Design of Custom Instruction Set for FFT Using FPGA-Based Nios Processors

Divya Lakshmi Sunkara
THE FLORIDA STATE UNIVERSITY

COLLEGE OF ENGINEERING

DESIGN OF CUSTOM INSTRUCTION SET FOR FFT
USING FPGA-BASED NIOS PROCESSORS

By
DIVYA LAKSHMI SUNKARA

A Thesis submitted to the
Department of Electrical and Computer Engineering
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 2004
The members of the Committee approve the Thesis of DivyaLakshmi Sunkara defended on June 17th, 2004.

____________________________________
Uwe Meyer-Baese
Professor Directing Thesis

____________________________________
Anke Meyer-Baese
Committee Member

____________________________________
Shonda Walker
Committee Member

Approved:

____________________________________
Reginald J Perry, Chair, Department of Electrical and Computer Engineering

The Office of Graduate Studies has verified and approved the above named committee members.
ACKNOWLEDGEMENTS

I am thankful to my major professor Dr. Uwe Mayer-Baese for his guidance, advice and constant support throughout my thesis work. I would like to thank him for being my adviser here at Florida State University. I would like to thank my committee members Dr. Anke Mayer-Baese and Dr. Shonda Walker for their support.

I am thankful to my parents and my family members for their love and affection. I would like to thank my beloved husband Madhusudhana Reddy for his cooperation and support throughout my stay in Florida State University.

I wish to thank the administrative staff of the Electrical and Computer Engineering Department for their support. Last but not the least, I would like to thank my friends here at Florida State University.
# TABLE OF CONTENTS

List of Tables ................................................................................................................................. vi
List of Figures ............................................................................................................................... vii
Abstract .......................................................................................................................................... ix

1 INTRODUCTION ..................................................................................................................... 1
   1.1 Introduction ..................................................................................................................... 1
   1.2 Overview of Nios Embedded Processor ................................................................. 1
   1.3 Outline Of Thesis ........................................................................................................... 6

2 FFT ALGORITHM ................................................................................................................... 7
   2.1 Importance of Fourier Transforms ............................................................................. 7
   2.2 Custom Implementation Advantages Of Radix-2 FFT .............................................. 8
   2.3 Radix-2 FFT Algorithm ............................................................................................. 9
      2.3.1 Decimation-in-time Radix-2 FFT Algorithm ...................................................... 9
      2.3.2 Decimation-in-frequency radix-2 FFT Algorithm ............................................. 14

3 HARDWARE FEATURES OF NIOS DEVELOPMENT BOARD ....................................... 19
   3.1 Hardware Features .................................................................................................... 19
   3.2 Functional Overview ............................................................................................... 29

4 NIOS SOFTWARE AND HARDWARE DEVELOPMENT TOOLS ................................... 30
   4.1 The Quartus II Design Software .............................................................................. 30
   4.2 Nios Embedded Processor Software Development Tools ...................................... 33
   4.3 The Gnupro Toolkit ................................................................................................. 37
   4.4 Third-Party Software Evaluation Tools ................................................................... 37
# Table of Contents

5  CUSTOM IMPLEMENTATION OF DIF RADIX-2 FFT .............................................................. 38  
5.1  Development Flow Considerations for Custom Implementation ............................... 38  
5.2  Custom Instruction Features ....................................................................................... 40  
5.3  Implementation of Butterfly Processor as Custom Instruction ................................. 47  

6  DISCUSSION OF RESULTS ................................................................................................. 58  
6.1  Performance ................................................................................................................ 58  
6.2  Quantization Error ...................................................................................................... 63  

7  CONCLUSIONS AND FUTURE WORK .............................................................................. 65  
7.1  Conclusion .................................................................................................................. 65  
7.2  Future work.................................................................................................................. 66  

APPENDIX A - PROGRAM FOR DIF RADIX-2 FFT ........................................................... 67  
APPENDIX B - VHDL PROGRAMS FOR BUTTERFLY COMPUTATION ................................. 80  
APPENDIX C - EXCALIBUR.H HEADER FILE ................................................................... 85  
APPENDIX D - NIOS COMPILER AND ASSEMBLER OUTPUT ............................................. 100  
APPENDIX E - PROGRAM FOR PERFORMANCE EVALUATION ....................................... 101  

REFERENCES ........................................................................................................................ 106  
BIOGRAPHICAL SKETCH ...................................................................................................... 108
LIST OF TABLES

Table 1.1 Multiplier Optimizations ................................................................. 5
Table 2.1 Characteristics of DIF radix-2 Algorithm ........................................ 18
Table 3.1 APEX20K200EFC484 Features ....................................................... 21
Table 3.2 Flash Memory Allocation ................................................................. 25
Table 5.1 Ports for Combinatorial Custom Logic ........................................... 43
Table 5.2 Control Signal Ports for Sequential Custom Logic ....................... 44
Table 5.3 Prefix port ...................................................................................... 45
Table 5.4 User Opcode, Type and Format .................................................... 46
Table 5.5 Physical Ports used for Custom Implementation of Butterfly Processor 48
Table 6.1 Speed Increase for Single Butterfly ............................................... 59
Table 6.2 Clock Cycles for Different Length FFTs ......................................... 60
Table 6.3 Comparison of Output Results with Test Bench .......................... 63
Table 6.4 Average Quantization Error of Different Length FFTs ................ 64
LIST OF FIGURES

Figure 1.1   Nios Processor Core ................................................................. 2
Figure 1.2   Nios CPU Pipelined Architecture .............................................. 3
Figure 2.1   Signal flow graph of eight-point DIT radix-2 FFT ....................... 13
Figure 2.2   Butterfly computation for DIT radix-2 FFT ............................... 14
Figure 2.3   Bit Reverse Index Mapping for Radix-2 FFT ............................. 14
Figure 2.4   Butterfly computation for DIF radix-2 FFT .............................. 16
Figure 2.5   Signal Flow Graph of DIF Radix –2 FFT ................................. 17
Figure 2.6   Flowchart for DIF Algorithm ..................................................... 18
Figure 3.1   Nios Development Board (APEX Edition) Features .................... 20
Figure 3.2   Pin-out Information of a UART ................................................. 23
Figure 3.3   Connect / Bypass Switches ......................................................... 24
Figure 3.4   Two-Digit Seven-Segment Display ............................................ 26
Figure 3.5   The 8-DIP Switch Block ............................................................... 26
Figure 3.6   Switches ..................................................................................... 27
Figure 3.7   Clock Circuitry ......................................................................... 28
Figure 4.1   Nios SDK Shell Window ............................................................. 34
Figure 5.1   Hardware/Software Development Flow for a Nios Processor System 39
Figure 5.2   Adding Custom Logic to the Nios ALU ...................................... 41
Figure 5.3   Custom Logic Block Interface for 32-bit Nios Processor .......... 42
Figure 5.4   Multi-Cycle Logic Timing Example (5 CPU Clock Cycles)......... 45
ABSTRACT

Nios Embedded processors provide a powerful, robust platform for developing and implementing complex algorithms. The unique custom instruction feature of Nios processors could be used to enhance the performance of these algorithms dramatically, while reducing the size and complexity of software. This feature involves implementing a part or entire algorithm in hardware and making it accessible to software through specially generated software macros known as custom instructions. Currently, fast Fourier transform (FFT) algorithms play an important role in many of the digital signal processing applications that are highly time critical. Hence there is a need to increase the performance of these algorithms. In the thesis, the decimation-in-frequency radix-2 FFT is implemented using custom instruction for the butterfly processor present in the algorithm. The performance enhancement of the custom implementation of this algorithm is then measured against software-only implementation.
CHAPTER 1
INTRODUCTION

1.1 Introduction

In the last few years, Field Programmable Logic Device (FPLD) manufacturers have used programmable logic as a medium to develop System-on-a Programmable Chip (SOPC). By combining logic, memory and configurable processor core, embedded processor FPGA solutions allow system designers to integrate an entire system on a single device. The embedded FPGA solution is especially useful in compute-intensive digital Signal processing applications. Altera embedded processor solutions include Excalibur devices that offer integrated processor subsystems and the soft-core (configurable) Nios embedded processor for Altera FPGAs. These devices include robust software support that provides easy-to-use design environment. The Nios embedded processor has the flexibility to integrate memory, peripherals and other intellectual property (IP) for SOPC designs. It provides a powerful, reliable platform for developing and implementing complex algorithms. It differs from other soft-core processor solutions in the market by including custom instruction feature. This feature aids in accelerating and optimizing Nios processor based designs. The custom instruction feature of Nios processor could also be used to efficiently implement complex FFT algorithm required for various DSP applications.

1.2 Overview of Nios Embedded Processor

Nios embedded processor has set a standard for soft-core embedded processors since its introduction in the fall of 2000. It is a configurable, general purpose RISC processor that can be easily integrated with user logic and programmed into an Altera FPGA. It has a 16-bit pipelined RISC architecture with user-selectable 16 or 32-bit data paths. It also features a library of
standard soft peripherals that are configurable for a wide variety of applications. Figure 1.1 illustrates the features of a Nios processor core.

![Nios Processor Core Diagram]

**Figure 1.1 Nios Processor Core**

The processor implements the CPU with modified–Harvard memory architecture, having separate data and instruction-memory bus masters. For a 32-bit Nios processor, the instruction bus is a 16-bit wide Avalon bus master while the data bus is a 32-bit wide Avalon bus. The configurable Nios CPU is at the heart of a Nios processor-based system. The Nios CPU Architecture includes

- Instruction set
- Register file
- Cache memory
- Exception handling
- Hardware acceleration

**Instruction Set**

The Nios instruction set is tailored to support programs compiled in C and C++. The instruction set includes a set of standard arithmetic and logical operations and instructions that support bit operations, byte extraction, data movement, control flow modification, etc. It contains various addressing modes that reduce code size and increase the processor performance. The Nios CPU has pipelined architecture with five pipeline stages as shown in Figure 1.2 that is hidden from software. The pipeline stage time is limited by the slowest resource, either the ALU operation or the memory access operation.

![Nios CPU Pipelined Architecture](image)

**Figure 1.2** Nios CPU Pipelined Architecture
Register File

The Nios CPU register file includes a large general-purpose windowed register file, several machine-control registers, a program counter and the K register, which is used for instruction prefixing. The register file size is configurable to a total of 128, 256, or 512 registers. At a particular time only 32 of these general-purpose register files are accessible by software through a sliding window called the register window. The register window is divided into four register file sections that include the Input, Local, Out and Global registers respectively. When the register window slides down 16 registers for a particular instruction such as a SAVE instruction, the Out registers become the Input registers of the new window position. The global registers do not change with the movement of the window. Apart from this there are five control registers that can be addressed independently from the general-purpose registers.

Cache Memory

The Nios CPU can be configured to contain an instruction and data cache. The cache memory is used to improve CPU performance by providing a local memory system that could respond quickly to CPU-generated bus transactions. This cache memory usage is optional and its memory size is configurable. The APEX device used in the thesis does not support cache memory. The Nios 32-bit reference design uses 256Kbytes external SRAM memory present in the Nios development board, APEX edition.

Exception Handling

The Nios processor supports up to 64 vectored exceptions. These exceptions include external hardware interrupts, internal exceptions or explicit software trap instructions. By optionally disabling the support for some of these exceptions the size of the Nios system could be reduced.
Hardware Acceleration

The Nios instruction set can be configured to increase the software performance. The instruction set could be modified either by adding custom instructions or by using predefined instruction set extensions provided with the processor. Specific cycle-intensive software operations can be transferred to hardware with the use of custom instructions. Therefore, custom instruction implementation increases the system performance significantly. The Nios predefined multiplier optimizations are shown in Table 1.1 giving the number of clock cycles required for each of the multiplier option. The MUL instruction includes a hardware 16x16 bit integer multiplier. The Mstep instruction is designed in hardware to execute one step of a 16x16 bit multiply. The Software multiplication uses the C-runtime libraries to implement integer multiplication with sequences of shift and add instructions. Any of the above three multiplier options could be used to implement 16x16 multiplication in software.

<table>
<thead>
<tr>
<th>Multiplication Option</th>
<th>Clock Cycles 16x16=&gt;32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>80</td>
</tr>
<tr>
<td>MSTEP</td>
<td>18</td>
</tr>
<tr>
<td>MUL</td>
<td>3</td>
</tr>
</tbody>
</table>

In addition to the above architectural features Nios processor’s low cost, smaller FPGA footprint, higher performance and robust software development tools make Nios processor a popular soft-core embedded processor.
1.3 Outline Of Thesis

This thesis is organized as follows. Chapter 2 describes the importance of Fourier transforms and discusses in detail the advantages of custom implementation of radix-2 FFT algorithm. Both the Decimation-in-time (DIT) and Decimation-in-frequency (DIF) radix-2 FFT algorithms are described in detail.

In chapter 3, hardware features as well as the functionality of the Nios development board used for custom implementation of DIF radix-2 FFT algorithm are discussed in detail.

Chapter 4 presents the different software development tools available with the Nios development board. The functionality of each of the software tool is discussed.

Chapter 5 discusses the basic development flow and the implementation features of custom instructions. Finally, the steps involved in custom implementation of the butterfly processor are illustrated in detail.

Chapter 6 presents the discussion of results obtained from custom implementation of DIF radix-2 FFT and makes a detailed comparison with test bench results.

Chapter 7 concludes the thesis and provides suggestions for future work.
CHAPTER 2
FFT ALGORITHM

2.1 Importance of Fourier Transforms

Fourier transform is a method for representing mathematical models of time domain signals and systems in frequency domain. Engineers and mathematicians use Fourier transforms as a versatile tool in many fields of science to alter mathematical problems into those that can be easily solved. A signal can be categorized as either continuous or discrete and can further be subdivided as periodic or aperiodic. The combinations of these features generate four kinds of Fourier transforms. The Fourier transform of aperiodic-continuous signals is called the Fourier transform while that of periodic-continuous signals is known as the Fourier series. Similarly, the Fourier transform of aperiodic-discrete signal is called the discrete time Fourier transform, while that of periodic-discrete signal is called just the discrete Fourier transform. With the wide use of digital computers, which operate only on discrete (sampled) data, discrete Fourier transforms play an important role in today’s world. Digital computers use discrete Fourier transform (DFT) algorithms to analyze systems in the frequency domain and to calculate frequency response of systems with high speed. The discrete Fourier transform for an $N$-point input signal $x(n)$ is given by

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn} \quad k = 0,1,\ldots,N-1$$

where, \( W_N^{kn} = e^{-j2\pi kn/N} \)

and the inverse DFT is given by

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k)W_N^{-kn} \quad n = 0,1,\ldots,N-1$$

where, \( W_N^{-kn} = e^{j2\pi kn/N} \)
The discrete Fourier transform finds applications in linear system analysis, antenna studies, optics, random process modeling, probability theory, quantum physics and many other digital signal processing and communication fields. Performance of discrete Fourier transform algorithms plays a critical role in many of the above applications. Thus fast Fourier transforms, which are a collection of efficient algorithms used to compute the discrete Fourier transform, have received considerable attention by many mathematicians, engineers and applied scientists. Some of the fast Fourier transform algorithms include Cooley-Turkey algorithm (radix-\( r \) algorithms), Good-Thomas FFT algorithm and the Winograd FFT algorithm. Radix-2 FFT is the most popular and widely used Cooley-Turkey FFT algorithm due to its implementation advantages.

2.2 Custom Implementation Advantages Of Radix-2 FFT

Many properties must be considered in choosing an FFT algorithm for customized implementation. These constraints include possible transform length, number of multiplications, number of additions, index computation overhead, coefficient or data memory size, run time code length and implementation advantages.

The radix-2 Cooley-Turkey algorithm calculates the DFT for all power of two transform lengths and hence can calculate DFT for a wide range of possible transform lengths. Also, when the number of data points is not a power of two, the transform length can be made a power of two by simply padding the input data with zeros. The basic DFTs (two-point FFT) in radix-2 FFT can be implemented without any multiplications although subsequent DFT computations require twiddle factor multiplications. For efficient computation the twiddle factors used can be computed once and stored in a table. The total number of addition computations performed in radix-2 algorithm is considered to be fair. The index mapping for the input and output data in radix-2 FFT is a simple bit reverse mapping algorithm and can be easily implemented. Since radix-2 FFT performs all the computations in place, it does not require extra memory locations
for computational purposes. This is possible due to the fact that the memory locations used by a butterfly (basic computation in the algorithm) can be overwritten, as successive computation steps do not require the data. For complex input, the total number of memory locations required is twice the input length. The code for Radix-2 FFT is easier to implement and has small run time code length when compared with other algorithms. For custom implementation, the small butterfly processor in radix-2 FFT algorithm can be easily implemented in hardware using Very High Speed Integrated Circuit Hardware Description Language (VHDL). All the above features make radix-2 FFT the best overall choice for custom implementation.

