signal_vresult: out std_logic
);
end reduceconv2_module;

architecture structure_reduceconv2_module of reduceconv2_module is
  component hresultconv2_module is
    generic ( address_width : integer := 8;
    data_width : integer := 24);

    port ( clk : in std_logic;
    reset : in std_logic;
    enable_matrixA : in std_logic;
    enable_conv2_module : in std_logic;
    read_address_matrixA : in std_logic_vector (...
      address_width - 1 downto 0);
    read_address_matrixC : in std_logic_vector (...
      address_width - 1 downto 0);
    input_matrixA_values : in std_logic_vector (data_width -...
      1 downto 0);
    output_data_matrixC : out std_logic_vector(data_width -...
      1 downto 0);
    enable_reduce : out std_logic;
    input_to_matrixC : out std_logic_vector(data_width -...
      1 downto 0);
    writeEnable_matrixC : out std_logic;
    address_matrixA : out std_logic_vector(...
      address_width - 1 downto 0);
    address_matrixB : out std_logic_vector(...
      address_width - 1 downto 0);
    address_matrixC : out std_logic_vector(...
      address_width - 1 downto 0);
matrixA_output : out std_logic_vector(data_width - 1 downto 0);
matrixB_output : out std_logic_vector(data_width - 1 downto 0);
control_adder_signal : out std_logic);
end component hresultconv2_module;

component hresultreducecontrol is
  generic (address_width: integer := 8;
           n1: integer := 5;
           n2: integer := 11);
  -- address_width must be an even number
  -- The hResultReduceControl module reduces the original 6x8
  -- input matrix in half from the number of columns from the
  -- hResultControl module. Example 6 x 12 will be reduce down
  -- to 6x4 n1 is the row position and n2 is the column position.
  -- you want n1 and n2 to be one size smaller then the 6 x 12
  -- therefore n1 should equal = 5 since you start at zero
  -- and n2 should equal = 11 since you start at zero
  -- The hResultReduceControl module reduces the original 6x8
  -- input matrix in half from the number of columns from the
  -- hResultControl module. Example 6 x 12 will be reduce down
  -- to 6x4 n1 is the row position and n2 is the column position.
  -- you want n1 and n2 to be one size smaller then the 6 x 12
  -- therefore n1 should equal = 5 since you start at zero
  -- and n2 should equal = 11 since you start at zero
  port (clk : in std_logic;
        reset : in std_logic;
        enable_control : in std_logic;
        writeEnable_matrix : out std_logic;
        enable_vresult : out std_logic;
        read_address_hResult : out std_logic_vector(...
                                                      address_width - 1 downto 0);
        write_address_reducehResult: out std_logic_vector(...
                                                       address_width - 1 downto 0)
      );
end component hresultreducecontrol;

component vresultconv2_module is
  generic (address_width: integer := 8;
           data_width: integer := 24);
  port (clk : in std_logic;
reset : in std_logic;
enable_matrixA : in std_logic;
enable_conv2_module : in std_logic;
read_address_matrixA : in std_logic_vector (address_width - 1 downto 0);
read_address_matrixC : in std_logic_vector (address_width - 1 downto 0);
input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
output_data_matrixC : out std_logic_vector (data_width - 1 downto 0);
enable_reduce : out std_logic;
input_to_matrixC : out std_logic_vector (data_width - 1 downto 0);
writeEnable_matrixC : out std_logic;
address_matrixA : out std_logic_vector (address_width - 1 downto 0);
address_matrixB : out std_logic_vector (address_width - 1 downto 0);
address_matrixC : out std_logic_vector (address_width - 1 downto 0);
matrixA_output : out std_logic_vector (data_width - 1 downto 0);
matrixB_output : out std_logic_vector (data_width - 1 downto 0);
control_adder_signal : out std_logic
);
end component vresultconv2_module;

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

component vresultreducecontrol is

  generic (address_width : integer := 8;
            n1: integer := 9;
            n2: integer := 3);
  -- address_width must be an even number
  port (clk : in std_logic;
        reset : in std_logic;
        enable_control : in std_logic;
        writeEnable_matrix : out std_logic;
        enable_vresult : out std_logic;
        read_address_vResult : out std_logic_vector (address_width - 1 downto 0);
        write_address_reducevResult: out std_logic_vector (address_width - 1 downto 0)
  );
end component vresultreducecontrol;

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

component mem_matrix is
generic (address_width: integer := 8;
data_width: integer := 24);

port (clk : in std_logic;
reset : in std_logic;
writeEnable : in std_logic;
matrix_values : in std_logic_vector (data_width - 1 downto 0);
write_address : in std_logic_vector (address_width - 1 downto 0);
read_address : in std_logic_vector (address_width - 1 downto 0);
output_matrix_values : out std_logic_vector(data_width - 1 downto 0));

end component mem_matrix;

-----------------------------------------------------------------
signal enable_reduce_h_to_enable_control, ...
writeEnable_matrix_to_enable_matrixA: std_logic;
signal enable_vresult_to_enable_conv2_module, ...
enable_reduce_v_to_enable_control: std_logic;
signal writeEnable_matrix_to_writeEnable: std_logic;
signal write_address_reducevResult_to_write_address:...
std_logic_vector (address_width - 1 downto 0);
signal read_address_hResult_to_read_address_matrixC:...
std_logic_vector (address_width - 1 downto 0);
signal read_addressvResult_to_read_address_matrixC:...
std_logic_vector (address_width - 1 downto 0);
signal write_address_reducehResult_to_read_address_matrixA:...
std_logic_vector(data_width - 1 downto 0);
signal output_data_matrixC_to_input_matrixA_values: ...
std_logic_vector(data_width - 1 downto 0);
signal output_data_matrixC_to_matrix_values: std_logic_vector(...
data_width - 1 downto 0));

begin
hresultconv2_module1: hresultconv2_module port map (clk => clk,
reset => reset,
enable_matrixA => enable_matrixA,
enable_conv2_module => enable_conv2_module,
read_address_matrixA => read_address_matrixA,
read_address_matrixC =>
read_address_hResult_to_read_address_matrixC,
input_matrixA_values => input_matrixA_values,
output_data_matrixC =>
output_data_matrixC_to_input_matrixA_values,
enable_reduce => enable_reduce_h_to_enable_control,
input_to_matrixC => input_to_matrixC_hresult,
writeEnable_matrixC => writeEnable_matrixC_hresult,
hresultreducecontrol1: hresultreducecontrol port map(clk => clk,
reset => reset,
enable_control => enable_reduce_h_to_enable_control,
writeEnable_matrix => writeEnable_matrix_to_enable_matrixA,
read_address_hResult => read_address_hResult_to_read_address_matrixC,
write_address_reducehResult => write_address_reducehResult_to_read_address_matrixA);

vresultconv2_module1: vresultconv2_module port map(clk => clk,
reset => reset,
enable_matrixA => writeEnable_matrix_to_enable_matrixA,
read_address_matrixA => write_address_reducehResult_to_read_address_matrixA,
read_address_matrixC => read_addressvResult_to_read_address_matrixC,
input_matrixA_values => output_matrixA_values_to_input_matrixA_values,
output_matrixA_values => output_matrixA_values_to_output_matrixA_values,
input_matrixC_values => input_matrixC_values_to_input_matrixC_values,
writeEnable_matrixC => writeEnable_matrixC_to_writeEnable_matrixC,
write_address_matrixA => write_address_matrixA_to_write_address_matrixA,
write_address_matrixB => write_address_matrixB_to_write_address_matrixB,
write_address_matrixC => write_address_matrixC_to_write_address_matrixC,
output_data_matrixC => output_data_matrixC_to_output_data_matrixC,
control_adder_signal => control_adder_signal_to_control_adder_signal);

mem_matrixReduce: mem_matrix port map(clk => clk,
reset => reset,
writeEnable => writeEnable_matrix_to_writeEnable,
matrix_values => output_data_matrixC_to_matrix_values,
C.2 Reduce Matrix Function Top-Level 2 Behavior Multiplier

Listing C.2: The ReduceConv2Module2.vhd VHDL file.

```vhdl
-- Capt. Jason Shirley
-- This is the Reduce Matrix Function Top-Level Design 2 using the
-- Behavior Multiplier. These Modules are structurally connected.

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

entity reduceconv2_module2 is
  generic (
    address_width : integer := 8;
    data_width : integer := 24
  );
  port ( clk : in std_logic;
         reset : in std_logic;
         enable_matrixA : in std_logic;
         enable_conv2_module : in std_logic;
         read_address_matrixA : in std_logic_vector (address_width - 1 downto 0);
         read_address_matrixReduce : in std_logic_vector (address_width - 1 downto 0);
         input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
         reduce_module_finished : out std_logic;
         output_data_matrixReduce : out std_logic_vector(data_width - 1 downto 0)
   );
end reduceconv2_module2;

architecture structure_reduceconv2_module2 of reduceconv2_module2 ...
  is
```
component hresultcontrol2 is
  generic (address_width: integer := 8;
    n1: integer := 5;
    n2: integer := 11);
  -- address_width must be an even number
  port (clk : in std_logic;
    reset : in std_logic;
    enable_control : in std_logic;
    update_reg_enable : out std_logic;
    writeEnable_matrixC : out std_logic;
    enable_reduce : out std_logic;
    read_address_matrixA : out std_logic_vector (address_width - 1 downto 0);
    read_address_matrixB : out std_logic_vector (address_width - 1 downto 0);
    write_address_matrixC : out std_logic_vector (address_width - 1 downto 0);
    mux_enable : out std_logic);
end component hresultcontrol2;

component hresultreducecontrol2 is
  generic (address_width: integer := 8;
    n1: integer := 5;
    n2: integer := 11);
  -- address_width must be an even number
  -- The hResultReduceControl module reduces the original 6x8
  -- input matrix in half from the number of columns from the
  -- hResultControl module. Example 6 x 12 will be reduce down
  -- to 6x4 n1 is the row position and n2 is the column position.
  -- you want n1 and n2 to be one size smaller then the 6 x 12
  -- therefore n1 should equal = 5 since you start at zero
  -- and n2 should equal = 11 since you start at zero
  -- The hResultReduceControl module reduces the original 6x8
  -- input matrix in half from the number of columns from the
  -- hResultControl module. Example 6 x 12 will be reduce down
  -- to 6x4 n1 is the row position and n2 is the column position.
  -- you want n1 and n2 to be one size smaller then the 6 x 12
  -- therefore n1 should equal = 5 since you start at zero
  -- and n2 should equal = 11 since you start at zero
  port (clk : in std_logic;
    reset : in std_logic;
    enable_control : in std_logic;
    writeEnable_matrix : out std_logic;
    enable_vresult : out std_logic);
read_address_hResult : out std_logic_vector (address_width - 1 downto 0);
write_address_reducehResult: out std_logic_vector (address_width - 1 downto 0);
mux_enable : out std_logic
);
end component hresultreducecontrol2;
----------------------------------------------------------------
component vresultcontrol2 is
  generic ( address_width : integer := 8;
    n1: integer := 9;
    n2: integer := 3);
  -- address_width must be an even number
  port ( clk : in std_logic;
    reset : in std_logic;
    enable_control : in std_logic;
    writeEnable_matrixC : out std_logic;
    reset_reg : out std_logic;
    enable_reduce : out std_logic;
    read_address_matrixA : out std_logic_vector (address_width - 1 downto 0);
    read_address_matrixB : out std_logic_vector (address_width - 1 downto 0);
    read_address_matrixC : out std_logic_vector (address_width - 1 downto 0)
  );
end component vresultcontrol2;
----------------------------------------------------------------
component vresultreducecontrol2 is
  generic ( address_width : integer := 8;
    n1: integer := 9;
    n2: integer := 3);
  -- address_width must be an even number
  port ( clk : in std_logic;
    reset : in std_logic;
    enable_control : in std_logic;
    writeEnable_matrix : out std_logic;
    enable_vresult : out std_logic;
    read_address_vResult : out std_logic_vector (address_width - 1 downto 0);
    write_address_reducevResult: out std_logic_vector (address_width - 1 downto 0)
  );
end component vresultreducecontrol2;
component mux2to1 is

    generic (address_width: integer := 8);

    port (sel : in std_logic;
        input1 : in std_logic_vector(address_width - 1 downto 0);
        input2 : in std_logic_vector(address_width - 1 downto 0);
        mux_output : out std_logic_vector(address_width - 1 downto 0)
    );

end component mux2to1;

component mux1to2data_width is

    generic (data_width: integer := 24);

    port (sel : in std_logic;
        input : in std_logic_vector(data_width - 1 downto 0);
        mux_output1 : out std_logic_vector(data_width - 1 downto 0);
        mux_output2 : out std_logic_vector(data_width - 1 downto 0)
    );

end component mux1to2data_width;

component mux2to1data_width is

    generic (data_width: integer := 24);

    port (sel : in std_logic;
        input1 : in std_logic_vector(data_width - 1 downto 0);
        input2 : in std_logic_vector(data_width - 1 downto 0);
        mux_output : out std_logic_vector(data_width - 1 downto 0)
    );

end component;

component mux6to3enable is

    port (sel : in std_logic;
        input1 : in std_logic;
        input2 : in std_logic;
        input3 : in std_logic;
        input4 : in std_logic;
    );

end component;
component mem_matrix is
    generic (address_width: integer := 8;
             data_width: integer := 24);
    port (clk : in std_logic;
          reset : in std_logic;
          write_enable : in std_logic;
          matrix_values : in std_logic_vector (data_width - 1 downto 0);
          write_address : in std_logic_vector (address_width - 1 downto 0);
          read_address : in std_logic_vector (address_width - 1 downto 0);
          output_matrix_values : out std_logic_vector (data_width - 1 downto 0)
    );
end component mem_matrix;

component mask1x5mem_matrix is
    generic (address_width: integer := 8;
             data_width: integer := 24);
    port (clk : in std_logic;
          reset : in std_logic;
          read_address : in std_logic_vector (address_width - 1 downto 0);
          output_matrix_values: out std_logic_vector(data_width - 1 downto 0)
    );
end component mask1x5mem_matrix;

component mask5x1mem_matrix is
    generic (address_width: integer := 8;
             data_width: integer := 24);
    port (clk : in std_logic;
          reset : in std_logic;
          read_address : in std_logic_vector (address_width - 1 downto 0);
          output_matrix_values: out std_logic_vector(data_width - 1 downto 0)
    );
end component mask5x1mem_matrix;
port(clk : in std_logic;
    reset : in std_logic;
    read_address : in std_logic_vector (address_width - 1 downto 0);
    output_matrix_values : out std_logic_vector(data_width - 1 downto 0)
);
end component mask5x1mem_matrix;

component conv2 is
    generic (data_width : integer := 24);
    port (clk : in std_logic;
        update_reg_enable : in std_logic;
        input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
        input_matrixB_values : in std_logic_vector (data_width - 1 downto 0);
        output_data_matrixC : out std_logic_vector (data_width - 1 downto 0)
    );
end component conv2;

signal update_reg_enable_to_mux_input1, ...
    control_writeEnable_memC_to_WriteEnableMatrixC_mux_input3: ...
    std_logic;
signal control_reset_reg_to_hvcontrol_reset_mux_input2, ...
    enable_reduce_to_hResultReduceControl: std_logic;
signal mux_enable_to_mux_select, writeEnable_matrix_to_MatrixA2, ...
    enable_vresult_to_vResultControl : std_logic;
signal mux_enable_to_mux_select_matrixA2, ...
    update_reg_enable_to_mux_input4: std_logic;
signal control_writeEnable_memC_to_mux_input6: std_logic;
signal control_reset_reg_to_mux_input5, ...
    enable_reduce_to_vResultReduceControl_enable : std_logic;
signal writeEnable_matrix_to_MatrixReduce, ...
    mux_output_to_conv2_update_reg: std_logic;
signal mux_output_to_conv2_reset, ...
    mux_out_writeEnableMatrixC_to_writeEnableMatrixC: std_logic;
signal read_address_matrixA_to_mux_input1, ...
    read_address_matrixB_to_mask1x5 : std_logic_vector (address_width - 1 downto 0);
signal write_address_matrixC_to_WriteAddressMatrix_mux_input1, ...
    write_address_reducehResult_to_mux_input1_matrixA2_address : ...
    std_logic_vector (address_width - 1 downto 0);
begin

hresultcontrol1: hresultcontrol2 port map (clk => clk,
reset => reset,
enable_control => enable_conv2_module,
update_reg_enable => update_reg_enable_to_mux_input1,
writeEnable_matrixC =>
control_writeEnable_memC_to_WriteEnableMatrixC_mux_input3,
reset_reg => control_reset_reg_to_hvcontrol_reset_mux_input2,
enable_reduce => enable_reduce_to_hResultReduceControl,
read_address_matrixA => read_address_matrixA_to_mux_input1,
read_address_matrixB => read_address_matrixB_to_mask1x5,
write_address_matrixC =>
write_address_matrixC_to_WriteAddressMatrix_mux_input1,
mux_enable => mux_enable_to_mux_select);

hresultreducecontrol1: hresultreducecontrol2 port map (clk => clk,
reset => reset,
enable_control => enable_reduce_to_hResultReduceControl,
writeEnable_matrix => writeEnable_matrix_to_MatrixA2,
enable_vresult => enable_vresult_to_vResultControl,
read_address_hResult => read_address_hResult_to_mux_input1_readAddressMatrixC,
write_address_reducehResult => write_address_reducehResult_to_mux_input1_matrixA2_address,
mux_enable => mux_enable_to_mux_select_matrixA2);

vresultcontrol1: vresultcontrol2 port map (clk => clk,
reset => reset,
enable_control => enable_vresult_to_vResultControl,
update_reg_enable => update_reg_enable_to_mux_input4,
writeEnable_matrixC => control_writeEnable_memC_to_mux_input6,
reset_reg => control_reset_reg_to_mux_input5,
enable_reduce => enable_reduce_to_vResultReduceControl_enable,
read_address_matrixA =>
write_address_reducevResult => write_address_reducevResult_to_mux_input2_matrixA2_address,
write_address_matrixB => read_address_matrixB_to_mask5x1,
write_address_matrixC => write_address_matrixC_to_WriteAddressMatrix_mux_input2);

vresultreducecontrol1 : vresultreducecontrol2 port map(clk => clk,
reset => reset,
enable_control => enable_reduce_to_vResultReduceControl_enable,
writeEnable_matrix => writeEnable_matrix_to_MatrixReduce,
enable_vresult => reduce_module_finished,
read_address_vResult => read_address_vResult_to_mux_input2_readAddressMatrixC,
write_address_reducevResult => write_address_reducevResult_to_MatrixReduce);

mux2to1_matrixA1 : mux2to1 port map(sel => enable_matrixA,
input1 => read_address_matrixA_to_mux_input1,
input2 => read_address_matrixA,
mux_output => mux_output_to_matrixA1);

mux2to1_matrixA_values: mux2to1data_width port map(
 sel => mux_enable_to_mux_select,
inpt1 => output_matrix_values_to_mux_input1_matrixA_values,
inpt2 => output_matrix_values_to_mux_input2_matrixA_values,
mux_output => mux_output_matrixA_values_to_conv2_input1);

mux2to1_matrixB_values: mux2to1data_width port map(
 sel => mux_enable_to_mux_select,
inpt1 => output_matrix_values_to_mux_input1_matrixB_values,
inpt2 => output_matrix_values_to_mux_input2_matrixB_values,
mux_output => mux_output_matrixB_to_conv2_input2);
mux6to3_Enable1 : mux6to3enable port map (  
  sel => mux_enable_to_mux_select,  
  input1 => update_reg_enable_to_mux_input1,  
  input2 => control_reset_reg_to_hvcontrol_reset_mux_input2,  
  input3 =>  
  control_writeEnable_memC_to_WriteEnableMatrixC_mux_input3,  
  input4 => update_reg_enable_to_mux_input4,  
  input5 => control_reset_reg_to_mux_input5,  
  input6 => control_writeEnable_memC_to_mux_input6,  
  mux_output1 => mux_output_to_conv2_update_reg,  
  mux_output2 => mux_output_to_conv2_reset,  
  mux_output3 => mux_out_writeEnableMatrixC_to_writeEnableMatrixC );

mux2to1_ReadAddressMatrixC: mux2to1 port map(  
  sel => mux_enable_to_mux_select_matrixA2,  
  input1 => read_address_hResult_to_mux_input1_readAddressMatrixC,  
  input2 => read_address_vResult_to_mux_input2_readAddressMatrixC,  
  mux_output => read_address_matrixC_to_address_matrixC);

mux2to1_WriteAddressMatrixC: mux2to1 port map(  
  sel => mux_enable_to_mux_select,  
  input1 => write_address_matrixC_to_WriteAddressMatrix_mux_input1,  
  input2 => write_address_matrixC_to_WriteAddressMatrix_mux_input2,  
  mux_output =>  
  mux_output_WriteAddressMatrixC_to_write_address_matrixC);

mux1to2_matrixA2_ReduceMatrix_values: mux1to2data_width port map(  
  sel => mux_enable_to_mux_select_matrixA2,  
  input => output_matrixC_values_to_mux_input_values,  
  mux_output1 => mux_output_MatrixC_values_to_matrixA2_values,  
  mux_output2 => mux_output_MatrixC_values_to_matrixReduce_values);

mem_matrixA1: mem_matrix port map (clk => clk, reset => reset,  
  writeEnable => enable_matrixA,  
  matrix_values => input_matrixA_values,  
  write_address => mux_output_to_matrixA1,  
  read_address => mux_output_to_matrixA1,  
  output_matrix_values =>  
  output_matrix_values_to_mux_input1_matrixA_values);

mem_matrixA2: mem_matrix port map (clk => clk, reset => reset,  
  writeEnable => writeEnable_matrix_to_MatrixA2,  
  matrix_values => mux_output_MatrixC_values_to_matrixA2_values,  
  write_address =>  
  write_address_reducehResult_to_mux_input1_matrixA2_address,  
  read_address =>)
write_address_reducevResult_to_mux_input2_matrixA2_address,
output_matrix_values =>
output_matrix_values_to_mux_input2_matrixA_values);

----------------------------------------------------------------
mask1x5mem_matrix0: mask1x5mem_matrix port map (clk => clk,
reset => reset,
read_address => read_address_matrixB_to_mask1x5,
output_matrix_values =>
output_matrix_values_to_mux_input1_matrixB_values);

----------------------------------------------------------------
mask5x1mem_matrix0: mask5x1mem_matrix port map (clk => clk,
reset => reset,
read_address => read_address_matrixB_to_mask5x1,
output_matrix_values =>
output_matrix_values_to_mux_input2_matrixB_values);

----------------------------------------------------------------
conv2_1: conv2 port map (clk => clk,
reset => mux_output_to_conv2_reset,
update_reg_enable => mux_output_to_conv2_update_reg,
input_matrixA_values => mux_output_matrixA_values_to_conv2_input1,
input_matrixB_values => mux_output_matrixB_to_conv2_input2,
output_data_matrixC => output_data_matrixC_to_matrixC_values);

----------------------------------------------------------------
mem_matrixC: mem_matrix port map (clk => clk,
reset => reset,
writeEnable => mux_out_writeEnableMatrixC_to_writeEnableMatrixC,
matrix_values => output_data_matrixC_to_matrixC_values,
write_address =>
write_address_matrixC_to_write_address_matrixC,
read_address => read_address_matrixC_to_address_matrixC,
output_matrix_values =>
output_matrixC_values_to_mux_input_values);