2.3 Radix-2 FFT Algorithm

Radix-2 FFT algorithm uses the divide-and-conquer approach for efficient computation of DFT. There are two forms of radix-2 algorithms known as the decimation-in-time (DIT) and decimation-in-frequency (DIF) algorithms. Both these algorithms have the same complexity but differ in the arrangement of the computation.

2.3.1 Decimation-in-time Radix-2 FFT Algorithm

In the divide and conquer approach, the N-point DFT is decomposed into successively smaller DFT. The input signal is stored in a two dimensional array indexed by l and m where l is the row index and m is the column index. For radix-2 decimation-in-time algorithm the column length M is chosen to be N/2 and the row length L is made equal to 2, which is the radix of the algorithm. This splits N-point data sequence into two N/2-point data sequences corresponding to the even-numbered samples \( f_1(n) \) and odd-numbered samples \( f_2(n) \), that is

\[
\begin{align*}
    f_1(n) &= x(2n) \\
    f_2(n) &= x(2n + 1), \quad n = 0, 1, 2, \ldots, \frac{N}{2} - 1
\end{align*}
\] (2.1)
The above functions are obtained by decimating the input signal $x(n)$ by a factor of 2 and hence the name decimation-in-time. Expressing the $N$-point DFT in terms of the decimated sequences we get

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn} \quad k = 0,1,\ldots,N-1$$

where, $W_N^{kn} = e^{-j2\pi kn/N}$

$$= \sum_{n\text{(even)}} x(n)W_N^{kn} + \sum_{n\text{(odd)}} x(n)W_N^{kn}$$

$$= \sum_{m=0}^{(N/2)-1} x(2m)W_N^{2km} + \sum_{m=0}^{(N/2)-1} x(2m+1)W_N^{k(2m+1)} \quad (2.2)$$

With the substitution $W_N^2 = W_{N/2}$, $f_1(m) = x(2m)$ and $f_2(m) = x(2m+1)$ equation 2.2 can be written as

$$X(k) = \sum_{m=0}^{(N/2)-1} f_1(m)W_{N/2}^{km} + W_N^k \sum_{m=0}^{(N/2)-1} f_2(m)W_{N/2}^{km} \quad (2.3)$$

$$= F_1(k) + W_N^kF_2(k) \quad k = 0,1,\ldots,N-1$$

where $F_1(k)$ and $F_2(k)$ are the $N/2$-point DFTs of corresponding sequences $f_1(m)$ and $f_2(m)$. Both of these DFT sequences are periodic with period $N/2$ which results in $F_1(k + N/2) = F_1(k)$ and $F_2(k + N/2) = F_2(k)$. Since $W_N^{k+N/2} = -W_N^k$, the equation (2.3) can be expressed as

$$X(k) = F_1(k) + W_N^kF_2(k) \quad k = 0,1,\ldots,N/2-1 \quad (2.4)$$

$$X(k + N/2) = F_1(k) - W_N^kF_2(k) \quad k = 0,1,\ldots,N/2-1 \quad (2.5)$$
The direct computation of each of the functions $F_1(k)$ and $F_2(k)$ require $(N/2)^2$ complex multiplications and the computation of $W_N^k F_2(k)$ requires $N/2$ complex multiplications. Hence the total number of complex multiplications required to compute $X(k)$ reduces from $N^2$ to $N^2/2 + N/2$.

The following equations are defined to be consistent with the previous notation

\[ G_1(k) = F_1(k) \quad k = 0, 1, \ldots, \frac{N}{2} - 1 \]

\[ G_2(k) = W_N^k F_2(k) \quad k = 0, 1, \ldots, \frac{N}{2} - 1 \]

Then the DFT of $X(k)$ can be expressed as

\[ X(k) = G_1(k) + G_2(k) \quad k = 0, 1, \ldots, \frac{N}{2} - 1 \tag{2.6} \]

\[ X(k + \frac{N}{2}) = G_1(k) - G_2(k) \quad k = 0, 1, \ldots, \frac{N}{2} - 1 \]

The process of decimation in time can be repeated for each of the sequences $f_1(n)$ and $f_2(n)$. The decimation of $f_1(n)$ results in two $N/4$ point sequences as follows

\[ v_{11}(n) = f_1(2n) \quad n = 0, 1, \ldots, \frac{N}{4} - 1 \tag{2.7} \]

\[ v_{12}(n) = f_1(2n+1) \quad n = 0, 1, \ldots, \frac{N}{4} - 1 \]
and the decimation of \( f_2(n) \) results in the following \( N/4 \) point sequences.

\[
v_{21}(n) = f_2(2n) \quad n = 0, 1, \ldots, \frac{N}{4} - 1
\]

\[
v_{22}(n) = f_2(2n + 1) \quad n = 0, 1, \ldots, \frac{N}{4} - 1
\]

The \( N/2 \)-point DFTs \( F_1(k) \) and \( F_2(k) \) can be calculated from the above \( N/4 \) point DFTs using the following relations.

\[
F_1(k) = V_{11}(k) + W_{N/2}^k V_{12}(k) \quad k = 0, 1, \ldots, \frac{N}{4} - 1
\]

\[
F_1(k + \frac{N}{4}) = V_{11}(k) - W_{N/2}^k V_{12}(k) \quad k = 0, 1, \ldots, \frac{N}{4} - 1
\]

\[
F_2(k) = V_{21}(k) + W_{N/2}^k V_{22}(k) \quad k = 0, 1, \ldots, \frac{N}{4} - 1
\]

\[
F_2(k + \frac{N}{4}) = V_{21}(k) - W_{N/2}^k V_{22}(k) \quad k = 0, 1, \ldots, \frac{N}{4} - 1
\]

where \( \{V_n(k)\} \) are the \( N/4 \)-point DFTs corresponding to the sequences \( \{v_n(n)\} \). It can be observed from above that by decimating the input sequence twice, the number of complex multiplications is reduced further from \( N^2 \) to \( \frac{N^2}{4} + N \). For length \( N \) input sequence, decimation can be performed \( \log_2 N \) times until the sequences are reduced to one-point sequences. Hence the total number of complex multiplications is reduced to \( (N/2)\log_2 N \). For large \( N \), the speed improvement is significant with radix-2 FFT computations when compared to direct computations of the DFT. Figure 2.1 depicts the signal flow graph of an eight-point DIT algorithm showing the different stages, groups and butterflies.
In the signal flow graph all the arriving arrows at a node are added and the constant coefficient multiplications are symbolized by a factor at an arrow. The basic computation performed at each stage is called a butterfly shown in Figure 2.2. Groups are combinations of two or more butterflies at each stage for which the same type of twiddle factors occur. In decimation-in-time algorithm the time domain input is read in bit reverse order and the DFT output appears in the normal order. The bit reverse index transform for index 41 is shown in Figure 2.3.
2.3.2 Decimation-in-frequency radix-2 FFT Algorithm

The divide and conquer approach is also implemented in decimation-in-frequency algorithm, where the $N$-point DFT is decomposed into successively smaller DFT. In this algorithm the input signal is stored in a two dimensional array with column length $M=2$ and row length $L=N/2$. This splits $N$-point data sequence into two $N/2$-point data sequences involving the sum over the first $N/2$ data points and the last $N/2$ data points. As a result,
\[ X(k) = \sum_{n=0}^{(N/2)-1} x(n)W_N^{kn} + W_N^{Nk/2} \sum_{n=0}^{N-1} x(n)W_N^{kn} \]
\[ = \sum_{n=0}^{(N/2)-1} x(n)W_N^{kn} + W_N^{Nk/2} \sum_{n=0}^{(N/2)-1} x\left(n + \frac{N}{2}\right)W_N^{kn} \]

As \( W_N^{Nk/2} = (-1)^k \) the equation (2.11) can be written as

\[ X(k) = \sum_{n=0}^{(N/2)-1} \left[ x(n) + (-1)^k x\left(n + \frac{N}{2}\right) \right] W_N^{kn} \] *(2.12)*

The original DFT above is split into shorter DFTs and hence the name decimation-in-frequency algorithm. By decimating the frequency component \( X(k) \) and using \( W_N^2 = W_{N/2} \), we get the following even and odd numbered samples

\[ X(2k) = \sum_{n=0}^{(N/2)-1} \left[ x(n) + x\left(n + \frac{N}{2}\right) \right] W_{N/2}^{kn} \quad k = 0,1,\ldots,\frac{N}{2}-1 \] *(2.13)*

\[ X(2k+1) = \sum_{n=0}^{(N/2)-1} \left\{ x(n) - x\left(n + \frac{N}{2}\right) \right\} W_N^n W_{N/2}^{kn} \quad k = 0,1,\ldots,\frac{N}{2}-1 \] *(2.14)*

By defining the following \( N/2 \) point sequences,

\[ g_1(n) = x(n) + x\left(n + \frac{N}{2}\right) \] *(2.15)*

\[ g_2(n) = \left[ x(n) - x\left(n + \frac{N}{2}\right) \right] W_N^n \quad n = 0,1,2,\ldots,\frac{N}{2}-1 \]
Equations (2.13) and (2.14) are modified as

\[
X(2k) = \sum_{n=0}^{(N/2)-1} g_1(n) W_n^{kn} \\
X(2k+1) = \sum_{n=0}^{(N/2)-1} g_2(n) W_n^{kn}
\]  

(2.16)

The decimation process for an \( N \)-point input sequence is carried out \( \log_2 N \) times. Each decimation step rearranges the input sequence into even and odd indexed sequences. The total number of complex multiplications is therefore reduced to \( (N/2) \log_2 N \) just as in the DIT algorithm. The basic butterfly computation for this algorithm is illustrated in Figure 2.4. Figure 2.5 illustrates the signal flow graph of eight-point decimation-in-frequency radix-2 FFT algorithm. In this algorithm, the input is in normal order while the DFT output is in reverse order.

\[
A = a + b \\
B = (a - b)W_N^1
\]

**Figure 2.4**  Butterfly computation for DIF radix-2 FFT
For the thesis, decimation-in-frequency radix-2 FFT algorithm is implemented with custom instruction for the compute-intensive basic butterfly processor. The characteristic values of a DIF radix-2 algorithm required for implementation is shown in Table 2.1. The first row gives the stage number for a length $N$ FFT. The second row gives the number of groups based on the stage number while the third row gives the number of butterflies per group based on the group number. The final row shows how the twiddle factors increment based on a particular stage. In the thesis, the entire DIF FFT algorithm is written in C language for $N$-point input sequence using nested loops for the stages, groups and butterflies. The inner butterfly computation is implemented as a custom instruction, which is described in Chapter 5. The flow chart for implementing DIF algorithm in C is illustrated in Figure 2.6. The overhead for the group shown in the Figure 2.6 requires the calculation for the number of groups for each stage while the overhead for the butterfly loop requires the calculation for the number of butterflies for
each group. The C-code for DIF radix-2 FFT algorithm using custom instruction and software-only implementation for the butterfly computation are present in Appendix A.

**Table 2.1  Characteristics of DIF radix-2 Algorithm**

<table>
<thead>
<tr>
<th>Stage Number ($k$)</th>
<th>Stage (0)</th>
<th>Stage (1)</th>
<th>Stage (2)</th>
<th>….</th>
<th>Stage ($\log_2(N) - 1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of groups per stage ($p = 2^k$)</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>….</td>
<td>$N/2$</td>
</tr>
<tr>
<td>Butterflies per group ($t = N/2p$)</td>
<td>$N/2$</td>
<td>$N/4$</td>
<td>$N/8$</td>
<td>….</td>
<td>1</td>
</tr>
<tr>
<td>Increment exponent twiddle-factors</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>….</td>
<td>$N/2$</td>
</tr>
</tbody>
</table>

---

**Figure 2.6  Flowchart for DIF Algorithm**
CHAPTER 3
HARDWARE FEATURES OF NIOS DEVELOPMENT BOARD

3.1 Hardware Features

The Nios CPU development board (APEX Edition) included in the Excalibur™ Development Kit includes the following hardware features. All these components are shown in Figure 3.1.

- An APEX™ 20K200EFC484-2X device
- 1 Mbyte (512 K x 16-bit) of flash memory pre-configured with the 32-bit Nios reference design and software
- 256 Kbytes of SRAM (in two 64 K x 16-bit chips)
- On-board logic for configuring APEX device from flash memory
- 3.3-V expansion/prototype headers (access to 40 user I/Os)
- 5-V-tolerant expansion/prototype headers (access to 40 user I/Os)
- Small outline DIMM (SODIMM) socket, compatible with standard SDRAM modules
- Two IEEE-1386 peripheral component interconnect (PCI) mezzanine connectors
- One RS-232 serial connector
- One user-definable 8-bit DIP switch block
- Four user-definable push-button switches
- Dual 7-segment LED display
- Two user-controllable LEDs
- Joint test action group (JTAG) connector for ByteBlasterMV™ and MasterBlaster™ download cables
- Oscillator and zero-skew clock distribution circuitry
- Power-on reset circuitry
- Power-supply circuitry (Input: 9-V unregulated, center-negative)
APEX 20k200EFC484 designated as U1 on the development board is a Field Programmable Logic Device (FPLD) in a 484-pin FineLine BGA™ package. The apex device can be configured using a JTAG connection (JP3) with the aid of Quartus II software via a ByteBlasterMV download cable. An alternative method to configure the APEX device is to use the configuration controller (U4) that configures the device at power-up from hexout files stored in the flash memory (U3). The device is pre-configured with 32-bit Nios RISC processor system module. A 16-bit Nios system module is also available for configuration on the device. The Nios system module including the CPU and peripherals occupy between 25% and 35% of the logic on this device. Table 3.1 illustrates the APEX20K200E features.
Table 3.1 APEX20K200EFC484 Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum system gates</td>
<td>526,000</td>
</tr>
<tr>
<td>Typical gates</td>
<td>211,000</td>
</tr>
<tr>
<td>LEs</td>
<td>8,320</td>
</tr>
<tr>
<td>ESBs</td>
<td>52</td>
</tr>
<tr>
<td>Maximum RAM bits</td>
<td>106,496</td>
</tr>
<tr>
<td>Maximum macrocells</td>
<td>832</td>
</tr>
<tr>
<td>Maximum user I/O pins</td>
<td>382</td>
</tr>
</tbody>
</table>

Flash Memory Chip

U3 is a 1 Mbyte flash memory chip manufactured by Advanced Micro Devices (AMD). The flash memory chip serves two purposes. One purpose is that it can be used as general-purpose readable, memory and the other purpose is for non-volatile storage by the Nios processor implemented on the APEX device. The flash memory is mainly used to hold an APEX device configuration file, which is used by the configuration controller to load the APEX device at power-up. The 32-bit Nios reference design hexout file is pre-loaded in this flash memory. It is present as a library component in the SOPC builder. Once loaded the reference design identifies the flash memory in its address space. The Nios software includes subroutines for writing and erasing the flash memory.

Dual SRAM Chips

The two 256 Kbyte (64k X 16-bit) asynchronous SRAM chips (U14 and U15) are connected to the APEX device and serve as general-purpose zero-wait-state memory. A 32-bit wide memory can be implemented by using the two 16-bit devices in parallel. The 32-bit Nios reference design identifies these chips as 32-bit wide zero-wait-state memory. The base address
of this memory location begins at 0x00040000 and ends at location 0x0007ffff. This memory is present as a library component in the SOPC builder.

**SODIMM Connector**

The 144-pin SODIMM socket J2 connected to the APEX device enables user logic to access SDRAM. The factory configured reference design does not make use of this connector.

**Expansion Prototype Connector Headers**

Headers JP8, JP9, and JP10 have 3.3V expansion prototype connector interface. They can be collectively used as an interface to a special-function daughter card. Headers JP11, JP12 and JP13 can also be used as an interface to a special-function daughter card with a 5V interface. The reference design uses JP12 as an interface to the dual seven-segment LCD included with the Nios Development kit, featuring the Nios embedded processor.

**PMC Connectors**

The two IEEE-compliant PCI Mezzanine Connectors (PMC), PMCJN1 and PMCJN2 enable the APEX device to access PMC daughter cards. The pre-loaded Nios reference design does not utilize these connectors.