----------------------------------------------------------------
mem_matrixReduce: mem_matrix port map (clk => clk,
reset => reset,
writeEnable => writeEnable_matrix_to_MatrixReduce,
matrix_values => mux_output_MatrixC_values_to_matrixReduce_values,
write_address => write_address_reducevResult_to_MatrixReduce,
read_address => read_address_matrixReduce,
output_matrix_values => output_data_matrixReduce);

end structure_reduceconv2_module2;
C.3 Reduce Matrix Function Top-Level 1 Booth Multiplier

(appendix3/ReduceConv2ModuleBooth.vhd)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;
---------------------------------------------------------------
entity reduceconv2_modulebooth is
  generic (address_width: integer := 8;
            data_width: integer := 24);
  port (clk : in std_logic;
        reset : in std_logic;
        enable_matrixA : in std_logic;
        enable_conv2_module : in std_logic;
        read_address_matrixA : in std_logic_vector (address_width - 1 downto 0);
        read_address_matrixReduce : in std_logic_vector (address_width - 1 downto 0);
        input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
        output_data_matrixC : out std_logic_vector (data_width - 1 downto 0);
        reduce_module_finished : out std_logic;
        input_to_matrixC_hresult : out std_logic_vector (data_width - 1 downto 0);
        writeEnable_matrixC_hresult : out std_logic;
        address_matrixA_hresult : out std_logic_vector (address_width - 1 downto 0);
        address_matrixB_hresult : out std_logic_vector (address_width - 1 downto 0);
        address_matrixC_hresult : out std_logic_vector (address_width - 1 downto 0);
        matrixA_output_hresult : out std_logic_vector (data_width - 1 downto 0);
        matrixB_output_hresult : out std_logic_vector (data_width - 1 downto 0);
        control_adder_signal_hresult : out std_logic;
        input_to_matrixC_vresult : out std_logic_vector (data_width - 1 downto 0);
        writeEnable_matrixC_vresult : out std_logic;
```

85
address_matrixA_vresult  : out std_logic_vector(...
  address_width - 1 downto 0);
address_matrixB_vresult  : out std_logic_vector(...
  address_width - 1 downto 0);
address_matrixC_vresult  : out std_logic_vector(...
  address_width - 1 downto 0);
matrixA_output_vresult   : out std_logic_vector(...
  data_width - 1 downto 0);
matrixB_output_vresult   : out std_logic_vector(...
  data_width - 1 downto 0);
custom_adder_signal_vresult : out std_logic;
mult_ready_hresult       : out std_logic;
mult_done_hresult        : out std_logic;
mult_ready_vresult       : out std_logic;
mult_done_vresult        : out std_logic
);
matrixA_output : out std_logic_vector(data_width - 1 downto 0);
matrixB_output : out std_logic_vector(data_width - 1 downto 0);
control_adder_signal : out std_logic;
mult_ready : out std_logic;
mult_done : out std_logic);

end component hresultboothconv2_module;

component hresultreducecontrolbooth is

    generic (address_width: integer := 8;
n1: integer := 5;
n2: integer := 11);

    -- address_width must be an even number
    -- The hResultReduceControl module reduces the original 6x8
    -- input matrix in half from the number of columns from the
    -- hResultControl module. Example 6 x 12 will be reduce down
    -- to 6x4 n1 is the row position and n2 is the column position.
    -- you want n1 and n2 to be one size smaller then the 6 x 12
    -- therefore n1 should equal = 5 since you start at zero
    -- and n2 should equal = 11 since you start at zero
    -- The hResultReduceControl module reduces the original 6x8
    -- input matrix in half from the number of columns from the
    -- hResultControl module. Example 6 x 12 will be reduce down
    -- to 6x4 n1 is the row position and n2 is the column position.
    -- you want n1 and n2 to be one size smaller then the 6 x 12
    -- therefore n1 should equal = 5 since you start at zero
    -- and n2 should equal = 11 since you start at zero

    port (clk : in std_logic;
        reset : in std_logic;
        enable_control : in std_logic;
        writeEnable_matrix : out std_logic;
        enable_vresult : out std_logic;
        read_address_hResult : out std_logic_vector(address_width - 1 downto 0);
        write_address_reducehResult: out std_logic_vector(address_width - 1 downto 0);)

end component hresultreducecontrolbooth;

component vresultboothconv2_module is

    generic (address_width: integer := 8;
data_width: integer := 24);

end component vresultboothconv2_module;
port (clk : in std_logic;
    reset : in std_logic;
    enable_matrixA : in std_logic;
    enable_conv2_module : in std_logic;
    read_address_matrixA : in std_logic_vector (address_width - 1 downto 0);
    read_address_matrixC : in std_logic_vector (address_width - 1 downto 0);
    input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
    output_data_matrixC : out std_logic_vector (data_width - 1 downto 0);
    enable_reduce : out std_logic;
    input_to_matrixC : out std_logic_vector (data_width - 1 downto 0);
    writeEnable_matrixC : out std_logic;
    address_matrixA : out std_logic_vector (address_width - 1 downto 0);
    address_matrixB : out std_logic_vector (address_width - 1 downto 0);
    address_matrixC : out std_logic_vector (address_width - 1 downto 0);
    matrixA_output : out std_logic_vector (data_width - 1 downto 0);
    matrixB_output : out std_logic_vector (data_width - 1 downto 0);
    control_adder_signal : out std_logic;
    mult_ready : out std_logic;
    mult_done : out std_logic);
end component vresultboothconv2_module;

component vresultreducecontrolbooth is
  generic (address_width : integer := 8;
           n1: integer := 9;
           n2: integer := 3);
  -- address_width must be an even number
  port (clk : in std_logic;
        reset : in std_logic;
        enable_control : in std_logic;
        writeEnable_matrix : out std_logic;
        enable_vresult : out std_logic;
        read_address_vResult : out std_logic_vector (address_width - 1 downto 0);
        write_address_reducevResult: out std_logic_vector (address_width - 1 downto 0)
    )
end component vresultreducecontrolbooth;
end component vresultreducecontrolbooth;

component mem_matrix is

    generic (address_width : integer := 8;
              data_width : integer := 24);

    port (clk       : in std_logic;
          reset    : in std_logic;
          writeEnable : in std_logic;
          matrix_values : in std_logic_vector (data_width - 1 downto 0);
          write_address : in std_logic_vector (address_width - 1 downto 0);
          read_address : in std_logic_vector (address_width - 1 downto 0);
          output_matrix_values : out std_logic_vector (data_width - 1 downto 0));

end component mem_matrix;

signal enable_reduce_h_to_enable_control, ...
signal enable_reduce_v_to_enable_control, ...
signal enable_vresult_to_enable_conv2_module, ...
signal writeEnable_matrix_to_enable_matrixA: std_logic;

begin

hresultboothconv2_module1: hresultboothconv2_module port map (clk => clk,
reset => reset,
enable_matrixA => enable_matrixA,
enable_conv2_module => enable_conv2_module,
read_address_matrixA => read_address_matrixA,
read_address_matrixC => read_address_matrixC,
output_data_matrixC_to_input_matrixA_values : ...
output_data_matrixC_to_matrix_values : std_logic_vector(...
data_width - 1 downto 0));
input_matrixA_values => input_matrixA_values,
output_data_matrixC =>
output_data_matrixC_to_input_matrixA_values,
enable_reduce => enable_reduce_h_to_enable_control,
input_to_matrixC => input_to_matrixC_hresult,
writeEnable_matrixC => writeEnable_matrixC_hresult,
address_matrixA => address_matrixA_hresult,
address_matrixB => address_matrixB_hresult,
address_matrixC => address_matrixC_hresult,
matrixA_output => matrixA_output_hresult,
matrixB_output => matrixB_output_hresult,
control_adder_signal => control_adder_signal_hresult,
mult_ready => mult_ready_hresult,
mult_done => mult_done_hresult);

hresultreducecontrolbooth1 : hresultreducecontrolbooth port map(
clk => clk,
reset => reset,
enable_control => enable_reduce_h_to_enable_control,
writeEnable_matrix => writeEnable_matrix_to_enable_matrixA,
enable_vresult => enable_vresult_to_enable_conv2_module,
read_address_hResult =>
read_address_hResult_to_read_address_matrixC,
write_address_reducehResult =>
write_address_reducehResult_to_read_address_matrixA);

evresultboothconv2_module1 : vresultboothconv2_module port map(
clk => clk,
reset => reset,
enable_matrixA => writeEnable_matrix_to_enable_matrixA,
enable_conv2_module => enable_vresult_to_enable_conv2_module,
read_address_matrixA =>
read_addressvResult_to_read_address_matrixC,
read_address_matrixC =>
write_addressvResult_to_read_address_matrixC,
input_matrixA_values =>
output_data_matrixC_to_input_matrixA_values,
output_data_matrixC => output_data_matrixC_to_matrix_values,
enable_reduce => enable_reduce_v_to_enable_control,
input_to_matrixC => input_to_matrixC_hresult,
writeEnable_matrixC => writeEnable_matrixC_vresult,
address_matrixA => address_matrixA_vresult,
address_matrixB => address_matrixB_vresult,
address_matrixC => address_matrixC_vresult,
matrixA_output => matrixA_output_vresult,
matrixB_output => matrixB_output_vresult,
control_adder_signal => control_adder_signal_vresult,
mult_ready => mult_ready_vresult,
mult_done => mult_done_vresult);

evresultreducecontrolbooth1 : vresultreducecontrolbooth port map(
clk => clk,
C.4 Reduce Matrix Function Top-Level 2 Booth Multiplier

Listing C.4: The ReduceConv2ModuleBooth2.vhd VHDL file.
(appendix3/ReduceConv2ModuleBooth2.vhd)

-- Capt. Jason Shirley
-- This is the Reduce Matrix Function Top-Level Design 2 using the
-- Booth Multiplier. These Modules are structurally connected.

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

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

entity reduceconv2_modulebooth2 is
  generic (address_width: integer := 8;
            data_width: integer := 24);
  port (clk : in std_logic;
         reset : in std_logic;
         enable_matrixA : in std_logic;
         enable_conv2_module : in std_logic;
         read_address_matrixA : in std_logic_vector (... address_width - 1 downto 0);
         read_address_matrixReduce : in std_logic_vector (... address_width - 1 downto 0);
input_matrixA_values : in std_logic_vector (
  data_width - 1 downto 0);
reduce_module_finished : out std_logic;
output_data_matrixReduce : out std_logic_vector(
  data_width - 1 downto 0);
mult_ready_hresult : out std_logic;
mult_done_hresult : out std_logic;
mult_ready_vresult : out std_logic;
mult_done_vresult : out std_logic
);
end reduceconv2_modulebooth2;

architecture structure_reduceconv2_modulebooth2 of ...
  reduceconv2_modulebooth2 is
component hresultcontrolbooth2 is
  generic ( address_width : integer := 8;
    n1: integer := 5;
    n2: integer := 11) ;
  -- address_width must be an even number
  -- n1 and n2 are the size of the input matrix conv2 with the
  -- 1x5 mask example, input 6x8 matrix, mask 1x5, new conv2 matrix
  -- equals Conv2 (n1xn2) = (6+1-1) x (8+5-1), therefore n1 = 6,
  -- n2 = 12, but the matrix starts at 0 so, n1 = (6-1) = 5,
  -- n2 = (12-1) = 11.
  port ( clk : in std_logic;
    reset : in std_logic;
    enable_control : in std_logic;
    update_reg_enable : out std_logic;
    writeEnable_matrixC : out std_logic;
    reset_reg : out std_logic;
    enable_reduce : out std_logic;
    execute : out std_logic;
    read_address_matrixA : out std_logic_vector (
      address_width - 1 downto 0);
    read_address_matrixB : out std_logic_vector (
      address_width - 1 downto 0);
    write_address_matrixC : out std_logic_vector (
      address_width - 1 downto 0);
    mux_enable : out std_logic
  );
end component hresultcontrolbooth2;

component hresultreducecontrolbooth2 is
generic (address_width: integer := 8; -- address_width ...
    must be an even number
    n1: integer := 5;
    n2: integer := 11);
-- address_width must be an even number
-- The hResultReduceControl module reduces the original 6x8
-- input matrix in half from the number of columns from the
-- hResultControl module. Example 6 x 12 will be reduce down
-- to 6x4 n1 is the row position and n2 is the column position.
-- you want n1 and n2 to be one size smaller than the 6 x 12
-- therefore n1 should equal = 5 since you start at zero
-- and n2 should equal = 11 since you start at zero
-- The hResultReduceControl module reduces the original 6x8
-- input matrix in half from the number of columns from the
-- hResultControl module. Example 6 x 12 will be reduce down
-- to 6x4 n1 is the row position and n2 is the column position.
-- you want n1 and n2 to be one size smaller than the 6 x 12
-- therefore n1 should equal = 5 since you start at zero
-- and n2 should equal = 11 since you start at zero
port (clk : in std_logic;
    reset : in std_logic;
    enable_control : in std_logic;
    writeEnable_matrix : out std_logic;
    enable_vresult : out std_logic;
    read_address_hResult : out std_logic_vector (... address_width - 1 downto 0);
    write_address_reducehResult: out std_logic_vector (... address_width - 1 downto 0);
    mux_enable : out std_logic
    );
end component hresultreducecontrolbooth2;
-----------------------------------------------------------------
component vresultcontrolbooth2 is
    generic (address_width: integer := 8;
        n1: integer := 9;
        n2: integer := 3);
-- address_width must be an even number
-- n1 and n2 are the size of the input matrix conv2 with the
-- 5x1 mask example, input 6x4 matrix, mask 5x1, new conv2
-- matrix equals Conv2 (n1xn2) = (6+5-1)x (4+1-1),
-- therefore n1 = 10, n2 = 4, but the matrix starts at
-- zero so, n1 = (10-1) = 9, n2 = (4-1) = 3.
port (clk : in std_logic;
    reset : in std_logic;
    enable_control : in std_logic;
    update_reg_enable : out std_logic;
    writeEnable_matrixC : out std_logic;
    );
reset_reg : out std_logic;
enable_reduce : out std_logic;
execute : out std_logic;
read_address_matrixA : out std_logic_vector (address_width - 1 downto 0);
read_address_matrixB : out std_logic_vector (address_width - 1 downto 0);
write_address_matrixC : out std_logic_vector (address_width - 1 downto 0);
);
end component vresultcontrolbooth2;

component vresultreducecontrolbooth2 is

generic (address_width : integer := 8;
n1: integer := 9;
n2: integer := 3);
-- address_width must be an even number
port (clk : in std_logic;
reset : in std_logic;
enable_control : in std_logic;
writeEnable_matrix : out std_logic;
enable_vresult : out std_logic;
read_address_vResult : out std_logic_vector (address_width - 1 downto 0);
write_address_reducevResult: out std_logic_vector (address_width - 1 downto 0)
);
end component vresultreducecontrolbooth2;

component mux2to1 is

generic (address_width : integer := 8);
port (sel : in std_logic;
input1 : in std_logic_vector(address_width - 1 downto 0);
input2 : in std_logic_vector(address_width - 1 downto 0);
mux_output : out std_logic_vector(address_width - 1 downto 0)
);
end component mux2to1;

component mux1to2data_width is
generic (data_width: integer := 24);

port(sel: in std_logic;
    input: in std_logic_vector(data_width - 1 downto 0);
    mux_output1: out std_logic_vector(data_width - 1 downto 0);
    mux_output2: out std_logic_vector(data_width - 1 downto 0)
);

end component mux1to2data_width;

component mux2to1data_width is

    generic (data_width: integer := 24);
    port (sel: in std_logic;
          input1: in std_logic_vector(data_width - 1 downto 0);
          input2: in std_logic_vector(data_width - 1 downto 0);
          mux_output: out std_logic_vector(data_width - 1 downto 0)
    );

end component;

component mux8to4enable is

    port (sel: in std_logic;
          input1: in std_logic;
          input2: in std_logic;
          input3: in std_logic;
          input4: in std_logic;
          input5: in std_logic;
          input6: in std_logic;
          input7: in std_logic;
          input8: in std_logic;
          mux_output1: out std_logic;
          mux_output2: out std_logic;
          mux_output3: out std_logic;
          mux_output4: out std_logic
    );

end component;

component mux2to4enable is

    port (sel: in std_logic;
          input1: in std_logic;
          input2: in std_logic;
          mux_output1: out std_logic;
          mux_output2: out std_logic
    );

end component;

component mux2to4enable is

    port (sel: in std_logic;
          input1: in std_logic;
          input2: in std_logic;
          mux_output1: out std_logic;
          mux_output2: out std_logic
    );

end component;
component mem_matrix is
  generic (address_width: integer := 8;
            data_width: integer := 24);
  port (clk : in std_logic;
        reset : in std_logic;
        writeEnable : in std_logic;
        matrix_values : in std_logic_vector (data_width - 1 downto 0);
        write_address : in std_logic_vector (address_width - 1 downto 0);
        read_address : in std_logic_vector (address_width - 1 downto 0);
        output_matrix_values : out std_logic_vector (data_width - 1 downto 0)
    );
end component mem_matrix;

component mask1x5mem_matrix is
  generic (address_width: integer := 8;
            data_width: integer := 24);
  port (clk : in std_logic;
        reset : in std_logic;
        read_address : in std_logic_vector (address_width - 1 downto 0);
        output_matrix_values: out std_logic_vector (data_width - 1 downto 0));
end component mask1x5mem_matrix;

component mask5x1mem_matrix is
  generic (address_width: integer := 8;
            data_width: integer := 24);
  port (clk : in std_logic;
        reset : in std_logic;
        read_address : in std_logic_vector (address_width - 1 downto 0);
output_matrix_values : out std_logic_vector(data_width - 1 downto 0);
end component mask5x1mem_matrix;

component boothconv2 is
  generic (data_width : integer := 24);
  port (clk : in std_logic;
        reset : in std_logic;
        execute : in std_logic;
        update_reg_enable : in std_logic;
        input_matrixA_values : in std_logic_vector(data_width - 1 downto 0);
        input_matrixB_values : in std_logic_vector(data_width - 1 downto 0);
        output_data_matrixC : out std_logic_vector(data_width - 1 downto 0);
        mult_ready : out std_logic;
        mult_done : out std_logic);
end component boothconv2;

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

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

signal update_reg_enable_to_mux_input1, ...
  control_writeEnable_memC_to_WriteEnableMatrixC_mux_input3: ...
  std_logic;
signal control_reset_reg_to_hvcontrol_reset_mux_input2, ...
  enable_reduce_to_hResultReduceControl: std_logic;
signal mux_enable_to_mux_select, writeEnable_matrix_to_MatrixA2, ...
  enable_vresult_to_vResultControl : std_logic;
signal mux_enable_to_mux_select_matrixA2, ...
  update_reg_enable_to_mux_input5: std_logic;
signal control_writeEnable_memC_to_mux_input7: std_logic;
signal control_reset_reg_to_mux_input6, ...
  enable_reduce_to_vResultReduceControl_enable : std_logic;
signal writeEnable_matrix_to_MatrixReduce, ...
  mux_output_to_conv2_update_reg: std_logic;
signal mux_output_to_conv2_reset, ...
  mux_outputWriteEnableMatrixC_to_writeEnableMatrixC: std_logic;
signal read_address_matrixA_to_mux_input1, ...
  read_address_matrixB_to_mask1x5 : std_logic_vector(address_width - 1 downto 0);
signal write_address_matrixC_to_WriteAddressMatrix_mux_input1, ...
  write_address_reducehResult_to_mux_input1_matrixA2_address : ...
  std_logic_vector(address_width - 1 downto 0);
signal read_address_hResult_to_mux_input1_readAddressMatrixC, ...  
write_address_matrixC_to_WriteAddressMatrix_mux_input2 : ...  
std_logic_vector (address_width - 1 downto 0);  
signal write_address_reducevResult_to_mux_input2_matrixA2_address, ...  
read_address_matrixB_to_mask5x1 : std_logic_vector (...  
address_width - 1 downto 0);  
signal read_address_vResult_to_mux_input2_readAddressMatrixC, ...  
write_address_reducevResult_to_MatrixReduce : std_logic_vector (...  
address_width - 1 downto 0);  
signal mux_output_to_matrixA1, mux_output_to_matrixA2_address: ...  
std_logic_vector (address_width - 1 downto 0);  
signal read_address_matrixC_to_address_matrixC, ...  
mux_output_WriteAddressMatrixC_to_write_address_matrixC: ...  
std_logic_vector (address_width - 1 downto 0);  
signal output_matrix_values_to_mux_input1_matrixA_values, ...  
output_matrix_values_to_mux_input2_matrixA_values: ...  
std_logic_vector(data_width - 1 downto 0);  
signal mux_output_matrixA_values_to_conv2_input1, ...  
output_matrix_values_to_mux_input1_matrixB_values: ...  
std_logic_vector(data_width - 1 downto 0);  
signal output_matrix_values_to_mux_input2_matrixB_values, ...  
mux_output_matrixB_to_conv2_input2: std_logic;  
signal output_matrixC_values_to_mux_input_values, ...  
mux_output_MatrixC_values_to_matrixA2_values: std_logic_vector(...  
data_width - 1 downto 0);  
signal mux_output_MatrixC_values_to_matrixReduce_values: ...  
std_logic_vector(data_width - 1 downto 0);  
signal output_data_matrixC_to_matrixC_values: std_logic_vector(...  
data_width - 1 downto 0);  
signal execute_to_mux_input4, execute_to_mux_input8, ...  
mux_out_execute_to_boothconv2: std_logic;  
signal mult_ready_to_mux_input1, mult_done_to_mux_input2: ...  
std_logic;  
begin  
 hresultcontrolbooth1: hresultcontrolbooth2 port map (clk => clk,  
 reset => reset,  
 enable_control => enable_conv2_module,  
 update_reg_enable => update_reg_enable_to_mux_input1,  
 writeEnable_matrixC =>  
 control_writeEnable_memC_to_WriteEnableMatrixC_mux_input3,  
 reset_reg => control_reset_reg_to_hvcontrol_reset_mux_input2,  
 enable_reduce => enable_reduce_to_hResultReduceControl,  
 execute => execute_to_mux_input4,  
 read_address_matrixA => read_address_matrixA_to_mux_input1,  
 read_address_matrixB => read_address_matrixB_to_mask1x5,  
 write_address_matrixC =>  
 write_address_matrixC_to_WriteAddressMatrix_mux_input1,  
mux_enable => mux_enable_to_mux_select);
hresultreducecontrolbooth1 : hresultreducecontrolbooth2 port map(
clk => clk,
reset => reset,
enable_control => enable_reduce_to_hResultReduceControl,
writeEnable_matrix => writeEnable_matrix_to_MatrixA2,
read_address_hResult =>
read_address_hResult_to_mux_input1_readAddressMatrixC,
write_address_reducedResult =>
write_address_reducedResult_to_mux_input1_matrixA2_address,
mux_enable => mux_enable_to_mux_select_matrixA2);

vresultcontrolbooth1 : vresultcontrolbooth2 port map (clk => clk,
reset => reset,
enable_control => enable_vresult_to_vResultControl,
update_reg_enable => update_reg_enable_to_mux_input5,
writeEnable_matrixC => control_writeEnable_memC_to_mux_input7,
reset_reg => control_reset_reg_to_mux_input6,
enable_reduce => enable_reduce_to_vResultReduceControl_enable,
execute => execute_to_mux_input8,
read_address_matrixA =>
write_address_reductionvResult_to_mux_input2_matrixA2_address,
read_address_matrixB => read_address_matrixB_to_mask5x1,
write_address_matrixC =>
write_address_matrixC_to_WriteAddressMatrix_mux_input2);

vresultreducecontrolbooth1 : vresultreducecontrolbooth2 port map (
clk => clk,
reset => reset,
enable_control => enable_reduce_to_vResultReduceControl,
writeEnable_matrix => writeEnable_matrix_to_MatrixReduce,
read_address_vResult =>
read_address_vResult_to_mux_input2_readAddressMatrixC,
write_address_reductionvResult =>
write_address_reductionvResult_to_MatrixReduce);

mux2to1_matrixA1 : mux2to1 port map(sel => enable_matrixA,
input1 => read_address_matrixA_to_mux_input1,
input2 => read_address_matrixA,
mux_output => mux_output_to_matrixA1);

mux2to1_matrixA_values : mux2to1data_width port map(
 sel => mux_enable_to_mux_select,
 input1 => output_matrix_values_to_mux_input1_matrixA_values,
 input2 => output_matrix_values_to_mux_input2_matrixA_values,

mux_output => mux_output_matrixA_values_to_conv2_input1);

mux2to1_matrixB_values : mux2to1data_width port map(
  sel => mux_enable_to_mux_select,
  input1 => output_matrix_values_to_mux_input1_matrixB_values,
  input2 => output_matrix_values_to_mux_input2_matrixB_values,
  mux_output => mux_output_matrixB_to_conv2_input2);

mux8to4_Enable1: mux8to4enable port map(
  sel => mux_enable_to_mux_select,
  input1 => update_reg_enable_to_mux_input1,
  input2 => control_reset_reg_to_hvcontrol_reset_mux_input2,
  input3 =>
    control_writeEnable_memC_to_WriteEnableMatrixC_mux_input3,
  input4 => execute_to_mux_input4,
  input5 => update_reg_enable_to_mux_input5,
  input6 => control_reset_reg_to_mux_input6,
  input7 => control_writeEnable_memC_to_mux_input7,
  input8 => execute_to_mux_input8,
  mux_output1 => mux_output_to_conv2_update_reg,
  mux_output2 => mux_output_to_conv2_reset,
  mux_output3 => mux_out_writeEnableMatrixC_to_writeEnableMatrixC,
  mux_output4 => mux_out_execute_to_boothconv2);

mux2to4_Enable1 : mux2to4enable port map(
  sel => mux_enable_to_mux_select,
  input1 => mult_ready_to_mux_input1,
  input2 => mult_done_to_mux_input2,
  mux_output1 => mult_ready_hresult,
  mux_output2 => mult_done_hresult,
  mux_output3 => mult_ready_vresult,
  mux_output4 => mult_done_vresult);

mux2to1_ReadAddressMatrixC: mux2to1 port map(
  sel => mux_enable_to_mux_select_matrixA2,
  input1 => read_address_hResult_to_mux_input1_readAddressMatrixC,
  input2 => read_address_vResult_to_mux_input2_readAddressMatrixC,
  mux_output => read_address_matrixC_to_address_matrixC);

mux2to1_WriteAddressMatrixC: mux2to1 port map(
  sel => mux_enable_to_mux_select,
  input1 => write_address_matrixC_to_WriteAddressMatrix_mux_input1,
  input2 => write_address_matrixC_to_WriteAddressMatrix_mux_input2,
  mux_output =>
    mux_output_WriteAddressMatrixC_to_write_address_matrixC);
mux1to2_matrixA2.ReduceMatrix_values: mux1to2data_width port map(
  sel => mux_enable_to_mux_select_matrixA2,
  input => output_matrixC_values_to_mux_input_values,
  mux_output1 => mux_output_MatrixC_values_to_matrixA2_values,
 mux_output2 => mux_output_MatrixC_values_to_matrixReduce_values);

mem_matrixA1: mem_matrix port map (clk => clk, reset => reset,
  writeEnable => enable_matrixA,
  matrix_values => input_matrixA_values,
  read_address => mux_output_to_matrixA1,
  output_matrix_values =>
  output_matrix_values_to_mux_input1_matrixA_values);

mem_matrixA2: mem_matrix port map (clk => clk, reset => reset,
  writeEnable => writeEnable_matrix_to_MatrixA2,
  matrix_values => mux_output_MatrixC_values_to_matrixA2_values,
  write_address =>
  write_address_reducehResult_to_mux_input1_matrixA2_address,
  output_matrix_values =>
  output_matrix_values_to_mux_input2_matrixA_values);

mask1x5mem_matrix0: mask1x5mem_matrix port map (clk => clk,
  reset => reset,
  read_address => read_address_matrixB_to_mask1x5,
  output_matrix_values =>
  output_matrix_values_to_mux_input1_matrixB_values);

mask5x1mem_matrix0: mask5x1mem_matrix port map (clk => clk,
  reset => reset,
  read_address => read_address_matrixB_to_mask5x1,
  output_matrix_values =>
  output_matrix_values_to_mux_input2_matrixB_values);

boothconv2_1: boothconv2 port map (clk => clk,
  reset => mux_output_to_conv2_reset,
  execute => mux_out_execute_to_boothconv2,
  update_reg_enable => mux_output_to_conv2_update_reg,
  input_matrixA_values => mux_output_matrixA_values_to_conv2_input1,
  input_matrixB_values =>
  mux_output_matrixB_to_conv2_input2,
  output_data_matrixC => output_data_matrixC_to_matrixC_values,
  mult_ready => mult_ready_to_mux_input1,
  mult_done => mult_done_to_mux_input2);
mem_matrixC: mem_matrix port map (clk => clk, reset => reset, writeEnable => mux_out_writeEnableMatrixC_to_writeEnableMatrixC, matrix_values => output_data_matrixC_to_matrixC_values, write_address => mux_output_WriteAddressMatrixC_to_write_address_matrixC, read_address => read_address_matrixC_to_address_matrixC, output_matrix_values => output_matrixC_values_to_mux_input_values);

-----------------------------------------------------------------
mem_matrixReduce: mem_matrix port map (clk => clk, reset => reset, writeEnable => writeEnable_matrix_to_MatrixReduce, matrix_values => mux_output_MatrixC_values_to_matrixReduce_values, write_address => write_address_reducevResult_to_MatrixReduce, read_address => read_address_matrixReduce, output_matrix_values => output_data_matrixReduce);

end_structure_reduceconv2_modulebooth2;

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

C.5 Compute Derivatives Fx using Booth Multiplier

Listing C.5: The FxComputeDerivativesBoothConv2Module.vhd VHDL file. (appendix3/FxComputeDerivativesBoothConv2Module.vhd)

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

entity fxcomputederivativesboothconv2_module is
  generic (address_width: integer := 8;
            data_width: integer := 24);
  port (clk : in std_logic;
         reset : in std_logic;
         enable_matrix_A_B : in std_logic;
         enable_conv2_module : in std_logic;
         read_address_matrixA : in std_logic_vector (address_width - 1 downto 0);
         read_address_matrixB : in std_logic_vector (address_width - 1 downto 0);
```
read_address_matrixC : in std_logic_vector (address_width - 1 downto 0);
input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
input_matrixB_values : in std_logic_vector (data_width - 1 downto 0);
output_data_matrixC : out std_logic_vector(data_width - 1 downto 0);
enable_fxcomputederivative_complete : out std_logic;
writeEnable_matrix_conv2A : out std_logic;
writeEnable_matrix_conv2B : out std_logic;
mult_ready_conv2A : out std_logic;
mult_done_conv2A : out std_logic;
mult_ready_conv2B : out std_logic;
mult_done_conv2B : out std_logic);
end fxcomputederivativesboothconv2_module;
----------------------------------------------------------
architecture structure_fxcomputederivativesboothconv2_module of ...
fxcomputederivativesboothconv2_module is
component boothconv2_module is
  generic (address_width: integer := 8;
    data_width: integer := 24);
  port (clk : in std_logic;
    reset : in std_logic;
    enable_matrixA : in std_logic;
    enable_conv2_module : in std_logic;
    read_address_matrixA : in std_logic_vector (... address_width - 1 downto 0);
    read_address_matrixC : in std_logic_vector (... address_width - 1 downto 0);
    input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
    output_data_matrixC : out std_logic_vector(data_width - 1 downto 0);
    enable_complete : out std_logic;
    writeEnable_matrix_conv2A : out std_logic;
    mult_ready : out std_logic;
    mult_done : out std_logic);
end component boothconv2_module;
----------------------------------------------------------
component adder is