**Serial Port Connector**

The DB-9 serial connector (J3) is used for host communication with a desktop workstation using a standard 9-pin serial cable connected to one of the COM-ports. Standard high-voltage RS-232 logic levels are used by transmit (TXD) from Nios, receive (RXD) by Nios, clear to send (CTS) and ready to send (RTS) signals. The level-shifting buffer (U13) accepts or presents 3.3V version of these signals to and from the APEX device. Figure 3.2 shows the pin-out information of a UART (serial port connector) with hardware handshaking. The reference
design uses the UART with hardware handshaking for downloading software programs and for communication purposes.

![Figure 3.2 Pin-out Information of a UART](image)

**JTAG Connector**

The JTAG connector (JP3) is an interface compatible with Altera ByteBlasterMV and MasterBlaster download cables. This connection can be used to configure the Apex device (U1) with a new bitstream (such as .sof) file via a ByteBlasterMV using Quartus II software. ByteBlasterMV is a parallel cable used for downloading programming files onto the board using Quartus II software. Reprogramming the EPM7064 device (U4) with a new .pof file takes place via a JTAG connection. JTAG connection can also be used for serial communication between user-provided host software and the card plugged into PMC connectors if the card makes use of the JTAG signals that are part of the IEEE-1386 standard.

The JTAG chain can include any of the following devices using the respective switches

- (SW8) The APEX device (U1)
- (SW9) The EPM7064 configuration controller (U4)
- (SW10) A card plugged into the PMC connectors, if present.
Figure 3.3 Connect / Bypass Switches

Figure 3.3 shows that the connect/bypass switches determine whether the JTAG chain includes (connects) or excludes (bypasses) a particular device. When using JTAG connection for downloading user configuration to the APEX device during logic development and debugging, it is convenient to keep SW8 in the connect position and the other two switches (SW9 and SW10) in the bypass position.

Configuration Controller

The EPM7064 is an Altera Programmable Logic Device (PLD) that is factory-programmed as a configuration controller. It configures the APEX device (U1) from data stored in flash memory (U3). The configuration controller begins reading the data out of the flash memory in passive-parallel mode at power-up (or when the reset switch SW2 is pressed).

The hexout configuration files are optionally produced by Quartus II software that are directly suitable for download and storage in the flash memory as configuration data. The GERMS monitor program included in the preloaded 32-bit Nios reference design supports downloading hexout files from a host (such as desktop workstation) into flash memory. The configuration controller manages both the user configuration and the factory configuration.
stored in the flash memory. It attempts to load the user configuration data upon reset. If this attempt fails the APEX device is loaded with the factory configuration data. Table 3.2 shows the flash memory allocation for different configuration data. Factory configuration region of the flash memory is pre-loaded with 32-bit Nios reference design.

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100000 – 0x17FFFF</td>
<td>512 Kbytes</td>
<td>Nios instruction and nonvolatile data space.</td>
</tr>
<tr>
<td>0x180000 – 0x1BFFFF</td>
<td>256 Kbytes</td>
<td>User-defined APEX device.</td>
</tr>
<tr>
<td>0x1C0000 – 0x1FFFFF</td>
<td>256 Kbytes</td>
<td>Factory-default APEX device configuration.</td>
</tr>
</tbody>
</table>

The configuration controller changes its behavior based on the jumper (JP2). IF the jumper pins are connected the APEX device is always configured from the factory configuration. This is useful when the user configuration is non-functional. The base address for 1 Mega byte flash memory in the pre-loaded Nios reference design is 0x100000. Hence, the user configuration files should be downloaded to address 0x180000, which is the flash memory base address plus the user configuration offset.

Two-Digit Seven-Segment Display

The APEX device is connected to the seven-segment display (D1) such that each segment is individually controlled by generic purpose I/O pin. The display is driven by parallel input/output (PIO) registers and logic provided in the Nios reference design.
Switches, Buttons and LEDs

An 8-DIP-switch block (SW1) is connected to an APEX general-purpose I/O and a pull-up register. An open switch indicates logic-1 and a closed switch indicates logic-0. Figure 3.5 illustrates the 8-DIP switch block.

Switches (SW4, SW5, SW6 and SW7) are contact push-button switches, which indicate a logic-0 when pressed. Discrete LEDs LED1 and LED2 light up when the APEX device lights a logic-1 on its controlling output. Switch (SW2) is a reset switch which when pressed drives a logic-0 to U7, the power-on-reset controller. When the reset switch is pressed the configuration controller will load the APEX device with a configuration file from the flash memory. The 32-bit reference design loaded in the APEX device will begin executing the GERMS monitor, a
serial debug/download utility. Switch (SW3) is a clear switch and drives a logic-0 when pressed. The action of this clear button is based on the configuration of APEX device. The pre-loaded Nios reference design treats this switch as a CPU-reset pin. When SW3 is pressed the Nios CPU will reset and start executing the code from its boot-address. Figure 3.6 shows all the switches mentioned above. The pin numbers given in the bracket are the apex device pin numbers that are connected to the switches.

![Figure 3.6 Switches](image)

**Power-Supply Circuitry**

A 9-volt unregulated power supply runs the Nios development board. On-board circuitry generates three different regulated power levels such as 5V, 3.3V, and 1.8V. Each of these regulated voltage levels are used for specific purposes. The APEX device core utilizes the 1.8V supply as a power source. The power for all APEX device I/O pins is supplied by 3.3V power level. Devices plugged into any of the expansion connectors and daughter cards, including the PMC connectors and the SDRAM SODIMM socket also use 3.3V supply. The limitation on the total load from all externally connected 3.3V devices is 500mA. The 5V supply is available for devices plugged into the 5V tolerant expansion connectors. The total load for these devices is limited to 50mA.
Clock Circuitry

The APEX device and pins on the expansion connectors, PMC connectors and SODIMM connector are driven by a 33.333 MHz free-running oscillator and a zero-skew, point-to-point clock distribution network. As shown in Figure 3.7, the zero-skew buffer aids in distributing both the free-running clock and the clock-output from one of the APEX’s device internal PLLs (CLKLK_OUT1).

Figure 3.7  Clock Circuitry
3.2 Functional Overview

The Nios development board provides a hardware platform that enables the immediate development of embedded systems based on Altera APEX™ devices. The Nios development board is factory configured with a 32-bit Nios embedded processor system reference design. The Nios development software provided with the Excalibur development kit includes a Quartus II project directory containing the reference design example. The reference design and software that are preloaded in flash memory boot on power-up. The reference design software includes a monitor called the GERMS monitor that can be used to download and debug programs. Hence custom instructions can be immediately implemented based on the reference design that is provided with the Nios development kit.
CHAPTER 4
NIOS SOFTWARE AND HARDWARE DEVELOPMENT TOOLS

The Nios software and hardware development tools that are available in the Excalibur development kit include

- The Quartus II Design Software
- The Nios embedded processor software development tool
- The Gnupro toolkit
- Third-Party Software Evaluation tools

4.1 The Quartus II Design Software

System-on-a-programmable-chip (SOPC) hardware design is implemented with the aid of Quartus II Design Software. Using the Quartus II software the hardware designer can design files, test the design by simulating with the target FPGA device, synthesize a netlist for the design and create output configuration files for downloading to the target FPGA. Designers also use the Quartus II software to perform timing analysis on the FPGA design. The SOPC Builder system integration tool is included in the Quartus II software installation.

SOPC Builder

SOPC Builder is an automated system development tool used to define and integrate hardware systems. The SOPC Builder also generates a custom software development kit (SDK) based on the hardware system designed. Thus, the SOPC builder can be used to define an entire system that includes both hardware and software. The two principal components of the SOPC Builder are the graphical user interface (GUI) and the system generator program. The GUI
records all the hardware system settings in files known as PTF files. These PTF files are modified whenever any changes are made to the hardware system using the GUI. For instance, adding a custom instruction using the GUI to the reference 32-bit Nios CPU modifies its PTF file. Thus the GUI is considered as a specialized editor for system PTF files. The system generator is a stand-alone program that creates all the output files made by SOPC Builder. These output files include HDL logic files, C-software header and library files, simulation files etc. Nios hardware designers use the SOPC Builder to develop the Nios embedded processor system and its corresponding software development kit. Custom instructions are also implemented with the aid of SOPC Builder. The SDK developed can be used successfully to interact with fundamental system components at a low level by writing software. The SDK makes available software routines for accessing standard peripherals such as UARTs, PIOs, and DMA controllers. The standard peripherals that are present as SOPC Builder library component in the Nios development kit include UART, PIO, DMA, SPI and Nios Timer.

**Nios UART**

The UART module is used for serial communication with variable baud rate, parity, stop and data bits and optional control signals. Both UARTs Verilog HDL and VHDL source code are available for development and include the software subroutines for enabling the access to the module through software. Simple RS-232 asynchronous transmit and receive logic is implemented by the Nios UART. Pins (RxD and TxD) are used for sending and receiving signals. The Nios UART is used for downloading software on to the board and for communication between host and the target during software execution. Software communication with the UART takes place through five-memory mapped, 16-bit registers. The UART runs within a single synchronous clock input. The transmission rate of Nios UART is approximately 11,500 characters per second at 115,200 bps. The UART can be used in conjunction with Direct Memory Access (DMA) for possible data transfers between the UART and memory.
**Nios PIO**

The Nios parallel input/output module enables 1 to 32-bit parallel I/O (PIO). The PIOs source code is available for development and its corresponding software subroutines are available for easy system integration. The parallel input/output module provides a memory-mapped PIO interface between software and user-defined logic that resides both within the same device and outside the device. In the Nios development board, the PIO interface is available for the peripherals, such as seven-segment display, LCD, LED, buttons. This enables the software to interact with these peripherals using the respective software subroutines.

**Nios DMA**

The direct memory access (DMA) enables efficient data transfer between peripherals and memory by eliminating the CPU from the data path. The DMAs software subroutines and its source code are available for development and easy system integration. DMAs fixed or variable data transfers can take place between two memories, between a memory and peripheral or between two peripherals without CPU intervention.

**Nios SPI**

The serial peripheral interface (SPI) is a standard serial communication interface used in embedded systems. The SPI source code for Nios is also available for development and includes the software subroutines for proper system interaction. Many semiconductor devices are available in the market that use SPI interface. The Nios SPI peripheral is 3-wired and allows software to interact with one or more external devices. The SPI peripheral can be used as either a master or slave device. When used as a master device, the SPI can interface with up to 16 different SPI slave devices.
Nios Timer

The Nios Timer module serves its purpose as a periodic pulse generator or system watchdog timer. The timer source code is available for modification and development. The timer software subroutines are used for measuring the performance of software instructions and aid in evaluating the importance of custom instructions. The timer registers, the status register and the control registers control the operation of the timer.

4.2 Nios Embedded Processor Software Development Tools

The Nios software development tools that are present in Nios embedded processor CD-ROM include the Nios SDK and the GERMS monitor. The Nios CPU is a 32-bit RISC processor with limited set of instructions. The SOPC builder uses the Nios CPU core library component to generate the hardware and software foundations for Nios systems. The SOPC Builder generates function prototypes for software subroutines in the excalibur.h header file during custom instruction implementation. These subroutines could be used in C/C++ software programs by including the header file.

Nios SDK

The SOPC Builder generates the software development kit (SDK) for every Nios processor system generated. The Nios SDK shell provides a bash environment (UNIX-like environment) on a PC platform to develop Nios software and to communicate with the Nios development board. The Cygwin bash shell is a part of GnuPro toolkit. To generate, debug and download software the Nios SDK shell utilizes Nios-specific utilities available with the Nios development kit. The common utilities used are the nios-build and nios-run utilities for compiling, linking and executing software programs. The SDK is a collection of files with the top-level directory designated as <CPU name>_sdk (where <CPU name> is the CPU name
given in SOPC Builder). The 32-bit Nios reference design used for the project has the top-level directory named as CPU_sdk. If a system has more than one Nios CPU, each CPU has a unique custom SDK associated with it. The Nios SDK Shell is used to run various test programs on the Nios development board. Figure 4.1 depicts the Nios SDK shell window.

![Nios SDK Shell Window](image)

**Figure 4.1** Nios SDK Shell Window

**Nios SDK Directories**

The three main directories that are generated within the SDK directory include

1. inc Directory
2. lib Directory
3. src Directory
**inc Directory**

The inc directory carries include files (.h and .s) that carry peripheral addresses, register structures, function prototypes for some useful routines, interrupt numbers and assembly language macros. The .h include files are used in high-level language programs such as C/C++ and the .s include files are used in assembly language programs. The #define statements present in the include files describe the memory map and interrupt priorities. These include files also contain function prototypes for software subroutines generated with custom instruction implementation. Access to the peripheral registers is made possible by C structures present in the include directory. Since the Nios architecture and memory map are configurable, it is convenient to reference the hardware using the symbols defined in the include files without the need to change the code each time the Nios processor is configured. The excalibur.h file generated for the project is present in Appendix C. The prefixes na_ and np_ designate the Nios address and Nios peripheral, respectively.

The peripherals are accessed using the user-defined name assigned in SOPC builder as base address followed by the prefix. Thus the peripherals are referred to as na_<peripheral name> while the peripheral registers are accessed as np_<register name>.

**lib Directory**

The make file, and archive, source and object files for libraries usable by the Nios system are present in the lib directory. The assembled (or compiled) versions of routines from each file present in the archive assist in linking the software program. The make file contains instructions for rebuilding the archive file.
src Directory

All the software source code that is to be compiled and executed must be placed in the src directory. The Nios utilities nios-build and nios-run could then be used at the SDK shell command prompt to compile, link and execute these source codes.

GERMS monitor

The GERMS monitor is a simple monitor program used for development facilities included in the default software reference design present on the Nios development board. The GERMS monitor is the first code to execute on power-up and controls the boot process. Special purpose utility program (nios-run –t –r) could be used at the Nios SDK shell command prompt to establish a terminal connection with the GERMS monitor program. While executing, the monitor program is always waiting for commands. GERMS is a mnemonic that stands for the command set of the monitor program.

G  Go (run a program)
E  Erase flash
R  Relocate next download
M  Memory set and dump
S  Send S-records
:  Send I-Hex records

With the above command set, the monitor program provides a way to read from and write to the on-board SRAM or flash memories, to download and run executable code and to erase flash memory. This Nios system monitor program is always stored in the on-chip ROM.
4.3 The Gnupro Toolkit

The Gnupro Toolkit manufactured by RedHat Inc. is included in the Nios development kit. In addition to the SDK, the Gnupro tools are the foundation for developing Nios software. The toolkit includes basic facilities for software development such as an assembler, C/C++ compiler, linker and debugger (command-line GDB and GUI-based Insight debugger). The Cygwin bash shell, which provides a command line interface similar to UNIX is also a part of Gnupro toolkit. These facilities are used to compile and link software with the SDK routines, header files and other software libraries. While designing a system, rapid edit-compile-download-debug iterations are made possible with the available Gnupro Toolkit and Nios development utilities.

4.4 Third-Party Software Evaluation Tools

First Silicon Solutions provide evaluation software tools to enhance Nios development environment. These evaluation tools include Nios On-chip Instrumentation (OCI) Debug Module core, which provides a JTAG-based in-system analyzer including complex break points and memory and register visibility for debugging software. The Nios OCI Debug Console is available to provide a command-line interface for communicating with the Nios OCI Debug Module. These evaluation software tools are used for complex debugging purposes.
CHAPTER 5
CUSTOM IMPLEMENTATION OF DIF RADIX-2 FFT

5.1 Development Flow Considerations for Custom Implementation

Before implementing any custom software, the hardware and software development flow for compiling, debugging and prototyping embedded software for the Nios processor must be considered. Figure 5.1 illustrates both the hardware and software design flow required to create a working system.

In the Figure 5.1, the right side depicts the software development flow while the left side illustrates the hardware design flow. For this research, the factory configured Nios reference design is used as a platform to develop software for implementing DIF radix-2 FFT algorithm. To accelerate the performance of the algorithm, the butterfly computation in the algorithm is implemented in hardware using custom instruction. The Nios system is then recompiled with the integrated custom instruction block using Quartus II software, which is then downloaded to the development board. This is the only hardware step involved in the project. The software implementation of the algorithm involves the following steps.

i. The SDK for the target 32-bit Nios system is obtained and serves as the foundation for software development environment.

ii. The C code for the DIF radix-2 algorithm is then written using a text editor. The butterfly computation in the algorithm is implemented in C using custom instruction subroutines generated by the SOPC Builder. This C code is present in Appendix A. The source code is then compiled using Nios-build utility to convert it into an executable code.
Figure 5.1 Hardware/Software Development Flow for a Nios Processor System
iii. The executable code is then downloaded and executed with the aid of GERMS monitor program. The Nios-run utility is used for downloading and executing the code.

iv. The code is then debugged using `printf()` statements. Nios provides an option for special kind of printf statement known as small `printf()` that support only integer values. This `printf()` takes about 1kbyte of Nios code when compared to the standard `printf()` statement, which takes about 40kbytes of Nios code.

v. Once the debugging process is complete the code can be stored either in off-chip memory or on-chip memory. In the research, the code is stored in flash memory (off-chip) memory using the srec2flash utility so that it executes automatically at start-up.