generic (data_width: integer := 24);

port (input_mult_data : in std_logic_vector (data_width - 1 downto 0);
      input_reg_data : in std_logic_vector (data_width - 1 downto 0);
      output_data : out std_logic_vector (data_width - 1 downto 0));
end component adder;

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

component fxfyftcontroladder is

  generic (address_width: integer := 8; -- address_width ...
            must be an even number
            n1: integer := 5;
            n2: integer := 7);

  -- address_width must be an even number
  -- n1 and n2 are the size of the input matrix for example 6x8
  -- matrix starts at zero so, n1 = (6-1) = 5, n2 = (8-1) = 7.

  port (clk : in std_logic;
        reset : in std_logic;
        enable_control_conv2A : in std_logic;
        enable_control_conv2B : in std_logic;
        writeEnable_matrix_image1_image2 : out std_logic;
        enable_fxfy_complete : out std_logic;
        read_address_matrix_image1 : out std_logic_vector...
                              (address_width - 1 downto 0);
        read_address_matrix_image2 : out std_logic_vector...
                              (address_width - 1 downto 0);
        write_address_matrix_image1_image2 : out std_logic_vector...
                              (address_width - 1 downto 0));

end component fxfyftcontroladder;
-------------------------------------------------------------

component mem_matrix is

  generic (address_width: integer := 8;
           data_width: integer := 24);

  port (clk : in std_logic;
        reset : in std_logic;
        writeEnable : in std_logic;
        matrix_values : in std_logic_vector (data_width ... - 1 downto 0);
        write_address : in std_logic_vector (...
                        address_width - 1 downto 0));
read_address : in std_logic_vector (address_width - 1 downto 0);
output_matrix_values : out std_logic_vector(data_width - 1 downto 0);
end component mem_matrix;