### 5.2 Custom Instruction Features

Custom instruction is a process of implementing complex sequence of standard instructions in hardware in order to reduce them to a single instruction that could be accessed by software. Custom instructions are used to implement complex processing tasks in single-cycle (combinatorial) and multi-cycle (sequential) operations. Adding custom instructions to the Nios embedded processor instruction set accelerates time-critical software algorithms. In addition, these user-added custom instructions can access memory as well as logic outside the Nios system. This custom instruction feature present in Altera Nios embedded processor system can be used in variety of applications such as packet header processing, optimizing software inner loops for digital signal processing (DSP) and varied computation-intensive applications. The Nios CPU configuration wizard present in the SOPC builder software tool, provides a graphical user interface (GUI) that enables adding of up to five custom instruction to the Nios processor instruction set. For the project, a single custom instruction is used for implementing the inner butterfly loop present in DIF radix-2 FFT algorithm.
The two essential elements of custom instruction are the custom logic block and the software macro. Custom logic block is the hardware that performs the complex sequence of standard operations. Once implemented this block becomes part of Nios processor’s ALU. Figure 5.2 depicts how a custom logic is added to the Nios ALU. The number of custom logic blocks that could be implemented in Nios embedded processor system is restricted to five blocks but with the presence of 11-bit prefix port up to 2048 functions for each block can be performed. The software macro generated during custom logic implementation is used to access the custom logic through software code.

The custom logic block is implemented with the aid of Nios CPU configuration wizard. The Nios configuration wizard integrates the custom logic block with the Nios processor’s ALU and also creates the software macros in C/C++ and assembly. With the appropriate interface provided by the Nios processor, a custom logic block can be designed to perform any function.

Figure 5.2 Adding Custom Logic to the Nios ALU
Hardware Interface

The custom logic block can be created using any of the following formats:

- VHDL
- Verilog HDL
- EDIF netlist file
- Quartus II Block Design file (.bdf)
- Verilog Quartus Mapping File (.vqm)

Since the logic block connects directly to ALU as shown in Figure 5.2, it provides an interface with predefined ports and names. The predefined ports and names that could be used for custom logic for factory configured 32-bit Nios processor are shown in Figure 5.3.

Figure 5.3 Custom Logic Block Interface for 32-bit Nios Processor
The Nios configuration wizard first scans the custom logic block and searches the predefined ports used in the custom logic block and connects these ports to the ALU. The custom logic blocks can be implemented using either one or a combination of the following four options.

1. Combinatorial logic
2. Multi-cycle logic
3. Parameterization
4. User-defined ports

**Combinatorial Logic Option**

If only the combinatorial logic is used, the custom logic block is required to complete the operation in one clock cycle. Since only one clock cycle is used for the combinatorial logic, the custom logic requires only data ports and does not require any control signals. This logic block may require either one or both the data ports as inputs available for custom logic block. If only one data port is required, the dataa port is used. The ports available for combinatorial custom logic are shown in Table 5.1. When the ALU issues the combinatorial custom instruction, the data is read from dataa and datab ports at the rising edge of the CPU clock. The result port data is available on the following rising edge of the CPU clock.

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Width (Bits)</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataa</td>
<td>CPU width</td>
<td>Input</td>
<td>Operand</td>
</tr>
<tr>
<td>Datab</td>
<td>CPU width</td>
<td>Input</td>
<td>Operand (optional)</td>
</tr>
<tr>
<td>Result</td>
<td>CPU width</td>
<td>Output</td>
<td>Result</td>
</tr>
</tbody>
</table>
Multi-Cycle Logic Option

For multi-cycle logic option, both the data ports and the control signal ports must be provided in the custom logic block. The control signal ports are used to synchronize the custom logic block with the Nios processor. While using multi-cycle custom logic for an operation, the number of clock cycles required to perform the operation must be specified. Table 5.2 specifies the control signal ports and their description, required for multi-cycle logic.

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Width (Bits)</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk</td>
<td>1</td>
<td>Input</td>
<td>CPU master input clock, which is fed by the Nios system clock.</td>
</tr>
<tr>
<td>Reset</td>
<td>1</td>
<td>Input</td>
<td>CPU master asynchronous reset, which is fed by the Nios master reset.</td>
</tr>
<tr>
<td>clk_en</td>
<td>1</td>
<td>Input</td>
<td>Clock qualifier. The custom logic block should use the clk_en signal as a conventional clock qualifier signal and should ignore all rising clock edges when clk_en is not asserted.</td>
</tr>
<tr>
<td>Start</td>
<td>1</td>
<td>Input</td>
<td>Instructs the block to latch data and begin operation.</td>
</tr>
</tbody>
</table>

The start control signal is asserted on the first clock cycle the operation executes. The dataa and datab ports are valid at this time and are undefined for all subsequent clock cycles. The CPU waits for the number of clock cycles specified to complete the multi-cycle logic operation and then reads the result port. For example, Figure 5.4 illustrates the multi-cycle logic timing that requires five clock cycles. In the Figure, the start signal is asserted at T0 clock edge at which time valid input data is available. The custom instruction provides a valid result after
five CPU clock cycles. The reset signal is asserted only when the Nios system is reset. The logic block uses the clk_en signal as a conventional clock qualifier and ignores all rising edges of clock when the clk_en signal is disabled.

![Multi-Cycle Logic Timing Example (5 CPU Clock Cycles)](image_url)

**Figure 5.4** Multi-Cycle Logic Timing Example (5 CPU Clock Cycles)

**Parameterization Option**

The 11-bit prefix port can be used for passing parameters to the custom instruction. This port can be used either with combinatorial or multi-cycle custom logic blocks. Since the interpretation of prefix port is unspecified, it can be encoded to represent up to 2048 different functions contained in a single custom instruction or can be used as an input port to pass parameters. Before the custom instruction executes, the prefix port uses the Nios PFX instruction to load the 11-bit K register. The prefix port has valid data on the first clock cycle when the ALU issues the custom instruction. Table 5.3 illustrates the prefix port.

**Table 5.3 Prefix port**

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Width (Bits)</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix</td>
<td>11</td>
<td>Input</td>
<td>Loads the K register</td>
</tr>
</tbody>
</table>
User-Defined Port Option

Components outside the Nios system present in the Nios development board can be accessed with optional user defined ports. If the Nios CPU configuration wizard does not recognize a port, it considers it as a user defined port and routes it to the top level of the system module where external logic can access the signal.

By combining all the above-mentioned options, extremely versatile operations can be implemented as custom instructions. Since the custom instruction is integrated with the Nios CPU, the design of the custom instruction affects the maximum performance directly. Custom instruction must be designed in such a way that it does not become a performance bottleneck in a Nios system.

Software Interface

Once the custom logic block is integrated into the Nios processor’s ALU, the custom logic can be accessed through software through five user-definable opcodes shown in Table 5.4. Software macros are used to call these opcodes in C/C++.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>USR0</td>
<td>RR</td>
<td>Ra &lt;- Ra op Rb</td>
</tr>
<tr>
<td>USR1</td>
<td>Rw</td>
<td>Ra &lt;- Ra op %R0</td>
</tr>
<tr>
<td>USR2</td>
<td>Rw</td>
<td>Ra &lt;- Ra op %R0</td>
</tr>
<tr>
<td>USR3</td>
<td>Rw</td>
<td>Ra &lt;- Ra op %R0</td>
</tr>
<tr>
<td>USR4</td>
<td>Rw</td>
<td>Ra &lt;- Ra op %R0</td>
</tr>
</tbody>
</table>
The custom instruction takes the data contents from two general-purpose registers (Ra and Rb) and performs an operation as described by the custom logic block. The output of the operation is then stored in a general-purpose register Ra. The USR0 is an opcode of type RR, which can use any of the general-purpose registers for both Ra and Rb. The opcodes USR1 through USR2 are of type Rw, where only Ra can be any of the general-purpose register while Rb must always be %R0 register, which is one of the non-sliding Global registers present in the 32 general-purpose registers accessible to software. The register usage is transparent in C/C++ as the compiler automatically chooses the registers. If the code is written in assembly language, the registers used for a particular operation must be indicated. The Nios configuration wizard automatically builds the software macro when the custom instruction is added and also supports macro naming by the user. The Nios system header file (excalibur.h) generated by the SOPC builder includes the C/C++ macro definition. The custom instructions are accessed in C/C++ with the macro function call defined in the header file. Hence, the Nios system header file must be included in the C/C++ program.

5.3 Implementation of Butterfly Processor as Custom Instruction

The implementation of butterfly processor (Figure 2.4) in the DIF radix-2 FFT algorithm as custom instruction requires the following processes.

- Creation of custom logic block
- Instantiation of custom instruction
- Creation of software Code using Macros
Creation of Custom Logic Block

The custom logic block for the butterfly processor is written in VHDL. The logic design is implemented using multi-cycle logic option. The ports used for the design are illustrated in Table 5.5.

<table>
<thead>
<tr>
<th>Port</th>
<th>Width (Bits)</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataa</td>
<td>CPU width (32)</td>
<td>Input</td>
</tr>
<tr>
<td>Datab</td>
<td>CPU width (32)</td>
<td>Input</td>
</tr>
<tr>
<td>result</td>
<td>CPU width (32)</td>
<td>Output</td>
</tr>
<tr>
<td>Clk</td>
<td>1</td>
<td>Input</td>
</tr>
<tr>
<td>Reset</td>
<td>1</td>
<td>Input</td>
</tr>
<tr>
<td>clk_en</td>
<td>1</td>
<td>Input</td>
</tr>
<tr>
<td>Start</td>
<td>1</td>
<td>Input</td>
</tr>
<tr>
<td>prefix</td>
<td>11</td>
<td>Input</td>
</tr>
</tbody>
</table>

The prefix port is used in the design to define various read and write functions required for reading and writing various complex input and output data present in butterfly computation. The prefix values from 1 to 4 are used for read operations while the prefix values from 5 to 8 are used for writing the output values. The simulation result for the butterfly processor design shown in Figure 5.5 indicates that eight clock cycles are required for valid implementation of the butterfly processor as custom instruction.
Figure 5.5  Simulation Result of Butterfly Processor
The butterfly processor design is implemented with 16-bit input data. The twiddle factor coefficient data required for the design is quantized to 16-bit. The processor is implemented with one adder, one subtraction and a component instantiation for twiddle-factor multiplier. The twiddle factor multiplication is efficiently computed using component instantiations of three lpm_mult and three lpm_add_sub modules. The output of the twiddle factor multiplier is scaled such that it has the same data format as the input. The algorithm used for twiddle-factor multiplier uses three coefficients of the twiddle factor, C, C+S, and C-S where C and S are the real and imaginary coefficients of the twiddle factor. The complex twiddle factor multiplication \( R+j*I = (X+j*Y)(C+j*S) \) is efficiently performed when the above coefficients are used. The real and imaginary parts of complex twiddle factor multiplication using the above three coefficients can be computed as \( R = (C-S)*Y + C*(X-Y) \) and \( I = (C+S)*X – C*(X-Y) \) respectively. To ensure short latency for in-place FFT computation, the complex multiplier is implemented with no pipeline stages. The butterfly processor is designed to compute scaled outputs, where the output produced by the design is equal to half the actual output value. The butterfly processor design uses flip-flops for the input, coefficient and the output data to have single input/output registered design. The VHDL codes for both the butterfly processor (bfproc.vhd) and the twiddle factor multiplier (ccmul.vhd) are available in Appendix B. The design is compiled with EPF20k200EFC484-2X (APEX device) present in Nios development board using Quartus II software (version 2.2). It requires 1625/8320 logic cells of the apex device. The performance (fmax) of the design is found to be 30.82 MHz. Figure 5.5 shows the simulation result of the butterfly processor design, with input data 10+20i and 30+50i and with twiddle factor coefficient data \( W_8^0 \) quantized to 16 bits. The simulated output results shown are scaled by two (20, 35, -10, -15) as per the design when compared to the normal output of the butterfly (40, 70, -20, -30).
Instantiation of Custom Instruction

The following steps are performed to add the custom module created above.

i. The top-level block diagram file, standard_32.bdf, for the 32-bit Nios processor system is opened and by double-clicking the ref_32_system symbol present in the .bdf file the SOPC builder is launched. This is shown in Figure 5.6.

![Standard_32.bdf file](image-url)
ii. The Nios configuration wizard is then initiated by double clicking the CPU component present in the SOPC Builder. The Enable advanced configuration controls option is turned on in the architecture tab of Nios Configuration Wizard to add custom instruction. This is illustrated in Figure 5.7. In the hardware tab the Mul option is chosen as the multiplier option.

Figure 5.7  Nios Configuration Wizard Present in SOPC Builder
iii. The USR0 opcode in the custom instruction tab is selected and the import button is clicked to display the Interface to User logic window. The top-level design for the project (bfpr.vhd) is added using the Add button shown in Figure 5.8. By clicking on the Read Port-list from files tab, the wizard scans the file and displays the ports as shown in Figure 5.8.

**Figure 5.8 Interface to User Logic**
iv. Since the custom logic block designed in VHDL is of a different format when compared to that of the Nios system module, the Instantiate as Black Box option present under the instantiation tab is turned on.

v. The base name for the custom instruction macro appears as bfpr, the first four letters of the top-level design. The number of clock cycles required for the butterfly computation is entered as eight, based on the simulation output for the design, in the Cycle Count box shown in Figure 5.9. By clicking the Finish tab the custom instruction is added to the Nios CPU.
vi. Finally, clicking on the Generate tab in the SOPC builder generates the new custom-logic integrated Nios system module. The generation process also creates an excalibur.h header file that includes the software macro subroutines for the added custom instruction.

vii. The new Nios system module is then recompiled using Quartus II software. Figure 5.10 shows the compilation report giving the number of logic cells occupied by the new Nios system module with the butterfly processor custom module added to it. The maximum registered performance for the entire design is 30.82 MHz.
viii. The programming file (.sof) for the generated Nios system is then downloaded to the Nios development board by clicking the start button in the programmer window depicted in Figure 5.11. This provides a target platform for software development.

Figure 5.11 Programming Window
Creation Of software Code Using Macros

The SOPC builder generates the software macro for the butterfly processor as nm_bfpr_pfx (prefix, dataa, datab) in the excalibur.h header file. The prefix nm stands for Nios macro. The custom logic for the processor can be accessed in C/C++ through this macro function call. Before using the macro function, the excalibur.h file must be included in the software code. The excalibur.h file generated with macro subroutines for butterfly processor is shown in Appendix C. The Nios Gnupro compiler and assembler output for two of the instructions in butterfly computation is illustrated in Appendix D.
CHAPTER 6
DISCUSSION OF RESULTS

The implemented custom FFT code could be used to calculate any radix-2 length FFT. Since the Nios embedded processor has high memory capacity that includes the on-chip RAM and external SRAM chips, larger length fast Fourier transforms can be calculated. The customized FFT code could also be used to calculate inverse DFT by making a simple modification in the twiddle factor input calculation. The results of custom instruction code are compared with software only code based on the following two features.

1. Performance
2. Quantization error

6.1 Performance

The C program (timecheck.c) present in Appendix E compares the number of clock cycles required for custom implementation and software-only implementation of a single butterfly computation. This C program uses predefined timer subroutines generated in the excalibur.h file by the SOPC builder. The speed increase due to custom implementation for a single butterfly is measured with the three multiplier optimizations available with the Nios processor. It can be seen from the results shown in Figure 6.1 that the number of clock cycles required for the software-only implementation using Mul optimization is about 176 clock cycles more than that required for custom implementation.

The increase in speed due to custom implementation of the butterfly processor is given by

\[
\text{Speed Increase} = \frac{\text{Number of clock cycles required for software-only implementation}}{\text{Number of clock cycles required for Custom implementation}}
\]
Table 6.1  Speed Increase for Single Butterfly

<table>
<thead>
<tr>
<th>Multiplier Option</th>
<th>Clock Cycles with Software</th>
<th>Clock Cycles with Custom Instruction</th>
<th>Speed Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>1227</td>
<td>119</td>
<td>10.3</td>
</tr>
<tr>
<td>MSTEP</td>
<td>698</td>
<td>119</td>
<td>5.8</td>
</tr>
<tr>
<td>MUL</td>
<td>295</td>
<td>119</td>
<td>2.47</td>
</tr>
</tbody>
</table>

Table 6.1 shows the speed increase with custom implementation for different multiplier optimizations. From the Table, it becomes evident that use of custom instructions yields higher performance when compared to software-only implementation.

Figure 6.1  Output Results of timecheck.c
The C codes in Appendix A compare the number of clock cycles required for custom implementation and software-only implementation of the entire FFT. The number of clock cycles required for the computation of different length FFTs using both custom instruction and software only implementation for Mul optimization is summarized in Table 6.2. From the Table, it is evident that speed increase is higher for larger length FFT computations.

Table 6.2   Clock Cycles for Different Length FFTs

<table>
<thead>
<tr>
<th>Length of FFT</th>
<th>Number of Clock Cycles for Custom Implementation</th>
<th>Number of Clock Cycles for Software only Implementation</th>
<th>Speed Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>51046</td>
<td>52762</td>
<td>1.03</td>
</tr>
<tr>
<td>32</td>
<td>62463</td>
<td>72637</td>
<td>1.16</td>
</tr>
<tr>
<td>128</td>
<td>129769</td>
<td>185264</td>
<td>1.42</td>
</tr>
<tr>
<td>256</td>
<td>228036</td>
<td>353961</td>
<td>1.55</td>
</tr>
<tr>
<td>512</td>
<td>448064</td>
<td>729976</td>
<td>1.62</td>
</tr>
<tr>
<td>1024</td>
<td>927429</td>
<td>1551117</td>
<td>1.67</td>
</tr>
<tr>
<td>2048</td>
<td>1965985</td>
<td>3333629</td>
<td>1.69</td>
</tr>
</tbody>
</table>

Figure 6.2, 6.3 and 6.4 compares the performance of different length FFTs computed using custom implementation and software-only implementation with different multiplier optimizations. The performance of both the implementations for different length FFTs is calculated as

\[
\text{Performance} = \frac{\text{Nios CPU clock rate (33.33MHz)}}{\text{Number of CPU clock cycles required for a given length FFT computation}}
\]
It is observed that the overall performance decreases with increase in length of FFT for each implementation while the performance of custom implementation for each of the FFTs computed is higher than software-only implementation. Of all the multiplier optimizations the Mul optimization gives better overall performance for both the implementations.

Figure 6.2 Performance Comparison with Mul Optimization
Figure 6.3 Performance Comparison with Mstep Optimization

Figure 6.4 Performance Comparison with Software Emulation
6.2 Quantization Error

Table 6.3 shows the comparison of customized DIF radix-2 FFT code results with that of the test bench results for length-8 FFT. The test bench results are obtained from MatLab by using the function fft(). The quantization error present in the custom instruction code output is evident from the tabulated values. This error occurs due to the quantization of input and coefficient data for custom implementation.

<table>
<thead>
<tr>
<th>Input Test Data</th>
<th>Custom Instruction Code Results using Nios Board</th>
<th>Test Bench Results using MatLab</th>
</tr>
</thead>
<tbody>
<tr>
<td>20, 40, 60, 80, 100, 120, 140, 160</td>
<td>720+0i, -81+192i, -80+80i, -80+33i, -80+0i, -79-32i, -80-80i, -80-193i</td>
<td>720+0i, -80+193.14i, -80+80i, -80+33.14i, -80+0i, -80-33.14i, -80-80i, -80-193.14i</td>
</tr>
<tr>
<td>20i, 40i, 60i, 80i, 100i, 120i, 140i, 160i</td>
<td>0+720i, -194-81i, -80-80i, -33-80i, 0-80i, 34-79i, 80-80i, 193-80i</td>
<td>0+720i, -193.14-80i, -80-80i, 33.14-80i, 0-80i, 33.14-80i, 80-80i, 193.14-80i</td>
</tr>
<tr>
<td>20+20i, 40+40i, 60+60i, 80+80i, 100+100i, 120+120i, 140+140i, 160+160i</td>
<td>720+720i, -274+113i, -160+0i, -113-46i, -80-80i, -46-113i, 0-160i, 113-274i</td>
<td>720+720i, -273.14+113.14, -160+0i, -113.14-46.86i, -80-80i, -46.86-113.14i, 0-160i, 113.14-273.14i</td>
</tr>
</tbody>
</table>

Table 6.4 shows the average quantization errors of length-8, length-16 and length-32 DIF radix-2 FFT implemented with custom instruction for the butterfly processor. The input given to
these different length FFTs is identical with larger length input values being padded with zeroes. It is observed from the tabulated results that the average quantization error increases with increase in length of the FFT. This is a result of inherent quantizing effect present in custom implementation of the butterfly processor.

Table 6.4  Average Quantization Error of Different Length FFTs

<table>
<thead>
<tr>
<th>Length Of FFT</th>
<th>Input Values</th>
<th>Average quantization error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length-8</td>
<td>20+20i, 40+40i, 60+60i, 80+80i, 100+100i, 120+120i, 140+140i, 160+160i</td>
<td>0.86</td>
</tr>
<tr>
<td>Length-16</td>
<td>20+20i, 40+40i, 60+60i, 80+80i, 100+100i, 120+120i, 140+140i, 160+160i, 0, 0, 0, 0, 0, 0, 0</td>
<td>2.99</td>
</tr>
<tr>
<td>Length-32</td>
<td>20+20i, 40+40i, 60+60i, 80+80i, 100+100i, 120+120i, 140+140i, 160+160i, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0</td>
<td>3.60</td>
</tr>
</tbody>
</table>

From the above results it is observed that, by using custom instruction for implementing the butterfly processor in DIF radix-2 FFT increases the overall performance of the algorithm. Additionally, the software code required for implementing the algorithm using custom instruction is comparatively smaller than software-only implementation. But the only disadvantage of the custom instruction implementation is the quantization error present in the output values.
CHAPTER 7
CONCLUSIONS AND FUTURE WORK

7.1 Conclusion

With its configurable hardware and software features, the Nios processor provides a powerful, reliable platform for developing and implementing complex algorithms. In addition features such as low cost, smaller FPGA footprint, higher performance and robust software development tools make Nios an attractive choice for software development. In the thesis, the custom instruction feature of Nios processor is used to implement an FFT algorithm.

For the thesis, the DIF radix-2 FFT algorithm is chosen for custom implementation. The radix-2 FFT algorithm has many characteristics that are favorable for custom implementation. These features include higher possible transform length, smaller run time code length, bit-reverse index mapping and small butterfly processor for easier implementation in hardware.

For the custom implementation of the algorithm, a VHDL code was initially written for the butterfly processor. The software macro generated for this code was then used in the C code that implements the entire FFT algorithm. The DIF radix-2 FFT was, therefore, successfully implemented using custom instruction feature of the Nios development board.

By implementing the butterfly processor as custom instruction, it is observed that the performance of DIF radix-2 FFT algorithm is increased dramatically. Additionally, the software code required for implementing the butterfly processor as custom instruction was comparatively smaller than that required for software-only implementation. The only disadvantage of the custom implementation is the quantization error that is present in the output values. This quantization error increases with increase in length of the FFT.
In conclusion, the customization of DIF radix-2 FFT, by using the powerful custom instruction feature of Nios processors increases the performance dramatically, while reducing the size and complexity of software. In addition, quantization error that results due to custom implementation is small for smaller length FFTs and hence could be ignored.

7.2 Future work

The customized DIF radix-2 FFT design could be migrated to other high-performance low-cost Altera FPGA devices. Since the forward and inverse DFT transforms are similar, as shown below.

\[
x = Wx \quad \rightarrow \quad x = \frac{1}{N} W^*X
\]

The custom FFT design could then be used to calculate inverse DFT by taking the conjugate of the inverse DFT and by scaling the DFT of \(X^*\) by \(1/N\) as follows.

\[
x^* = \frac{1}{N} (W^*X)^* = WX^*
\]

Higher performance for the design could be achieved by implementing the entire FFT algorithm as custom instruction using DSP IP blocks provided by Altera. The DIT radix-2 FFT algorithm can also be implemented with similar implementation techniques as used in the custom implementation of DIF radix-2 FFT. Chirp-z FFT algorithm could also be implemented in Nios processors using custom instructions. This algorithm requires a length \(N\) convolution and \(2N\) multiplications. The convolution operation in the algorithm can be implemented using a Finite Impulse Response (FIR) filter as custom instruction. The performance of many other complex DSP algorithms can also be improved by transferring the software cycle-intensive operations to hardware using custom instruction feature of Nios processors.
APPENDIX A

C PROGRAM FOR DIF RADIX-2 FFT

 /**************************************************************************/
 /**************************************************************************/
 /**************************************************************************/
 /** Programmer: Divya Lakshmi Sunkara   **/
 /** FLORIDA STATE UNIVERSITY        **/
 /** MS in Electrical Engineering    **/
 /**************************************************************************/

 #include <stdio.h>
 #include <math.h>
 #include <stdlib.h>
 #include "excalibur.h"

 /**************************************************************************/
 /* Included all this extra declaration for timer to measure the performance */
 /* of FFT calculation */
 /**************************************************************************/

 #define TIMER_LOAD_VAL 0xFFFFFFFF
 #define take_float_as_int32(x) (*((int *)(&(x))))
 #define take_int32_as_float(i) (*((float *)(&(i))))

 typedef unsigned long DWORD;

 typedef struct
 {
   long interruptCount;               // Increment With Each Interrupt
   np_timer *timer;
 } TimerISRContext;

 static TimerISRContext gC = {0,0};

 int interrupt_count;                // global variable

 void MyTimerISR(int context)
 {
   TimerISRContext *c;
   c = (TimerISRContext *)context;
   c->interruptCount++;

   printf("\n(timer %d!)",c->interruptCount);
   interrupt_count = c->interruptCount;

   c->timer->np_timerstatus = 0;        // Write Anything to Clear the IRQ
 }
long GetTickCount()
{
    volatile long timerVal;
    volatile long timerPeriod;

    timer->np_timersnapl = 0;                  // Snapshot
    timerVal = (timer->np_timersnapl & 0x0ffff)
               + ((long)timer->np_timersnaph << 16);

    return timerVal;
}

void InitTimer()
{
    long timerPeriod = TIMER_LOAD_VAL;

    //Initialize timer
    timer->np_timerperiodh = timerPeriod >> 16;
    timer->np_timerperiodl = timerPeriod & 0xffff;

    //Set timer to continuous
    timer->np_timercontrol = timer->np_timercontrol
        | np_timercontrol_cont_mask;

    //Enable timer interrupt
    gC.timer = na_timer1;
    nr_installuserisr(na_timer1_irq,MyTimerISR,(long)&gC);
    gC.timer->np_timercontrol = gC.timer->np_timercontrol
                                | np_timercontrol_ito_mask;
    printf("\n\nTimer interrupt enabled.\n\n");

    //Start the timer
    timer->np_timercontrol = (timer->np_timercontrol & 3)
                               + np_timercontrol_start_mask;
}

void CheckTimeStamp (DWORD dwStartTick,DWORD lTicksUsed,DWORD timer_overhead)
{
    volatile DWORD duration;
    volatile long int_cnt;

    int_cnt = (interrupt_count-1) * TIMER_LOAD_VAL;
    duration=0;

    if (interrupt_count >=2){
        duration = TIMER_LOAD_VAL - lTicksUsed;
        duration += dwStartTick;
        duration += int_cnt;
    }
    else
    {
        duration = dwStartTick - lTicksUsed;
    }
    
}
printf("Number of clock cycles (minus overhead): %lu",
        duration - timer_overhead);
printf("n
**************************************************\n\n");

/*
 * Main Program
 */
main()
{
    int a, d, e, f, i, j, k, l, m, n, p, t, I, J, M, N, R, S;
    int sw, temp;
    int Are, Aim, Bre, Bim;
    long int Wr, W1, W2, Wi;
    long int *xr, *xi, *wnr, *wni;
    float b;
    float PI = 3.14159265;        // PI value
    volatile DWORD timer_overhead;
    volatile DWORD dwStartTick;
    volatile DWORD lTicksUsed;
    volatile DWORD our_dwStartTick;
    volatile DWORD our_lTicksUsed;

    // Initialize the timer
    InitTimer();

    // timer_overhead, assume no interrupts
dwStartTick = GetTickCount();    /* record start time*/
lTicksUsed = GetTickCount();     /* record end time */

    timer_overhead = dwStartTick - lTicksUsed;

    sw = 1;
    while (sw != 2)
    {
        printf("Enter Input Length\n");
        scanf("%d", &M);
        N = 2;
        for (n = 0; M > N; n++)             //Calculate Radix-2 Length
        {
            N = pow(2, n+1);
        }
printf("\n\nRadix-2 Length is %d",N);

/*
 * Dynamic Memory Allocation
 */
xr = (long int *) calloc (sizeof(long int), N);
xi = (long int *) calloc (sizeof(long int), N);

if (xr == (long int *)NULL)
    printf("\n xr - Memory Allocation Failure");
if (xi == (long int *)NULL)
    printf("\n xi - Memory Allocation Failure");

R = N/2;                          // Length of Twiddle Factor

wnr = (long int *) calloc (sizeof(long int), R);
wni = (long int *) calloc (sizeof(long int), R);

if (wnr == (long int *)NULL)
    printf("\n wnr - Memory Allocation Failure");
if (wni == (long int *)NULL)
    printf("\n wni - Memory Allocation Failure");

/*
 * End of Dynamic Allocation
 */

b = (2 * PI)/N;              //Calculation Of wnr and wni values

for (n=0; n<R; n++)
{
    *(wnr+n) = cos(b*n) * pow(2,15);     //Quantization of wnr
    *(wni+n) = -sin(b*n) * pow(2,15);   //Quantization of wni
}

printf("\n\nEnter the Real Input Values\n");
for (n=0; n<M; n++)
{
    scanf("%ld", (xr + n));   //Read the Real Input Values
}
for (n=M; n<N; n++)
{
    *(xr + n) = 0;
}

printf("\n\nEnter the Imaginary Input Values\n");
for (n=0; n<M; n++)
{
    scanf("%ld", (xi + n));   //Read the Imaginary Input Values
}

for (n=M; n<N; n++)
{
    *(xi + n) = 0;
}

dwStartTick=GetTickCount();    // Record Start Time
S = log10(N)/log10(2);         // Number of Stages
for (k=0; k<S; k++)     // Stage Loop
{
    p = 1<<k;     // Number of Groups
    t = N/(p << 1);    // Number of Butterflies in each Group
    f = t << 1;
    for (j=0; j<p; j++)   // Group Loop
    {
        I = f * j;         // Jump to Each Group in a Stage
        for (i=0; i<t; i++)  // Butterfly Loop
        {
            // Butterfly Calculation
            a = I + t;

            Are = (*(xr+I) << 1);
            Aim = (*(xi+I) << 1);

            ***Custom Instructions***
            nm_bfpr_pfx(1,Are,Aim);   // Read Are and Aim

            Bre = (*(xr+a) << 1);
            Bim = (*(xi+a) << 1);

            nm_bfpr_pfx(2,Bre,Bim);   // Read Bre and Bim

            Wr = *(wnr+(p*i));
            Wi = *(wni+(p*i));

            nm_bfpr_pfx(3,Wr,0);     // Read C

            W1 = Wr + Wi;
            W2 = Wr - Wi;

            nm_bfpr_pfx(4,W1,W2);    // Read C+S and C-S
// Write the real value
*(xr+I) = nm_bfpr_pfx(5,0,0);

// Write the imaginary value
*(xi+I) = nm_bfpr_pfx(6,0,0);

// Write the real value
*(xr+a) = nm_bfpr_pfx(7,0,0);

// Write the imaginary value
*(xi+a) = nm_bfpr_pfx(8,0,0);

I = I + 1;  //Jump to Each Butterfly in a Group

lTicksUsed=GetTickCount();           //Record End Time

CheckTimeStamp (dwStartTick, lTicksUsed, timer_overhead);

d = N / 2;
for (n=1; n < (N-1); n++)           //Final Bit Reversal Process
{
    if (n < d)
    {
        temp = *(xr + d);
        *(xr + d) = *(xr + n);
        *(xr + n) = temp;

        temp = *(xi + d);
        *(xi + d) = *(xi + n);
        *(xi + n) = temp;
    }

    e = N / 2;

    while (e <= d)
    {
        d = d - e;
        e = e / 2;
    }

    d = d + e;
}
/** Display Radix-2 FFT Output **/

printf("\n\n***************\n");
printf("RADIX-2 FFT OUTPUT ");
printf("\n***************\n");

for (n=0; n<N; n++)
{
    printf("\n(%ld)+i(%ld)", *(xr + n), *(xi + n));
}
printf("\n\n");