----------------------------------------------------------
signal enable_complete_to_enable_control_conv2A, ...
    enable_complete_to_enable_conrol_conv2B : std_logic;
signal writeEnable_matrix_image1_image2_to_writeEnable: std_logic;
signal mux_output_matrixA, mux_output_matrixB :std_logic_vector (...
    address_width - 1 downto 0);
signal control_matrixA_to_muxA_input1, ...
    control_matrixB_to_muxB_input1 : std_logic_vector (address_width ...
    - 1 downto 0);
signal write_address_matrix_image1_image2_to_writeEnable: ...
    std_logic_vector (address_width - 1 downto 0);
signal read_address_matrix_image1_to_read_address_conv2A, ...
    std_logic_vector (address_width - 1 downto 0);
signal output_data_matrix_conv2A_to_input_mult_data, ...
    output_data_matrix_conv2B_to_input_reg_data: std_logic_vector(...
    data_width - 1 downto 0);
signal output_data_to_mem_matrix_input_values: std_logic_vector(...
    data_width - 1 downto 0);
begin
boothconv2_module1: boothconv2_module port map (clk => clk,
    reset => reset,
    enable_matrixA => enable_matrix_A_B,
    enable_conv2_module => enable_conv2_module,
    read_address_matrixA => read_address_matrixA,
    read_address_matrixC =>
    read_address_matrix_image1_to_read_address_conv2A,
    input_matrixA_values => input_matrixA_values,
    output_data_matrix =>
    output_data_matrix_conv2A_to_input_mult_data,
    enable_complete => enable_complete_to_enable_control_conv2A,
    writeEnable_matrix => writeEnable_matrix_conv2A,
    mult_ready => mult_ready_conv2A,
    mult_done => mult_done_conv2A);
boothconv2_module2: boothconv2_module port map (clk => clk,
    reset => reset,
    enable_matrixA => enable_matrix_A_B,
    enable_conv2_module => enable_conv2_module,
    read_address_matrixA => read_address_matrixB,
read_address_matrix_image2_to_read_address_conv2B,  
input_matrixA_values => input_matrixB_values,  
output_data_matrixC =>  
output_data_matrix_conv2B_to_input_reg_data,  
enable_complete => enable_complete_to_enable_control_conv2B,  
writeEnable_matrixC => writeEnable_matrix_conv2B,  
mult_ready => mult_ready_conv2B,  
mult_done => mult_done_conv2B);

ffxfyftcontroladder0: fffyftcontroladder port map (clk => clk,  
reset => reset,  
enable_control_conv2A => enable_complete_to_enable_control_conv2A,  
enable_control_conv2B => enable_complete_to_enable_control_conv2B,  
writeEnable_matrix_image1_image2 =>  
writeEnable_matrix_image1_image2_to_writeEnable,  
enable_ffxy_complete => enable_ffxcomputedervative_complete,  
read_address_matrix_image1 =>  
read_address_matrix_image2 =>  
read_address_matrix_image1_to_read_address_conv2A,  
read_address_matrix_image2_to_read_address_conv2B,  
write_address_matrix_image1_image2 =>  
write_address_matrix_image1_image2_to_writeEnable);

mem_matrixC: mem_matrix port map (clk => clk,  
reset => reset,  
writeEnable =>  
writeEnable_matrix_image1_image2_to_writeEnable,  
matrix_values => output_data_to_mem_matrix_input_values,  
write_address =>  
write_address_matrix_image1_image2_to_writeEnable,  
read_address => read_address_matrixC,  
output_matrix_values => output_data_matrixC);

adder1: adder port map (input_mult_data =>  
output_data_matrix_conv2A_to_input_mult_data,  
input_reg_data => output_data_matrix_conv2B_to_input_reg_data,  
output_data => output_data_to_mem_matrix_input_values);

end structure_ffxcomputedervativesboothconv2_module;
C.6 Compute Derivatives Fy using Booth Multiplier

(appendix3/FyComputeDerivativesBoothConv2Module.vhd)

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

--------------------------------------------------------------
entity fycomputederivativesboothconv2_module is

  generic (address_width: integer := 8;
            data_width: integer := 24);

  port ( clk : in std_logic;
         reset : in std_logic;
         enable_matrix_A_B : in std_logic;
         enable_conv2_module : in std_logic;
         read_address_matrixA : in std_logic_vector (... address_width - 1 downto 0);
         read_address_matrixB : in std_logic_vector (... address_width - 1 downto 0);
         read_address_matrixC : in std_logic_vector (... address_width - 1 downto 0);
         input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
         input_matrixB_values : in std_logic_vector (data_width - 1 downto 0);
         output_data_matrixC : out std_logic_vector(data_width - 1 downto 0);
         enable_fycomputederivative_complete : out std_logic;
         writeEnable_matrix_conv2A : out std_logic;
         writeEnable_matrix_conv2B : out std_logic;
         mult_ready_conv2A : out std_logic;
         mult_done_conv2A : out std_logic;
         mult_ready_conv2B : out std_logic;
         mult_done_conv2B : out std_logic
       );

end fycomputederivativesboothconv2_module;
--------------------------------------------------------------
```

107
architecture structure_fycomputederivativesboothconv2_module of fycomputederivativesboothconv2_module is
component boothconv2_module is
  generic (address_width: integer := 8;
            data_width: integer := 24);
  port (clk : in std_logic;
        reset : in std_logic;
        enable_matrixA : in std_logic;
        enable_conv2_module : in std_logic;
        read_address_matrixA : in std_logic_vector (address_width - 1 downto 0);
        read_address_matrixC : in std_logic_vector (address_width - 1 downto 0);
        input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
        output_data_matrixC : out std_logic_vector (data_width - 1 downto 0);
        enable_complete : out std_logic;
        writeEnable_matrixC : out std_logic;
        mult_ready : out std_logic;
        mult_done : out std_logic
    );
end component boothconv2_module;

--------------------------------------------------------------
component adder is
  generic (data_width: integer := 24);
  port (input_mult_data : in std_logic_vector (data_width - 1 downto 0);
        input_reg_data : in std_logic_vector (data_width - 1 downto 0);
        output_data : out std_logic_vector (data_width - 1 downto 0)
    );
end component adder;

--------------------------------------------------------------
component fxfyftcontroladder is
  generic (address_width: integer := 8;
           n1: integer := 5;
           n2: integer := 7);
  -- address_width must be an even number
  -- n1 and n2 are the size of the input matrix for example 6x8
-- matrix starts at zero so, n1 = (6-1) = 5, n2 = (8-1) = 7.

port (clk : in std_logic;
    reset : in std_logic;
    enable_control_conv2A : in std_logic;
    enable_control_conv2B : in std_logic;
    writeEnable_matrix_image1_image2 : out std_logic;
    enable_fxfy_complete : out std_logic;
    read_address_matrix_image1 : out std_logic_vector(address_width - 1 downto 0);
    read_address_matrix_image2 : out std_logic_vector(address_width - 1 downto 0);
    write_address_matrix_image1_image2 : out std_logic_vector(address_width - 1 downto 0)
);

end component fxfyftcontroladder;

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

component mem_matrix is

    generic (address_width : integer := 8;
        data_width : integer := 24);

    port (clk : in std_logic;
        reset : in std_logic;
        writeEnable : in std_logic;
        matrix_values : in std_logic_vector(...
            data_width - 1 downto 0);
        write_address : in std_logic_vector(...
            address_width - 1 downto 0);
        read_address : in std_logic_vector(...
            address_width - 1 downto 0);
        output_matrix_values : out std_logic_vector(...
            data_width - 1 downto 0)
    );

end component mem_matrix;

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

signal enable_complete_to_enable_control_conv2A, ...
    enable_complete_to_enable_conrol_conv2B: std_logic;
signal writeEnable_matrix_image1_image2_to_writeEnable: std_logic;
signal mux_ouput_matrixA, mux_ouput_matrixB: std_logic_vector(...
    address_width - 1 downto 0);
signal control_matrixA_to_muxA_input1, ...
    control_matrixB_to_muxB_input1:std_logic_vector (address_width ...
        - 1 downto 0);
signal write_address_matrix_image1_image2_to_writeEnable:...
    std_logic_vector (address_width - 1 downto 0);
signal read_address_matrix_image1_to_read_address_conv2A, ...
    read_address_matrix_image2_to_read_address_conv2B:
    std_logic_vector (address_width - 1 downto 0);
signal output_data_matrix_conv2A_to_input_mult_data, ...
    output_data_matrix_conv2B_to_input_reg_data: std_logic_vector(...
    data_width - 1 downto 0);
signal output_data_to_mem_matrix_input_values: std_logic_vector(...
    data_width - 1 downto 0);

begin
    boothconv2_module1: boothconv2_module port map (clk => clk,
        reset => reset,
        enable_matrixA => enable_matrix_A_B,
        enable_conv2_module => enable_conv2_module,
        read_address_matrixA => read_address_matrixA,
        read_address_matrixC =>
            read_address_matrix_image1_to_read_address_conv2A,
        input_matrixA_values => input_matrixA_values,
        output_data_matrixC =>
            output_data_matrix_conv2A_to_input_mult_data,
        enable_complete => enable_complete_to_enable_control_conv2A,
        writeEnable_matrixC => writeEnable_matrix_conv2A,
        mult_ready => mult_ready_conv2A,
        mult_done => mult_done_conv2A);
    boothconv2_module2: boothconv2_module port map (clk => clk,
        reset => reset,
        enable_matrixA => enable_matrix_A_B,
        enable_conv2_module => enable_conv2_module,
        read_address_matrixA => read_address_matrixB,
        read_address_matrixC =>
            read_address_matrix_image2_to_read_address_conv2B,
        input_matrixA_values => input_matrixB_values,
        output_data_matrixC =>
            output_data_matrix_conv2B_to_input_reg_data,
        enable_complete => enable_complete_to_enable_control_conv2B,
        writeEnable_matrixC => writeEnable_matrix_conv2B,
        mult_ready => mult_ready_conv2B,
        mult_done => mult_done_conv2B);
    fxfyftcontroladder0: fxfyftcontroladder port map (clk => clk,
        reset => reset,
        enable_control_conv2A => enable_complete_to_enable_control_conv2A,
        enable_control_conv2B => enable_complete_to_enable_control_conv2B,
        writeEnable_matrix_image1_image2 =>
            writeEnable_matrix_image1_image2_to_writeEnable,
        enable_fxfy_complete => enable_fycomputedervative_complete,
        read_address_matrix_image1 =>
            read_address_matrix_image1_to_read_address_conv2A,
        read_address_matrix_image2 =>
            read_address_matrix_image2_to_read_address_conv2B,
C.7 Compute Derivatives $F_t$ using Booth Multiplier

Listing C.7: The FtComputeDerivativesBoothConv2Module.vhd VHDL file. (appendix3/FtComputeDerivativesBoothConv2Module.vhd)

```vhdl
-- Capt. Jason Shirley
-- This is the FtComputeDerivatives Module using the Booth Multiplier. These Modules are structurally connected