//Free Up the Dynamic Allocation Space

free (xr);
free (xi);
free (wnr);
free (wni);

/*
 *  Asking for Do you wish to continue ?
 */

printf("\n\nDo you wish to continue? (1-Yes / 2-No)\n");
scanf("%d", &sw);

}

printf("\n\n======= END =======\n");
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "excalibur.h"

/* Included all this extra declaration for timer to calculate performance */
/* of FFT calculations */

#define TIMER_LOAD_VAL 0xFFFFFFFF
#define take_float_as_int32(x) (*((int *)(&(x))))
#define take_int32_as_float(i) (*((float *)(&(i))))

np_timer *timer = na_timer1;
typedef unsigned long DWORD;

typedef struct
{
    long interruptCount;       // Increment With Each Interrupt
    np_timer *timer;
} TimerISRContext;

static TimerISRContext gC = {0,0};

//global variable
int interrupt_count;

void MyTimerISR(int context)
{
    TimerISRContext *c;
    c = (TimerISRContext *)context;
    c->interruptCount++;
    printf("\n(timer #\%d!)",c->interruptCount);
    interrupt_count = c->interruptCount;
    c->timer->np_timerstatus = 0; // Write Anything To Clear the IRQ
}

long GetTickCount()
{
    volatile long timerVal;
    volatile long timerPeriod;
timer->np_timersnapl = 0;                      // Snapshot
timerVal = (timer->np_timersnapl & 0x0ffff) + ((long)timer->np_timersnaph << 16);
return timerVal;
}

void InitTimer()
{
    long timerPeriod = TIMER_LOAD_VAL;

    //Initialize timer
timer->np_timerperiodh = timerPeriod >> 16;
timer->np_timerperiodl = timerPeriod & 0xffff;

    //Set timer to continuous
timer->np_timercontrol = timer->np_timercontrol | np_timercontrol_cont_mask;
    //Enable timer interrupt
gC.timer = na_timer1;
nr_installuserisr(na_timer1_irq,MyTimerISR,(long)&gC);
gC.timer->np_timercontrol = gC.timer->np_timercontrol | np_timercontrol_ito_mask;
    printf("\n\nTimer interrupt enabled.\n\n");

    //Start the timer
timer->np_timercontrol = (timer->np_timercontrol & 3) + np_timercontrol_start_mask;
}

void CheckTimeStamp (DWORD dwStartTick,DWORD lTicksUsed,DWORD timer_overhead)
{
    volatile DWORD duration;
    volatile long int_cnt;

    int_cnt = (interrupt_count-1) * TIMER_LOAD_VAL;
duration=0;

    if (interrupt_count >=2){
        duration = TIMER_LOAD_VAL - lTicksUsed;
duration += dwStartTick;
duration += int_cnt;
    }
else
{
    duration = dwStartTick - lTicksUsed;
}
printf("\nNumber of clock cycles (minus overhead): %lu",
        duration - timer_overhead);
printf("\n\n\n\n************************************

*/

/*
 * Main Program
 */

main()
{
    int a, d, e, f, i, j, k, l, m, n, p, t, I, J, M, N, R, S;
    float b;
    int temp;
    long int *xr, *xi;
    long int wr, wi, *wnr, *wni;
    float PI = 3.14159265; // PI Value

    volatile DWORD timer_overhead;
    volatile DWORD dwStartTick;
    volatile DWORD lTicksUsed;
    volatile DWORD our_dwStartTick;
    volatile DWORD our_lTicksUsed;

    // Initialize the timer
    InitTimer();

    // timer_overhead, assume no interrupts
    dwStartTick=GetTickCount(); // record start time
    lTicksUsed=GetTickCount();  // record end time

    timer_overhead = dwStartTick - lTicksUsed;

    printf("\n\nEnter Input Length\n");
    scanf("%d",&M);

    N = 2;

    for (n=0; M > N; n++)  //Calculate Radix-2 Length
    {
        N = pow(2,n+1);
    }

    printf("\n\nRadix-2 Length is %d",N);
/* Define the input Length */
xr = (long int *) calloc (sizeof(long int), N);
xi = (long int *) calloc (sizeof(long int), N);

if (xr == (long int *)NULL)
    printf("\n xr - Memory Allocation Failure");
if (xi == (long int *)NULL)
    printf("\n xi - Memory Allocation Failure");

/*
* Calculate Twiddle Factor Length
*/
R = N/2;                               // Length of Twiddle Factor
wnr = (long int *) calloc (sizeof(long int), R);
if (wnr == (long int *)NULL)
    printf("\n wnr - Memory Allocation Failure");
wni = (long int *) calloc (sizeof(long int), R);
if (wni == (long int *)NULL)
    printf("\n wni - Memory Allocation Failure");

b = (2 * PI)/N;
for (n=0; n<R; n++)               // Calculation of wnr and wni values
    { *(wnr+n) = cos(b*n) * pow(2,15);           //Quantization of wnr
      *(wni+n) = sin(b*n) * pow(2,15);   //Quantization of wni
    }

printf("\n\nEnter the Real Input Values\n");
for (n=0; n<M; n++)                 // Read the Real Input Values
    { scanf("%ld", (xr + n));
    }
for (n=M; n<N; n++)
    { *(xr + n) = 0;
    }

printf("\n\nEnter the Imaginary Input Values\n");
for (n=0; n<M; n++)                 // Read the Imaginary Input Values
    { scanf("%ld", (xi + n));
    }
for (n=M; n<N; n++)
{
    *(xi + n) = 0;
}

/*
 * Beginning of FFT Algorithm
 */

dwStartTick=GetTickCount();       // Record Start Time

S = log10(N)/log10(2);            // Number of Stages

for (k=0; k<S; k++)    // Stage Loop
{
    p = 1 << k;       // Number of Groups

    t = N/(p << 1);   // Number of Butterflies in each Group
    f = t << 1;

    for (j=0; j<p; j++) // Group Loop
    {
        I = f * j;    // Jump to Each Group in a Stage

        for (i=0; i<t; i++) // Butterfly Loop
        {
            a = I + t;

            // Butterfly Calculation
            temp = *(xr+I);
            *(xr+I) = *(xr+I) + *(xr+a);
            *(xr+a) = temp - *(xr+a);

            temp = *(xi+I);
            *(xi+I) = *(xi+I) + *(xi+a);
            *(xi+a) = temp - *(xi+a);

            wr = *(wnr+(p*i));
            wi = *(wni+(p*i));

            temp = *(xr+a);
            *(xr+a) = (*(xr+a)) * wr + (*(xi+a)) * wi;
            *(xr+a) = (*(xr+a)) / (1 << 15);

            *(xi+a) = (*(xi+a)) * wr - (temp * wi);
            *(xi+a) = (*(xi+a)) / (1 << 15);

            I = I + 1;     // Jump to Each Butterfly in a Group
        }
    }
lTicksUsed=GetTickCount();       // Record End Time

CheckTimeStamp (dwStartTick, lTicksUsed, timer_overhead);

d = N / 2;

for (n=1; n < (N-1); n++)          // Final Bit Reversal Process
{
   if (n < d)
   {
      temp = *(xr + d);
      *(xr + d) = *(xr + n);
      *(xr + n) = temp;

      temp = *(xi + d);
      *(xi + d) = *(xi + n);
      *(xi + n) = temp;
   }

   e = N / 2;

   while (e <= d)
   {
      d = d - e;
      e = e / 2;
   }

   d = d + e;
}

/** Display Radix-2 FFT Output **/

printf("\n\n***************************\n");
printf("RADIX-2 FFT OUTPUT   ");
printf("\n***************************\n");

for (n=0; n<N; n++)
{
   printf("\n(%ld)+i(%ld)", *(xr + n),*(xi + n));
}

printf("\n\n");

//Free Up the Dynamic Allocation Space

free (xr);
free (xi);
free (wnr);
free (wni);

} /* END OF THE PROGRAM */
APPENDIX B

VHDL PROGRAMS FOR BUTTERFLY COMPUTATION

Butterfly Processor (bfproc.vhd)

LIBRARY lpm;
USE lpm.lpm_components.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

PACKAGE mul_package IS
    COMPONENT ccmul
        GENERIC (W2 : INTEGER := 34;    -- Multiplier bit width
                  W1 : INTEGER := 17;    -- Bit width c+s sum
                  W  : INTEGER := 16);   -- Input bit width
        PORT
            (clk   : IN STD_LOGIC; -- Clock for the output register
             x_in, y_in      : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);   -- Inputs
             cps_in, cms_in, c_in  : IN  STD_LOGIC_VECTOR(W1-1 DOWNTO 0);  -- Inputs
             r_out, i_out    : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0));  -- Results
        END COMPONENT;
    END mul_package;

LIBRARY work;
USE work.mul_package.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY lpm;
USE lpm.lpm_components.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY bfproc IS
    GENERIC (W2 : INTEGER := 34;     -- Multiplier bit width
              W1 : INTEGER := 17;     -- Bit width c+s sum
              W  : INTEGER := 16);    -- Input bit width
    PORT(     -- Template for Custom Instruction
        SIGNAL clk : IN STD_LOGIC;
ARCHITECTURE flex OF bfproc IS
  SIGNAL dif_re, dif_im                          -- Bf out
    : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
  SIGNAL Are, Aim, Bre, Bim : integer RANGE -32768 TO 32767;
    -- Inputs as integers
  SIGNAL c           : STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
    -- Coefficient Input
  SIGNAL cps, cms    : STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
    -- Coefficient input
  SIGNAL Cre, Cim    : STD_LOGIC_VECTOR(W-1 DOWNTO 0);

  SIGNAL Ere_out, Eim_out, Ere_out1, Eim_out1 :
    STD_LOGIC_VECTOR(W-1 DOWNTO 0);              -- Results
  SIGNAL tempr, tempi   : STD_LOGIC_VECTOR(32-1 DOWNTO 0);

BEGIN
  PROCESS   -- Compute the additions of the butterfly using
  BEGIN      -- integers and store inputs in flip-flops
    WAIT UNTIL clk = '1';
    IF reset = '0' THEN
      IF clk_en = '1' THEN
        IF prefix = "00000000001" THEN
          IF start = '1' THEN
            Are <= CONV_INTEGER(dataa(W-1 DOWNTO 0)); -- Read Are
            Aim <= CONV_INTEGER(datab(W-1 DOWNTO 0)); -- Read Aim
          END IF;
        END IF;
      END IF;
      IF prefix = "00000000010" THEN
        IF start = '1' THEN
          Bre <= CONV_INTEGER(dataa(W-1 DOWNTO 0)); -- Read Bre
          Bim <= CONV_INTEGER(datab(W-1 DOWNTO 0)); -- Read Bim
        END IF;
      END IF;
      IF prefix = "00000000011" THEN
        IF start = '1' THEN
          c  <=  dataa(W1-1 DOWNTO 0);     -- Load from memory cos
        END IF;
      END IF;
      IF prefix = "00000000100" THEN
        IF start = '1' THEN
          cps <= dataa(W1-1 DOWNTO 0); --Load from memory cos+sin
          cms <= datab(W1-1 DOWNTO 0); --Load from memory cos-sin
        END IF;
      END IF;
    END IF;
  END PROCESS;
END bfproc;
END IF;
END PROCESS;

-- No FF because butterfly difference "diff" is not an output port
PROCESS (Are, Bre, Aim, Bim)
BEGIN
   dif_re <= CONV_STD_LOGIC_VECTOR(Are/2 - Bre/2, W);
   dif_im <= CONV_STD_LOGIC_VECTOR(Aim/2 - Bim/2, W);
   temper <= CONV_STD_LOGIC_VECTOR( (Are + Bre )/2, 32);
   tempi <= CONV_STD_LOGIC_VECTOR( (Aim + Bim )/2, 32);
END PROCESS;

---- Instantiate the complex twiddle factor multiplier ----
ccolm_1: ccolm
GENERIC MAP ( W2 => W2, W1 => W1, W => W)
PORT MAP  ( clk => clk, x_in => dif_re, y_in => dif_im,
           c_in => c, cps_in => cps, cms_in => cms,
           r_out => Ere_out, i_out => Eim_out);
PROCESS (Ere_out, Eim_out)
BEGIN
   Ere_out1(W-1 DOWNTO 0) <= Ere_out;
   Eim_out1(W-1 DOWNTO 0) <= Eim_out;
END PROCESS;

PROCESS
BEGIN
   WAIT UNTIL clk = '1';
   IF reset = '0' THEN
      IF clk_en = '1' THEN
         IF prefix = "00000000101" THEN
            RESULT(31 DOWNTO 0) <= tempn(31 DOWNTO 0); -- Real part
         END IF;
      END IF;
      IF prefix = "00000000110" THEN
         RESULT(31 DOWNTO 0) <= tempi(31 DOWNTO 0);  -- Imaginary part
      END IF;
      IF prefix = "00000000111" THEN                  -- Write real part
         IF Ere_out1(Ere_out1'high) = '1' THEN
            RESULT(31 DOWNTO 0) <= "11111111111111" & Ere_out1;
         ELSE
            RESULT(31 DOWNTO 0) <= "0000000000000000" & Ere_out1;
         END IF;
      END IF;
      IF prefix = "00000001000" THEN              -- Write imaginary part
         IF Eim_out1(Eim_out1'high) = '1' THEN
            RESULT(31 DOWNTO 0) <= "11111111111111" & Eim_out1;
         ELSE
            RESULT(31 DOWNTO 0) <= "0000000000000000" & Eim_out1;
         END IF;
      END IF;
   END IF;
END PROCESS;
END flex;
ENTITY ccmul IS
  GENERIC (W2  : INTEGER := 34;  -- Multiplier bit width
            W1  : INTEGER := 17;  -- Bit width c+s sum
            W   : INTEGER := 16);  -- Input bit width
  PORT (clk  : STD_LOGIC;  -- Clock for the output register
         x_in, y_in                        -- Inputs
            : IN  STD_LOGIC_VECTOR(W-1 DOWNTO 0);
         cps_in, cms_in, c_in              -- Inputs
            : IN  STD_LOGIC_VECTOR(W1-1 DOWNTO 0);
         r_out, i_out                      -- Results
            : OUT STD_LOGIC_VECTOR(W-1 DOWNTO 0));
END ccmul;

ARCHITECTURE flex OF ccmul IS

  SIGNAL x, y   : STD_LOGIC_VECTOR(W-1 DOWNTO 0);  -- Inputs and outputs
  SIGNAL r, i, cmsy, cpsx, xmyc           -- Products
            : STD_LOGIC_VECTOR(W2-1 DOWNTO 0);
  SIGNAL xmy, cps, cms, sxtx, shty,c   -- x-y etc.
            : STD_LOGIC_VECTOR(W1-1 DOWNTO 0);

BEGIN
  x   <= x_in;   -- x
  y   <= y_in;   -- j * y
  c   <= c_in;   -- c
  cps <= cps_in; -- cos + sin
  cms <= cms_in; -- cos - sin

  PROCESS
    BEGIN
      WAIT UNTIL clk='1';
      r_out <= r(W2-4 DOWNTO W-1); -- Scaling and FF
      i_out <= i(W2-4 DOWNTO W-1); -- for output
    END PROCESS;