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

entity ftcomputederivativesboothconv2_module is
  generic (address_width : integer := 8;
            data_width : integer := 24);
  port (clk : in std_logic;
         reset : in std_logic;
         enable_matrix_A_B : in std_logic;
         enable_conv2_module : in std_logic;
         read_address_matrixA : in std_logic_vector (address_width - 1 downto 0);
         read_address_matrixB : in std_logic_vector (address_width - 1 downto 0);
```
read_address_matrixC : in std_logic_vector (address_width - 1 downto 0);
input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
input_matrixB_values : in std_logic_vector (data_width - 1 downto 0);
output_data_matrixC : out std_logic_vector (data_width - 1 downto 0);
enable_ftcomputederivative_complete : out std_logic;
writeEnable_matrix_conv2A : out std_logic;
writeEnable_matrix_conv2B : out std_logic;
mult_ready_conv2A : out std_logic;
mult_done_conv2A : out std_logic;
mult_ready_conv2B : out std_logic;
mult_done_conv2B : out std_logic);
end component boothconv2_image1_module;

end architecture structure_ftcomputederivativesboothconv2_module of ...

architecture structure_ftcomputederivativesboothconv2_module is

component boothconv2_image1_module is
  generic (address_width: integer := 8;
           data_width: integer := 24);
  port (clk : in std_logic;
        reset : in std_logic;
        enable_matrixA : in std_logic;
        enable_conv2_module : in std_logic;
        read_address_matrixA : in std_logic_vector (address_width - 1 downto 0);
        read_address_matrixC : in std_logic_vector (address_width - 1 downto 0);
        input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
        output_data_matrixC : out std_logic_vector (data_width - 1 downto 0);
        enable_complete : out std_logic;
        writeEnable_matrixC : out std_logic;
        mult_ready : out std_logic;
        mult_done : out std_logic);
end component boothconv2_image1_module;

component boothconv2_image2_module is
generic (address_width : integer := 8;
data_width : integer := 24);

port (clk : in std_logic;
reset : in std_logic;
enable_matrixA : in std_logic;
enable_conv2_module : in std_logic;
read_address_matrixA : in std_logic_vector (address_width - 1 downto 0);
read_address_matrixC : in std_logic_vector (address_width - 1 downto 0);
input_matrixA_values : in std_logic_vector (data_width - 1 downto 0);
output_data_matrixC : out std_logic_vector (data_width - 1 downto 0);
enable_complete : out std_logic;
writeEnable_matrixC : out std_logic;
mult.ready : out std_logic;
mult.done : out std_logic);

end component boothconv2_image2_module;

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

cOMPONENT adder IS

generic (data_width : integer := 24);

port (input_mult_data : in std_logic_vector (data_width - 1 downto 0);
input_reg_data : in std_logic_vector (data_width - 1 downto 0);
output_data : out std_logic_vector (data_width - 1 downto 0))
);

end component adder;

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

cOMPONENT fxfyftcontroladder IS

genrating (address_width : integer := 8;
n1 : integer := 5;
n2 : integer := 7);
-- address_width must be an even number
-- n1 and n2 are the size of the input matrix for example 6x8
-- matrix starts at zero so, n1 = (6-1) = 5, n2 = (8-1) = 7.

port (clk : in std_logic;
reset : in std_logic;
enable_control_conv2A : in std_logic;
enable_control_conv2B : in std_logic;
)
writeEnable_matrix_image1_image2 : out std_logic;
enable_fxfy_complete : out std_logic;
read_address_matrix_image1 : out std_logic_vector(address_width - 1 downto 0);
read_address_matrix_image2 : out std_logic_vector(address_width - 1 downto 0);
write_address_matrix_image1_image2 : out std_logic_vector(address_width - 1 downto 0);

end component fxfyftcontroladder;

component mem_matrix is

generic (address_width : integer := 8;
         data_width : integer := 24);

port (clk : in std_logic;
      reset : in std_logic;
      writeEnable : in std_logic;
      matrix_values : in std_logic_vector(data_width - 1 downto 0);
      write_address : in std_logic_vector(address_width - 1 downto 0);
      read_address : in std_logic_vector(address_width - 1 downto 0);
      output_matrix_values : out std_logic_vector(data_width - 1 downto 0))

end component mem_matrix;

signal enable_complete_to_enable_control_conv2A, ...
   enable_complete_to_enable_conrol_conv2B : std_logic;
signal writeEnable_matrix_image1_image2_to_writeEnable : std_logic;
signal mux_output_matrixA, mux_output_matrixB : std_logic_vector(address_width -
   1 downto 0);
signal control_matrixA_to_muxA_input1, ...
   control_matrixB_to_muxB_input1 : std_logic_vector(address_width ...
   - 1 downto 0);
signal write_address_matrix_image1_image2_to_writeEnable : ...
   std_logic_vector(address_width - 1 downto 0);
signal read_address_matrix_image1_to_read_address_conv2A, ...
   read_address_matrix_image2_to_read_address_conv2B : ...
   std_logic_vector(address_width - 1 downto 0);
signal output_data_matrix_conv2A_to_input_mult_data, ...
   output_data_matrix_conv2B_to_input_reg_data : std_logic_vector(data_width ...
   - 1 downto 0);
signal output_data_to_mem_matrix_input_values : std_logic_vector(data_width ...
   - 1 downto 0);
148 begin
149 boothconv2_image1_module1: boothconv2_image1_module port map ( clk => clk, reset => reset,
152 enable_matrixA => enable_matrix_A_B,
155 enable_conv2_module => enable_conv2_module,
156 read_address_matrixA => read_address_matrixA,
157 read_address_matrixC =>
158 read_address_matrix_image1_to_read_address_conv2A,
159 input_matrixA_values => input_matrixA_values,
160 output_data_matrixC =>
161 output_data_matrix_conv2A_to_input_mult_data,
162 enable_complete => enable_complete_to_enable_control_conv2A,
163 writeEnable_matrixC => writeEnable_matrix_conv2A,
164 mult_ready => mult_ready_conv2A,
165 mult_done => mult_done_conv2A);
166 boothconv2_image2_module2: boothconv2_image2_module port map ( clk => clk, reset => reset,
170 enable_matrixA => enable_matrix_A_B,
171 enable_conv2_module => enable_conv2_module,
173 read_address_matrixA => read_address_matrixB,
174 read_address_matrixC =>
175 read_address_matrix_image2_to_read_address_conv2B,
176 input_matrixA_values => input_matrixB_values,
178 output_data_matrixC =>
180 output_data_matrix_conv2B_to_input_reg_data,
181 enable_complete => enable_complete_to_enable_control_conv2B,
183 writeEnable_matrixC => writeEnable_matrix_conv2B,
184 mult_ready => mult_ready_conv2B,
185 mult_done => mult_done_conv2B);
188 fxfyftcontroladder0: fxfyftcontroladder port map ( clk => clk, reset => reset,
190 enable_control_conv2A =>
191 enable_complete_to_enable_control_conv2A,
193 enable_control_conv2B =>
195 enable_complete_to_enable_control_conv2B,
197 writeEnable_matrix_image1_image2 =>
199 enable_fxfy_complete => enable_ftcomputedervative_complete,
201 read_address_matrix_image1 =>
203 read_address_matrix_image1_to_read_address_conv2A,
205 read_address_matrix_image2 =>
207 read_address_matrix_image2_to_read_address_conv2B,
209 write_address_matrix_image1_image2 =>
210 write_address_matrix_image1_image2_to_writeEnable);
212 mem_matrixC: mem_matrix port map ( clk => clk, reset => reset,
writeEnable => writeEnable_matrix_image1_image2_to_writeEnable,
matrix_values => output_data_to_mem_matrix_input_values,
write_address =>
write_address_matrix_image1_image2_to_writeEnable,
read_address => read_address_matrixC,
output_matrix_values => output_data_matrixC);

adder1: adder port map (input_mult_data =>
output_data_matrix_conv2A_to_input_mult_data,
input_reg_data => output_data_matrix_conv2B_to_input_reg_data,
output_data => output_data_to_mem_matrix_input_values);
end structure_ftcomputedervativesboothconv2_module;

C.8 Matrix Transpose

Listing C.8: The TransposeMatrixModule.vhd VHDL file.
(appendix3/TransposeMatrixModule.vhd)

-- Capt. Jason Shirley
-- This is the Transpose Matrix Module
-- These Modules are structurally connected

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

entity transpose_matrix_module is
  generic (address_width: integer := 8;
            data_width: integer := 24);
  port (clk : in std_logic;
         reset : in std_logic;
         enable_transpose : in std_logic;
         enable_matrix_original : in std_logic;
         read_address_matrix_original : in std_logic_vector (address_width - 1 downto 0);
         input_matrix_original_values : in std_logic_vector (data_width - 1 downto 0);
         read_address_matrix_transpose : in std_logic_vector (address_width - 1 downto 0);
         output_data_matrix_transpose : out std_logic_vector (data_width - 1 downto 0);
         enable_transpose_complete : out std_logic);
end transpose_matrix_module;

architecture structure_transpose_matrix_module of ...
  transpose_matrix_module is

component transposecontrol is

  generic (address_width: integer := 8;
    n1: integer := 5;
    n2: integer := 7);
-- address_width must be an even number
-- n1 and n2 are the size of the input matrix for example 6x8
-- matrix starts at zero so, n1 = (6-1) = 5, n2 = (8-1) = 7.

  port ( clk : in std_logic;
    reset : in std_logic;
    enable_control : in std_logic;
    writeEnable_matrix_transpose : out std_logic;
    enable_transpose_complete : out std_logic;
    read_address_matrix_original : out std_logic_vector (address_width - 1 downto 0);
    write_address_matrix_transpose : out std_logic_vector (address_width - 1 downto 0)
  );

end component transposecontrol;

component mem_matrix is

  generic (address_width: integer := 8;
    data_width: integer := 24);

  port ( clk : in std_logic;
    reset : in std_logic;
    writeEnable : in std_logic;
    matrix_values : in std_logic_vector (data_width - 1 downto 0);
    write_address : in std_logic_vector (address_width - 1 downto 0);
    read_address : in std_logic_vector (address_width - 1 downto 0);
    output_matrix_values : out std_logic_vector(data_width - 1 downto 0)
  );

end component mem_matrix;
component mux2to1 is
  generic (address_width: integer := 8);
  port (sel : in std_logic;
       input1 : in std_logic_vector (address_width - 1 downto 0);
       input2 : in std_logic_vector (address_width - 1 downto 0);
       multiplexer_output : out std_logic_vector (address_width - 1 ... downto 0))
end component mux2to1;

begin
  transposecontrol1: transposecontrol port map (clk => clk,
               reset => reset,
               enable_control => enable_transpose,
               writeEnable_matrix_transpose =>
               writeEnable_matrix_transpose_to_mem_matrix_transpose,
               enable_transpose_complete => enable_transpose_complete,
               read_address_matrix_original =>
               control_matrix_original_to_mux_input1,
               write_address_matrix_transpose =>
               control_address_transpose_to_transpose_matrix);
  mem_matrix_orginal: mem_matrix port map (clk => clk,
               reset => reset,
               writeEnable => enable_matrix_original,
               matrix_values => input_matrix_original_values,
               write_address => mux_output_to_original_matrix,
               read_address => mux_output_to_original_matrix,
               output_matrix_values =>
               output_matrix_original_values_to_matrix_transpose_values);
  mem_matrix_transpose: mem_matrix port map (clk => clk,
### C.9 Pseudoinverse


```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
--use ieee.std_logic_signed.all;
--use ieee.std_logic_arith.all;
--------------------------------------------------------------
entity xresultbooth_module is
  generic (address_width : integer := 8;
            data_width : integer := 32);
  port (clk : in std_logic;
         reset : in std_logic;
         enable_matrixAtA : in std_logic; -- load A matrix ...
         enable_AtA_module : in std_logic; -- start AtA
         enable_matrixBvector : in std_logic; -- load B matrix ...
         read_address_matrixA : in signed (address_width - 1 downto 0);
         read_address_matrixBvectors : in signed (address_width - 1 downto 0);
```
read_address_matrixX : in signed (address_width - 1 downto 0);
input_matrixA_values : in signed (data_width - 1 downto 0);
input_matrixBvectors_values : in signed (data_width - 1 downto 0);
output_data_matrixX : out signed (data_width - 1 downto 0);
enable_finished : out std_logic;

end xresultbooth_module;

architecture structure_xresultbooth_module of xresultbooth_module is
component yresultbooth_module is
  generic (address_width: integer := 8;
            data_width: integer := 32);
  port (clk : in std_logic;
        reset : in std_logic;
        enable_matrixAtA : in std_logic; -- load A matrix ...
        enable_AtA_module : in std_logic; -- start AtA
        enable_matrixBvector : in std_logic; -- load B matrix ...
        read_address_matrixA : in signed (address_width - 1 downto 0);
        read_address_matrixBvectors : in signed (address_width - 1 downto 0);
        read_address_matrixU : in signed (address_width - 1 downto 0);
        read_address_matrixY : in signed (address_width - 1 downto 0);
        input_matrixA_values : in signed (data_width - 1 downto 0);
        input_matrixBvectors_values : in signed (data_width - 1 downto 0);
        output_data_matrixY : out signed (data_width - 1 downto 0);
        output_data_matrixU : out signed (data_width - 1 downto 0);
        enable_finished : out std_logic);
end component yresultbooth_module;

component xcontrolbooth is
generic (address_width: integer := 8;
data_width: integer := 32);

port (clk : in std_logic;
reset : in std_logic;
enable_control : in std_logic;
enable_predivider_register : out std_logic;
enable_postdivider_register: out std_logic;
enable_checkneg : out std_logic;
enable_divider : out std_logic;
mux_load_divider : out std_logic;
muxU : out std_logic;
muxUXY : out signed (1 downto 0);
writeEnable_matrixX : out std_logic;
enable_finished : out std_logic;
execute : out std_logic;
read_address_matrixU : out signed (address_width - 1 downto 0);
write_address_matrixX : out signed (address_width - 1 downto 0);
read_address_matrixY : out signed (address_width - 1 downto 0);
);

end component xcontrolbooth;
--------------------------------------------------------------
component reciprocal_top Is
Generic (high_bit : natural := 31; fraction_size : natural := 16);
Port (clk , reset, load, mux_control : In std_logic;
data_in : In signed (high_bit Downto 0); --signed
data_out : Out signed (high_bit Downto 0); --signed
overflow : Out std_logic);
end component reciprocal_top;
--------------------------------------------------------------
component mux2to1data_width is

generic (data_width: integer := 32);

port (sel : in std_logic;
input1 : in signed (data_width - 1 downto 0);
input2 : in signed (data_width - 1 downto 0);
mux_output : out signed (data_width - 1 downto 0)
);
end component mux2to1data_width;
--------------------------------------------------------------
component mux3to1data_width is
generic (data_width: integer := 32);

port(sel : in signed (1 downto 0);
    input1 : in signed (data_width - 1 downto 0);
    input2 : in signed (data_width - 1 downto 0);
    input3 : in signed (data_width - 1 downto 0);
    mux_output : out signed (data_width - 1 downto 0)
);
end component mux3toidata_width;

component mem_matrix is
    generic (address_width: integer := 8;
              data_width: integer := 32);
    port (clk : in std_logic;
          reset: in std_logic;
          writeEnable : in std_logic;
          matrix_values : in signed (data_width - 1 downto 0);
          write_address : in signed (address_width - 1 downto 0);
          read_address : in signed (address_width - 1 downto 0);
          output_matrix_values : out signed (data_width - 1 downto 0)
    );
end component mem_matrix;

component booth2bit is
    generic (data_width: integer := 32;
             shift_size: integer := 8);
    port (clk : in std_logic;
          reset : in std_logic;
          execute : in std_logic;
          input_matrixA_values : in signed (data_width - 1 downto 0);
          input_matrixB_values : in signed (data_width - 1 downto 0);
          mult_ready : out std_logic;
          booth2bit_output : out signed (data_width - 1 downto 0);
          mult_done : out std_logic
    );
component booth2bit;
component checkneg is
  generic (data_width: integer := 32);
  port (clk : in std_logic;
        enable : in std_logic;
        matrix_value : in signed (data_width - 1 downto 0);
        output_value : out signed(data_width - 1 downto 0)
    );
end component checkneg;
component adder is
  generic (data_width: integer := 32);
  port (input_mult_data : in signed (data_width - 1 downto 0);
        input_reg_data : in signed (data_width - 1 downto 0);
        output_data : out signed (data_width - 1 downto 0)
    );
end component adder;
component dff is
  generic (data_width: integer := 32);
  port (clk : in std_logic;
        enable : in std_logic;
        input_data : in signed (data_width - 1 downto 0);
        output_data : out signed (data_width - 1 downto 0)
    );
end component dff;

signal enable_finished_to_xcontrol,...
  enable_preadividercontrol_to_preadivider_diff : std_logic;
signal enable_postdivider_registercontrol_to_postdivider_diff: ...
  std_logic;
signal enable_divider_to_load, mux_load_divider_to_mux_control: ...
  std_logic;
signal execute_to_execute, muxU_to_mucU, ...
  writeEnable_matrix_to_write_memoryX: std_logic;
signal muxUXY_to_muxUXY : signed (1 downto 0);
signal output_data_to_data_input, data_out_to_post_divider_dff: ...
    signed (data_width - 1 downto 0);
signal output_data_to_mux_input1, ...
    output_data_matrixU_to_predivider_dff: signed (data_width - 1 ... 
downto 0);
signal read_address_matrixU_control_to_read_address_matrixU: signed...
    (address_width - 1 downto 0);
signal read_address_matrixY_control_to_read_address_matrixY: signed...
    (address_width - 1 downto 0);
signal write_address_matrixX_control_to_write_address_matrixX: ...
    signed (address_width - 1 downto 0);
signal mux_output_to_boothinputA, output_data_matrixY_to_muxinput1...
    : signed (data_width... - 1 downto 0);
signal booth2bit_output_to_input_value_memoryX: signed (data_width...
    - 1 downto 0);
signal mux_output_to_booth_inputB, output_data_to_muxinput3: ...
    signed (data_width - 1 downto 0);
signal checkneg_output_value_to_input_data_dff, ...
    dff_output_to_adder_inputA: signed (data_width - 1 downto 0);
signal enable_checkneg_to_checkneg: std_logic;

begin

yresultbooth_module1 : yresultbooth_module port map (clk => clk,
reset => reset,
enable_matrixAtA => enable_matrixAtA,
enable_AtA_module => enable_AtA_module,
enable_matrixBvector => enable_matrixBvector,
read_address_matrixA => read_address_matrixA,
read_address_matrixBvectors => read_address_matrixBvectors,
read_address_matrixU =>
read_address_matrixU_control_to_read_address_matrixU,
read_address_matrixY =>
read_address_matrixY_control_to_read_address_matrixY,
input_matrixA_values => input_matrixA_values,
input_matrixBvectors_values => input_matrixBvectors_values,
output_data_matrixY => output_data_matrixY_to_muxinput1,
output_data_matrixU => output_data_matrixU_to_predivider_dff,
enable_finished => enable_finished_to_xcontrol);

xcontrolbooth1 : xcontrolbooth port map(clk => clk,
reset => reset,
enable_control => enable_finished_to_xcontrol,
enable_preadivder_register =>
enable_preadivdercontrol_to_preadivder_dff,
enable_preadivder_register =>
enable_postdivder_registercontrol_to_postdivder_dff,
enable_checkneg => enable_checkneg_to_checkneg,
enable_divider => enable_divider_to_load,
mux_load_divider => mux_load_divider_to_mux_control,
muxU => muxU_to_muxU,
muxUXY => muxUXY_to_muxUXY,
writeEnable_matrixX => writeEnable_matrix_to_write_memoryX,
enable_finished => enable_finished,
execute => execute_to_execute,
read_address_matrixU =>
write_address_matrixU_control_to_write_address_matrixU,
write_address_matrixX =>
write_address_matrixX_control_to_write_address_matrixX,
read_address_matrixY =>
read_address_matrixY_control_to_read_address_matrixY);

reciprocal_top1: reciprocal_top port map (clk => clk,
reset => reset,
load => enable_divider_to_load,
mux_control => mux_load_divider_to_mux_control,
data_in => output_data_to_data_input,
data_out => data_out_to_post_divider_dff,
overflow => open);

mux2to1data_widthY: mux2to1data_width port map
sel => muxU_to_muxU,
input1 => output_data_to_mux_input1,
input2 => output_data_matrixU_to_predivider_dff,
input3 => output_data_to_muxinput3,
mux_output => mux_output_to_boothinputA);

mux3to1data_widthUXY: mux3to1data_width port map
(sel => muxUXY_to_muxUXY,
input1 => output_data_matrixY_to_muxinput1,
input2 => booth2bit_output_to_input_value_memoryX,
input3 => output_data_to_muxinput3,
mux_output => mux_output_to_booth_inputB);

mem_matrixX: mem_matrix port map (clk => clk,
reset => reset,
writeEnable => writeEnable_matrix_to_write_memoryX,
matrix_values => booth2bit_output_to_input_value_memoryX,
write_address =>
write_address_matrixX_control_to_write_address_matrixX,
read_address => read_address_matrixX,
output_matrix_values => output_data_matrixX);

booth2bit1: booth2bit port map(clk => clk,
reset => reset,
execute => execute_to_execute,
input_matrixA_values => mux_output_to_boothinputA,
input_matrixB_values => mux_output_to_booth_inputB,
mult_ready => open,--
booth2bit_output => booth2bit_output_to_input_value_memoryX,
mult_done => open);--
adder1: adder port map(
input_mult_data => dff_output_to_adder_inputA,
input_reg_data => output_data_matrixY_to_muxinput1,
output_data => output_data_to_muxinput3);

checkneg1 : checkneg port map(clk => clk,
enable => enable_checkneg_to_checkneg,
matrix_value => booth2bit_output_to_input_value_memoryX,
output_value => checkneg_output_value_to_input_data_dff);

dffpredivider : dff port map(clk => clk,
enable => enable_predividercontrol_to_predivider_dff,
input_data => output_data_matrixU_to_predivider_dff,
output_data => output_data_to_data_input);

dffpostdivider : dff port map(clk => clk,
enable =>
enable_postdivider_registercontrol_to_postdivider_dff,
input_data => data_out_to_post_divider_dff,
output_data => output_data_to_mux_input1);

dffpostcheckneg : dff port map(clk => clk,
enable =>
enable_postdivider_registercontrol_to_postdivider_dff,
input_data => checkneg_output_value_to_input_data_dff,
output_data => dff_output_to_adder_inputA);

end structure_xresultbooth_module;

--------------------------------------------------------------------...


Vita

Captain Jason W. Shirley graduated from Valencia High School in Placentia, CA in 1997. He attended Fullerton Junior College for three years before transferring to California State Polytechnic University of Pomona as an Electrical Engineering student, and graduated in 2003 with a Bachelor’s of Science in Electrical Engineering. Shortly after, Captain Shirley joined the US Air Force and was commissioned as an officer in September 2003 through Officer’s Training School. Captain Shirley’s first assignment was at the Oklahoma City Air Logistics Center at Tinker Air Force Base, OK. During his tour at Tinker he earned a Master’s of Science in Aerospace Administration from Southeastern Oklahoma State University in 2006. His second assignment was at the Air Force Institute of Technology (AFIT) to complete a Master’s of Science in Electrical Engineering. He will graduate in March of 2008.

Permanent address: 2950 Hobson Way
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433
14. ABSTRACT
There is a need to expedite the process of designing military hardware to stay ahead of the adversary. The core of this project was to build reusable, synthesizable libraries to make this a possibility. In order to build these libraries, Matlab commands and functions, such as Conv2, Round, Floor, Pinv, etc., had to be converted into reusable VHDL modules. These modules make up reusable libraries for the Mission Specific Process (MSP) which will support AFRL/RY. The MSP allows the VLSI design process to be completed in a mere matter of days or months using an FPGA or ASIC design, as opposed to the current way of developing a system which can take 1-2 years to complete. By having the libraries built, the components can be implemented in an FPGA or ASIC design over and over again. The libraries make it possible to make upgrades to weapons systems to meet the ever-changing needs the War Fighter faces. MSP makes it possible to develop various algorithms, including algorithms implemented in Matlab. The MSP libraries were built and tested using TSMC 250-nm technology library from the Taiwan Semiconductor Manufacturing Company. They were also synthesized for an FPGA. The modules were all synthesized using the CAD tools from Cadence and Mentor Graphics. Power, area, and delay results for each module were presented.

15. SUBJECT TERMS
Mission Specific Process, Field Programmable Gate Array, Optical Flow, VHDL

16. SECURITY CLASSIFICATION OF:
17. LIMITATION OF ABSTRACT
18. NUMBER OF PAGES

<table>
<thead>
<tr>
<th>REPORT</th>
<th>ABSTRACT</th>
<th>c. THIS PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

19a. NAME OF RESPONSIBLE PERSON
Yong C. Kim, PhD (ENG)

19b. TELEPHONE NUMBER (Include area code)
(937) 255-3636, ext 4620; email: Yong.Kim@afit.edu

Approved for public release; distribution unlimited