---------- ccmul with 3 mul. and 3 add/sub -----------
  sxtx  <= x(x'high) & x;         -- Possible growth for
  shty  <= y(y'high) & y;         -- sub_1 -> sign extension

  sub_1: lpm_add_sub                -- Sub: x - y;
    GENERIC MAP ( LPM_WIDTH => W1, LPM_DIRECTION => "SUB",
                   LPM_REPRESENTATION => "SIGNED")
PORT MAP (dataa => sxtx, datab => sxty, result => xmy);

mul_1: lpm_mult
  -- Multiply \((x-y)\times c = xmyc\)
  GENERIC MAP (LPM_WIDTHA => W1, LPM_WIDTHB => W1,
                LPM_WIDTHP => W2, LPM_WIDTHS => W2,
                LPM_REPRESENTATION => "SIGNED")
  PORT MAP (dataa => xmy, datab => c, result => xmyc);

mul_2: lpm_mult
  -- Multiply \((c-s)y = cmsy\)
  GENERIC MAP (LPM_WIDTHA => W1, LPM_WIDTHB => W,
                LPM_WIDTHP => W2, LPM_WIDTHS => W2,
                LPM_REPRESENTATION => "SIGNED")
  PORT MAP (dataa => cms, datab => y, result => cmsy);

mul_3: lpm_mult
  -- Multiply \((c+s)x = cpsx\)
  GENERIC MAP (LPM_WIDTHA => W1, LPM_WIDTHB => W,
                LPM_WIDTHP => W2, LPM_WIDTHS => W2,
                LPM_REPRESENTATION => "SIGNED")
  PORT MAP (dataa => cps, datab => x, result => cpsx);

sub_2: lpm_add_sub
  -- Sub: \(i <= (c-s)x - (x-y)c\)
  GENERIC MAP (LPM_WIDTH => W2, LPM_DIRECTION => "SUB",
                LPM_REPRESENTATION => "SIGNED")
  PORT MAP (dataa => cpsx, datab => xmyc, result => i);

add_1: lpm_add_sub
  -- Add: \(r <= (x-y)c + (c+s)y\)
  GENERIC MAP (LPM_WIDTH => W2, LPM_DIRECTION => "ADD",
                LPM_REPRESENTATION => "SIGNED")
  PORT MAP (dataa => cmsy, datab => xmyc, result => r);

END flex;
APPENDIX C
EXCALIBUR.H HEADER FILE

/*
 * File: excalibur.h
 *
 * This file is a machine generated address map for a CPU named cpu.
 * /
#endif _excalibur_
define _excalibur_

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#ifdef __cplusplus
extern "C" {
#endif

// The Memory Map

#define na_boot_monitor_rom ((void *) 0x00000000) // altera_avalon_onchip_memory
#define na_boot_monitor_rom_base 0x00000000
#define na_boot_monitor_rom_end (void *) 0x00000400
#define na_boot_monitor_rom_size 0x00000400
#define na_cpu ((void *) 0x00000000) // altera_nios
#define na_cpu_base 0x00000000
#define na_uart1 ((np_uart *) 0x00000400) // altera_avalon_uart
#define na_uart1_base 0x00000400
#define na_uart1_irq 26
#define na_seven_seg_pio ((np_pio *) 0x00000420) // altera_avalon_pio
#define na_seven_seg_pio_base 0x00000420
#define na_timer1 ((np_timer *) 0x00000440) // altera_avalon_timer
#define na_timer1_base 0x00000440
#define na_timer1_irq 25
#define na_led_pio ((np_pio *) 0x00000460) // altera_avalon_pio
#define na_led_pio_base 0x00000460
#define na_button_pio ((np_pio *) 0x00000470) // altera_avalon_pio
#define na_button_pio_base 0x00000470
#define na_button_pio_irq 27
#define na_lcd_pio ((np_pio *) 0x00000480) // altera_avalon_pio

#endif
#define na_lcd_pio_base                        0x00000480
#define na_uart2_debug           ((np_uart *) 0x000004c0)
    // altera_avalon_uart
#define na_uart2_debug_base                    0x000004c0
#define na_uart2_debug_irq                     28
#define na_ext_ram               ((void *)     0x00040000)
    // altera_nios_dev_board_sram32
#define na_ext_ram_base                        0x00040000
#define na_ext_ram_end           ((void *)     0x00080000)
#define na_ext_ram_size                        0x00040000
#define na_ext_flash             ((void *)     0x00100000)
    // altera_nios_dev_board_flash
#define na_ext_flash_base                      0x00100000
#define na_ext_flash_end         ((void *)     0x00200000)
#define na_ext_flash_size                      0x00100000
#define na_null                           0
#define nasys_device_family               "APEX20KE"
#define nasys_flash_count                 1
#define nasys_flash_0                     na_ext_flash
#define nasys_pio_count                   4
#define nasys_pio_0                       na_seven_seg_pio
#define nasys_pio_1                       na_led_pio
#define nasys_pio_2                       na_button_pio
#define nasys_pio_2_irq                   27
#define nasys_pio_3                       na_lcd_pio
#define nasys_timer_count                 1
#define nasys_timer_0                     na_timer1
#define nasys_timer_0_irq                 25
#define nasys_uart_count                  2
#define nasys_uart_0                      na_uart1
#define nasys_uart_0_irq                  26
#define nasys_uart_1                      na_uart2_debug
#define nasys_vector_table      ((int *) 0x0007ff00)
#define nasys_vector_table_size           0x00000100
#define nasys_vector_table_end  ((int *) 0x00080000)
#define nasys_reset_address     ((void *) 0x00000000)
#define nasys_clock_freq                  33333000
#define nasys_clock_freq_1000             33333
#define nasys_debug_core                  0
#define nasys_printf_uart                 na_uart1
#define nasys_printf_uart_irq             na_uart1_irq
#define nm_printf_txchar                  nr_uart_txchar
#define nm_printf_rxchar                  nr_uart_rxchar
#define nasys_debug_uart                  na_uart2_debug
#define nasys_debug_uart_irq              na_uart2_debug_irq
#define nasys_main_flash        ((void *) 0x00100000)
#define nasys_main_flash_size             0x00100000
#define nasys_main_flash_end    ((void *) 0x00200000)
#define nasys_program_mem       ((void *) 0x00040000)
#define nasys_program_mem_size            0x0003ff00
#define nasys_program_mem_end   ((void *) 0x0007ff00)
#define nasys_data_mem          ((void *) 0x00040000)
#define nasys_data_mem_size 0x0003ff00
#define nasys_data_mem_end ((void *) 0x0007ff00)
#define nasys_stack_top ((void *) 0x0007ff00)

#define __nios_catch_irqs__ 1
    // Include panic handler for all irqs (needs uart)
#define __nios_use_constructors__ 1  // Call c++ static constructors
#define __nios_use_cwpmgr__ 1  // Handle register window underflows
#define __nios_use_fast_mul__ 1
    // Faster but larger int multiply routine
#define __nios_use_small_printf__ 0
    // Smaller non-ANSI printf, with no floating point
#define __nios_use_multiply__ 0  // Use MUL instruction for 16x16
#define __nios_use_mstep__ 1  // Use MSTEP instruction for 16x16
#define __nios_jtag_stdio__ 0
    // True if Host Communication is OCI and not UART
#define nasys_nios_num_regs 256  // Number of CPU registers
#define nasys_has_icache 0  // True if instruction cache present
#define nasys_icache_size 0  // Size in bytes of instruction cache
#define nasys_icache_line_size 2  // Size in bytes of each icache line
#define nasys_has_dcach 0  // True if instruction cache present
#define nasys_dcach_size 0  // Size in bytes of data cache
#define nasys_dcach_line_size 4  // Size in bytes of each dcach line
#define nasys_nios_register_count 256
#define PLUGS_PLUG_COUNT 5  // Maximum number of plugs
#define PLUGS_ADAPTER_COUNT 2  // Maximum number of adapters
#define PLUGS_DNS 1  // Have routines for DNS lookups
#define PLUGS_PING 1  // Respond to icmp echo (ping) messages
#define PLUGS_TCP 1  // Support tcp in/out connections
#define PLUGS_IRQ 1  // Run at interrupte level
#define PLUGS_DEBUG 1  // Support debug routines

#define nm_system_name_string "ref_32_system"
#define nm_cpu_name_string "cpu"
#define nm_monitor_string "std_32_3.0"
#define nm_cpu_architecture nios_32
#define nm_cpu_architecture_string "nios_32"
#define nios_32 1

// Structures and Routines For Each Peripheral

#define nm_bfpr(_x, _y) ({
    int __x = (_x), __y = (_y);
    asm volatile("usr0 %0,%2 ; does bfpr" \
        : "r" (__x) \
        : "0" (__x), "r" (__y));
    __x;
})
#define nm_bfpr_pfx(_p, _x, _y) ({
    int __x = (_x), __y = (_y);
    asm volatile("pfx " #_p "\n    \
        : "r" (__x) \
        : "0" (__x), "r" (__y));
    __x;
})
// Nios CPU Routines
void nr_installcwpmanager(void); // called automatically at by nr_setup.s
void nr_delay(int milliseconds); // approximate timing based on clock speed
void nr_zerorange(char *rangeStart, int rangeByteCount);
void nr_jumpresethist(void);

typedef void (*nios_callfromresetproc)(void);

void nr_callfromreset(nios_callfromresetproc procptr);

// Nios ISR Manager Routines
typedef void (*nios_isrhandlerproc)(int context);
typedef void (*nios_isrhandlerproc2)(int context, int irq_number, int interrupte_pc);
void nr_installuserisr(int trapNumber, nios_isrhandlerproc handlerProc, int context);
void nr_installuserisr2(int trapNumber, nios_isrhandlerproc2 handlerProc, int context);
int nr_setirqenable(int onoff);

// Nios GDB Stub Functions
void nios_gdb_install(int active);
#define nios_gdb_breakpoint() asm("TRAP 5")

// Nios OCI Defines
#define nios_oci_breakpoint() asm("TRAP 0")

// Default UART routines
void nr_txchar(int c);
void nr_txchar2(int c, int channel);
void nr_txstring(char *s);
int nr_rxchar(void);

// Debug UART routines
void nr_debug_txchar(int c);
void nr_debug_txstring(char *s);
int nr_debug_rxchar(void);

// JTAG IO routines
int nr_jtag_rxchar(const void* ociBase);
int nr_jtag_tx_ready(const void* ociBase);
void nr_jtag_txchar(int c, const void* ociBase);
void nr_jtag_txcr(void);
void nr_jtag_txhex(int x);
void nr_jtag_txhex16(short x);
void nr_jtag_txhex32(long x);
void nr_jtag_txstring(const char *s);

// Nios Private Printf Routines
int nr_printf(const char *fmt,...);
//int nr_fprintf( FILE *fp, const char *fmt, ... )
int nr_sprintf(char *sOut, const char *fmt,...);

#define nk_stdout       1
#define nk_stderr       2

#if __nios_use_small_printf__
#define printf nr_printf
#define fprintf nr_fprintf
#define sprintf nr_sprintf
#define setbuf(v1,v2) nr_setbuf(v1,v2)  // does nothing
#endif

#if __nios_debug__
#define NIOS_GDB_SETUP        
    nios_gdb_install(1);    
    nios_gdb_breakpoint();
#else
#define NIOS_GDB_SETUP
#endif

// debug Core Declarations
#define nasys_debug_core_irq 8

// debug registers offsets from base
enum
{
    np_debug_interrupt = 0,  //read-only, 4 bits, reading stops trace
    np_debug_n_samples_lsb,  //read-only, 16 bits
    np_debug_n_samples_msb,  //read-only, 16 bits
    np_debug_data_valid,     //read-only, 1 bit,
        //true when trace registers contain valid sample
    np_debug_trace_address,  //read-only, 16 or 32 bits
    np_debug_trace_data,     //read-only, 16 or 32 bits
    np_debug_trace_code,     //read-only, 16 or 32 bits
    np_debug_write_status,   //read-only, 1 bit,
        //true when read to readback tracedata
    np_debug_start,          //write-only, write any value to start
    np_debug_stop,           //write-only, write any value to stop
    np_debug_read_sample,    //write-only, write any value to read
    np_debug_trace_mode,     //write-only, 1 bit
    np_debug_mem_int_enable, //write-only, 16 or 32 bits ??????
    np_debug_ext_brk_enable, //write-only, 1 bit
    np_debug_sw_reset,       //write-only, reset sampels and trace memory
    np_debug_address_pattern_0 = 16,      //write-only, 16 or 32 bits
    np_debug_address_mask_0,    //write-only, 16 or 32 bits

89
np_debug_data_pattern_0,           //write-only, 16 or 32 bits
np_debug_data_mask_0,               //write-only, 16 or 32 bits
np_debug_code_0,                     //write-only, 16 or 32 bits
np_debug_address_pattern_1 = 24,    //write-only, 16 or 32 bits
np_debug_address_mask_1,            //write-only, 16 or 32 bits
np_debug_data_pattern_1,            //write-only, 16 or 32 bits
np_debug_data_mask_1,               //write-only, 16 or 32 bits
np_debug_code_1,                     //write-only, 16 or 32 bits
};

// debug Register Bits/Codes
enum
{
  ************************************************/
  // debug_interrupt register
  // bit numbers
  np_debug_interrupt_code_dbp0_bit = 0,
  np_debugInterrupt_code_dbp1_bit = 1,
  np_debug_interrupt_code_ibp0_bit = 2,
  np_debug_interrupt_code_ibp1_bit = 3,
  np_debug_interrupt_code_mem_bit = 4,

  // bit masks
  np_debug_interrupt_code_ext_mask = (0),
  np_debug_interrupt_code_dbp0_mask = (1<<0),
  np_debug_interrupt_code_dbp1_mask = (1<<1),
  np_debug_interrupt_code_ibp0_mask = (1<<2),
  np_debug_interrupt_code_ibp1_mask = (1<<3),
  np_debug_interrupt_code_mem_mask = (1<<4),

  ************************************************/
  // debug_trace_code register
  // bit numbers
  np_debug_trace_code_skp_bit = 1,
  np_debug_trace_code_fifo_full_bit = 2,
  np_debug_trace_code_bus_bit = 3,
  np_debug_trace_code_rw_bit = 4,
  np_debug_trace_code_intr_bit = 5,

  // bit masks
  np_debug_trace_code_skp_mask = (1<<0),
  np_debug_trace_code_fifo_full_mask = (1<<1),
  np_debug_trace_code_data_trans_mask = (1<<2),
  np_debug_trace_code_write_mask = (1<<3),
  np_debug_trace_code_intr_mask = (1<<4),
  #ifdef __nios32__
    np_debug_trace_code_skp_cnt_mask = (63<<2),
  #else
    np_debug_trace_code_skp_cnt_mask = (31<<2),
  #endif

  // useful constants
np_debug_trace_code_op_mask = (np_debug_trace_code_data_trans_mask|np_debug_trace_code_write_mask),
np_debug_trace_code_read = np_debug_trace_code_data_trans_mask,
np_debug_trace_code_write = (np_debug_trace_code_data_trans_mask|np_debug_trace_code_write_mask),
np_debug_trace_code_fetch = 0,

/*************************************************
// debug_code_* registers
// bit numbers
np_debug_break_code_read_bit = 0,
np_debug_break_code_write_bit = 1,
np_debug_break_code_fetch_bit = 2,

// bit masks
np_debug_break_code_read_mask = (1<<0),
np_debug_break_code_write_mask = (1<<1),
np_debug_break_code_fetch_mask = (1<<2),

/*************************************************/
// debug_write_status register
// bit numbers
np_debug_write_status_writing_bit = 0,
np_debug_write_status_nios32_bit = 1,
np_debug_write_status_trace_bit = 2,

// bit masks
np_debug_write_status_writing_mask = (1<<0),
np_debug_write_status_nios32_mask = (1<<1),
np_debug_write_status_trace_mask = (1<<2)
} ;

// debug Routine

//Get the number of trace samples
unsigned long nr_debug_num_samples (void);

//Stop debug core - this must be a function so that a branch is the last
// thing in the fifo. Otherwise a skip or a read/write
// might be logged without the ability to sync it up
void nr_debug_stop (void);

//Read a trace sample
void nr_debug_get_sample (unsigned int *trace_addr,
unsigned int *trace_data,
unsigned char *trace_code);

//must have a uart for these functions
#ifdef nasys_uart_count
#if (nasys_uart_count > 0)

//print cause of break
void nr_debug_show_break (void *uart);

91
// 0 for default printf uart

//dump the trace memory
void nr_debug_dump_trace (void *uart);
  // 0 for default printf uart

//ISR for debug interrupts
//Show cause of break, dump trace, and halt
void nr_debug_isr_halt (int context);
  // 0 for default printf uart otherwise uart base address

//Show cause of break, dump trace, and continue
void nr_debug_isr_continue (int context);
  // 0 for default printf uart otherwise uart base address

#endif
#endif

// debug macros
//Read a debug register
#define nm_debug_get_reg(ret, offset)   \
  asm volatile (   \
    "PFX 3 \n    WRCTL %1 \n    PFX 4 \n    RDCTL %0;" \n    :"=r" (ret) \n    :"r"(offset) \
  );

//Write a debug register
#define nm_debug_set_reg(val,offset)   \
  asm volatile (   \
    "PFX 3 \n    WRCTL %1 \n    PFX 4 \n    WRCTL %0;" \n    : /* no outputs */ \n    :"r"(val),"r"(offset) \
  );

//Set breakpoint 0
#define nm_debug_set_bp0(ap,am,dp,dm,cd)
  nm_debug_set_reg(ap,np_debug_address_pattern_0);
  nm_debug_set_reg(am,np_debug_address_mask_0);
  nm_debug_set_reg(dp,np_debug_data_pattern_0);
  nm_debug_set_reg(dm,np_debug_data_mask_0);
  nm_debug_set_reg(cd,np_debug_code_0);

//Set breakpoint 1
#define nm_debug_set_bp1(ap,am,dp,dm,cd)
  nm_debug_set_reg(ap,np_debug_address_pattern_1);
  nm_debug_set_reg(am,np_debug_address_mask_1);
  nm_debug_set_reg(dp,np_debug_data_pattern_1);
nm_debug_set_reg(dm,np_debug_data_mask_1);

nm_debug_set_reg(cd,np_debug_code_1);

// Set extended trace mode
#define nm_debug_set_extended_trace nm_debug_set_reg(1,np_debug_trace_mode);

// Set memory interrupt point
#define nm_debug_set_wrap_point(size)
    nm_debug_set_reg((size>>2),np_debug_mem_int_enable);

// Include nios cache-control definitions and macros
#include "nios_cache.h"

// UART Registers
typedef volatile struct
{
    int np_uartrxdata;  // Read-only, 8-bit
    int np_uarttxdata;  // Write-only, 8-bit
    int np_uartstatus;  // Read-only, 8-bit
    int np_uartcontrol; // Read/Write, 9-bit
    int np_uartdivisor; // Read/Write, 16-bit, optional
    int np_uartendofpacket; // Read/Write, end-of-packet character
} np_uart;

// UART Status Register Bits
enum
{
    np_uartstatus_eop_bit = 12,
    np_uartstatus_cts_bit = 11,
    np_uartstatus_dcts_bit = 10,
    np_uartstatus_e_bit = 8,
    np_uartstatus_rrdy_bit = 7,
    np_uartstatus_trdy_bit = 6,
    np_uartstatus_tmt_bit = 5,
    np_uartstatus_toe_bit = 4,
    np_uartstatus_roe_bit = 3,
    np_uartstatus_brk_bit = 2,
    np_uartstatus_fe_bit = 1,
    np_uartstatus_pe_bit = 0,

    np_uartstatus_eop_mask = (1<<12),
    np_uartstatus_cts_mask = (1<<11),
    np_uartstatus_dcts_mask = (1<<10),
    np_uartstatus_e_mask = (1<<8),
    np_uartstatus_rrdy_mask = (1<<7),
    np_uartstatus_trdy_mask = (1<<6),
    np_uartstatus_tmt_mask = (1<<5),
    np_uartstatus_toe_mask = (1<<4),
    np_uartstatus_roe_mask = (1<<3),
    np_uartstatus_brk_mask = (1<<2),
    np_uartstatus_fe_mask = (1<<1),
np_uartstatus_pe_mask   = (1<<0)
);

// UART Control Register Bits
enum
{
  np_uartcontrol_ieop_bit  = 12,
  np_uartcontrol_rts_bit   = 11,
  np_uartcontrol_idcts_bit = 10,
  np_uartcontrol_tbrk_bit  = 9,
  np_uartcontrol_ie_bit    = 8,
  np_uartcontrol_irrdy_bit = 7,
  np_uartcontrol_itrdy_bit = 6,
  np_uartcontrol_itmt_bit  = 5,
  np_uartcontrol_itoe_bit  = 4,
  np_uartcontrol_iroe_bit  = 3,
  np_uartcontrol_ibrk_bit  = 2,
  np_uartcontrol_ife_bit   = 1,
  np_uartcontrol_ipe_bit   = 0,

  np_uartcontrol_ieop_mask  = (1<<12),
  np_uartcontrol_rts_mask   = (1<<11),
  np_uartcontrol_idcts_mask = (1<<10),
  np_uartcontrol_tbrk_mask  = (1<<9),
  np_uartcontrol_ie_mask    = (1<<8),
  np_uartcontrol_irrdy_mask = (1<<7),
  np_uartcontrol_itrdy_mask = (1<<6),
  np_uartcontrol_itmt_mask  = (1<<5),
  np_uartcontrol_itoe_mask  = (1<<4),
  np_uartcontrol_iroe_mask  = (1<<3),
  np_uartcontrol_ibrk_mask  = (1<<2),
  np_uartcontrol_ife_mask   = (1<<1),
  np_uartcontrol_ipe_mask   = (1<<0)
};

// UART Routines
int nr_uart_rxchar(np_uart *uartBase);         // 0 for default UART
void nr_uart_txcr(void);
void nr_uart_txchar(int c,np_uart *uartBase);  // 0 for default UART
void nr_uart_txhex(int x);                     // 16 or 32 bits
void nr_uart_txhex16(short x);
void nr_uart_txhex32(long x);
void nr_uart_txstring(char *s);

// PIO Peripheral

// PIO Registers
typedef volatile struct
{
  int np_piodata;       // read/write, up to 32 bits
  int np_piodirection;  // write/readable, up to 32 bits, 1->output bit
  int np_piointerruptmask; // write/readable, up to 32 bits, 1->enable interrupt
  int np_pioedgecapture; // read, up to 32 bits, cleared by any write
} np_pio;

// PIO Routines
void nr_pio_showhex(int value); // shows low byte on pio named na_seven_seg_pio

// ----------------------------------------------

// Timer Peripheral

// Timer Registers
typedef volatile struct
{
  int np_timerstatus;    // read only, 2 bits (any write to clear TO)
  int np_timercontrol;  // write/readable, 4 bits
  int np_timerperiodl;  // write/readable, 16 bits
  int np_timerperiodh;  // write/readable, 16 bits
  int np_timezerosnapl; // read only, 16 bits
  int np_timezerosnaph; // read only, 16 bits
} np_timer;

// Timer Register Bits
enum
{
  np_timerstatus_run_bit   = 1,  // timer is running
  np_timerstatus_to_bit    = 0,  // timer has timed out
  np_timercontrol_stop_bit = 3,  // stop the timer
  np_timercontrol_start_bit= 2,  // start the timer
  np_timercontrol_cont_bit = 1,  // continuous mode
  np_timercontrol_ito_bit  = 0,  // enable time out interrupt

  np_timerstatus_run_mask  = (1<<1),  // timer is running
  np_timerstatus_to_mask   = (1<<0),  // timer has timed out
  np_timercontrol_stop_mask= (1<<3),  // stop the timer
  np_timercontrol_start_mask= (1<<2),  // start the timer
  np_timercontrol_cont_mask= (1<<1),  // continuous mode
  np_timercontrol_ito_mask= (1<<0)   // enable time out interrupt
};

// Timer Routines
int nr_timer_milliseconds(void);  // Starts on first call, hogs timer1.

// Nios Flash Memory Routines

// All routines take a "flash base" parameter. If -1 is supplied,
// nasys_main_flash is used.

int nr_flash_erase_sector
(
  unsigned short *flash_base,
unsigned short *sector_address
);

int nr_flash_erase
(
    unsigned short *flash_base
);

int nr_flash_write
(
    unsigned short *flash_base,
    unsigned short *address,
    unsigned short value
);

int nr_flash_write_buffer
(
    unsigned short *flash_base,
    unsigned short *start_address,
    unsigned short *buffer,
    int halfword_count
);

// ===========================================================
// Parameters for Each Peripheral, Excerpted From The PTF File

// -------------
// Parameters for altera_nios_custom_instruction named USR0_cpu

// Module_Name = bfproc
// Publish_Component = 0
// ci_macro_name = bfpr
// ci_operands = 2
// ci_has_prefix = 1
// Synthesize Imported_HDL = 1
// Port_Type = Custom Instruction
// HDL_Import = 1
// ci_instr_format = RR
// ci_cycles = 8
// Address_Width =
// Imported_Wait =
// Timing_Units =
// Component_Desc =
// File_Count = 1
// Port_Count = 8
// Show_Streaming =
// Simulate Imported_HDL = 0
// Setup_Value =
// Wait_Value =
// Hold_Value =
// Unit_Multiplier = 30.00030000300003
// Technology =

// -------------
// Parameters for altera_avalon_onchip_memory named boot_monitor_rom

// Writeable = 0
// Size_Value = 1
// Size_Multiple = 1024
// Contents = blank
// Shrink_to_fit_contents = 0
// CONTENTS = srec
// use_altsyncram = 0
// use_ram_block_type =
// altsyncram_ram_block_type =
// dual_port = 0

// ------------------

// Parameters for altera_nios named cpu

// CPU_Architecture = nios_32
// mstep = 1
// multiply = 0
// rom_decoder = 1
// wvalid_wr = 0
// num_regs = 256
// do_generate = 1
// include_debug = 0
// include_trace = 0
// reset_slave = boot_monitor_rom/s1
// reset_offset = 0x0
// vecbase_slave = ext_ram/s1
// vecbase_offset = 0x03ff00
// support_interrupts = 1
// implement_forward_br1 = 0
// support_rlc_rrc = 0
// advanced = 1
// CONSTANTS =
// mainmem_slave = ext_ram/s1
// datamem_slave = ext_ram/s1
// maincomm_slave = uart1/s1
// debugcomm_slave = uart2_debug/s1
// germs_monitor_id = std_32_3.0
// include_oci = 0
// oci_offchip_trace = 0
// oci_onchip_trace = 0
// oci_num_xbrk = 0
// oci_num_dbrk = 0
// oci_dbrk_trace = 0
// oci_dbrk_pairs = 0
// oci_debugreq_signals = 0
// cache_has_icache = 0
// cache_icache_size_k = 0
// cache_has_dcache = 0
// cache_dcache_size_k = 0

// ------------------

// Parameters for altera_plugs_library named altera_plugs_library
// CONSTANTS =

// ------------------
// Parameters for altera_avalon_uart named uart1

// baud             = 115200
// data_bits        = 8
// fixed_baud       = 1
// parity           = N
// stop_bits        = 2
// use_cts_rts      = 0
// use_eop_register = 0
// sim_true_baud    = 0
// sim_char_stream  = g40000

// ------------------
// Parameters for altera_avalon_pio named seven_seg_pio

// has_tri   = 0
// has_out   = 1
// has_in    = 0
// edge_type = NONE
// irq_type  = NONE
// capture   = 0

// ------------------
// Parameters for altera_avalon_timer named timer1

// always_run           = 0
// fixed_period         = 0
// snapshot             = 1
// period               = 1
// period_units         = msec
// reset_output         = 0
// timeout_pulse_output = 0
// mult                 = 0.001

// ------------------
// Parameters for altera_avalon_pio named led_pio

// has_tri   = 1
// has_out   = 0
// has_in    = 0
// edge_type = NONE
// irq_type  = NONE
// capture   = 0

// ------------------
// Parameters for altera_avalon_pio named button_pio

// has_tri   = 0
// has_out   = 0
// has_in    = 1
// edge_type = ANY
// irq_type = EDGE
// capture = 1

// ------------------
// Parameters for altera_avalon_pio named lcd_pio
// has_tri = 1
// has_out = 0
// has_in = 0
// edge_type = NONE
// irq_type = NONE
// capture = 0

// ------------------
// Parameters for altera_avalon_uart named uart2_debug
// baud = 115200
// data_bits = 8
// fixed_baud = 1
// parity = N
// stop_bits = 1
// use_cts_rts = 0
// use_eop_register = 0
// sim_true_baud = 0
// sim_char_stream =

// ------------------
// Parameters for altera_nios_dev_board_sram32 named ext_ram
// sram_memory_size = 256
// sram_memory_units = 1024
// sram_data_width = 32
// CONTENTS = srec

// ------------------
// Parameters for altera_nios_dev_board_flash named ext_flash
// CONTENTS = srec

#ifdef __cplusplus
}  // 
#endif
#endif // _excalibur_

// end of file
APPENDIX D
NIOS COMPILER AND ASSEMBLER OUTPUT

tempa = Are2;
Are2 = Are2 + Bre2;

High-level language program in C

C compiler

Assembly language program for the Nios processor

Assembler

Binary machine language program for the Nios processor

1111000110011111100000101011011
1110101100111111100000101011111
1111000110011111100000101011011
111011111001111110000101011111
111011111001111110000101011111
01000001000000000111110000011111
11000001010111110000000000000000
APPENDIX E
C PROGRAM FOR PERFORMANCE EVALUATION

/*
 * Time Check Program for Performance Evaluation (timecheck.c)
 * This program uses timer subroutines generated in the excalibur.h file
 */

#include "excalibur.h"
#include <math.h>

/*@*******************************************************************************
 // Included all this extra declaration for timer -- ALTERA
/*@*******************************************************************************
#define TIMER_LOAD_VAL 0xFFFFFFFF
#define take_float_as_int32(x) (*((int *)(&(x))))
#define take_int32_as_float(i) (*((float *)(&(i))))

cmp_timer *timer = na_timer1;
typedef unsigned long DWORD;
typedef struct
{
    long interruptCount; // increment with each interrupt
    np_timer *timer;
} TimerISRContext;

static TimerISRContext gC = {0,0};

//global variable
int interrupt_count;

void MyTimerISR(int context)
{
    TimerISRContext *c;
    c = (TimerISRContext *)context;
    c->interruptCount++;
    printf("\n(timer #%d!)",c->interruptCount);
    interrupt_count = c->interruptCount;
    c->timer->np_timerstatus = 0; // write anything to clear the IRQ
long GetTickCount()
{
    volatile long timerVal;
    volatile long timerPeriod;

    timer->np_timersnapl = 0;    // snapshot
    timerVal = (timer->np_timersnapl & 0x0ffff)
               + ((long)timer->np_timersnaph << 16);

    return timerVal;
}

void InitTimer()
{
    long timerPeriod = TIMER_LOAD_VAL;

    //Initialize timer
    timer->np_timerperiodh = timerPeriod >> 16;
    timer->np_timerperiodl = timerPeriod & 0xffff;

    //Set timer to continuous
    timer->np_timercontrol = timer->np_timercontrol
                             | np_timercontrol_cont_mask;

    //Enable timer interrupt
    gC.timer = na_timer1;
    nr_installuserisr(na_timer1_irq, MyTimerISR, (long)&gC);
    gC.timer->np_timercontrol = gC.timer->np_timercontrol
                                | np_timercontrol_ito_mask;
    printf("

    Timer interrupt enabled.

    ");

    //Start the timer
    timer->np_timercontrol = (timer->np_timercontrol & 3)
                          + np_timercontrol_start_mask;

}

void CheckTimeStamp (DWORD dwStartTick, DWORD lTicksUsed, DWORD timer_overhead)
{
    volatile DWORD duration;
    volatile long int_cnt;

    int_cnt = (interrupt_count-1) * TIMER_LOAD_VAL;
    duration=0;

    if (interrupt_count >=2){

}
duration = TIMER_LOAD_VAL - lTicksUsed;
duration += dwStartTick;
duration += int_cnt;
}
else
{

duration = dwStartTick - lTicksUsed;
}

printf("\nNumber of clock cycles (minus overhead): %lu", duration -
timer_overhead);
printf("\n\n\n************************************************");

int main(void)
{
    volatile long int Are, Aim, Bre, Bim, Wr, Wi, W1, W2;
    volatile long int temp1, temp2, temp3, temp4, temp, n;
    volatile long int Are2, Aim2, Bre2, Bim2, Wr2, Wi2, tempa;
    volatile DWORD timer_overhead;
    volatile DWORD dwStartTick;
    volatile DWORD lTicksUsed;
    volatile DWORD our_dwStartTick;
    volatile DWORD our_lTicksUsed;

    Are2 = 10;
    Aim2 = 20;

    Bre2 = 30;
    Bim2 = 40;

    Wr2 = 23167; // Quantized to 16 bits.
    Wi2 = -23167; // Quantized to 16 bits

    printf("\n\n\n");
    printf("Hello, from Nios!\n");

    //Initialize the timer
    InitTimer();

    //timer_overhead, assume no interrupts
    dwStartTick=GetTickCount(); /* record start time*/
lTicksUsed=GetTickCount(); /* record end time */

timer_overhead = dwStartTick - lTicksUsed;

/*****************
/* BUTTERFLY     */
******************/
dwStartTick=GetTickCount(); /* record start time*/

tempa = Are2;
Are2 = Are2 + Bre2;
Bre2 = tempa - Bre2;

tempa = Aim2;
Aim2 = Aim2 + Bim2;
Bim2 = tempa - Bim2;

tempa = Bre2;
Bre2 = Bre2 * Wr2 - Bim2 * Wi2;
Bim2 = Bim2 * Wr2 + tempa * Wi2;

Bre2 = Bre2 / 32768;
Bim2 = Bim2 / 32768;

lTicksUsed=GetTickCount(); /* record end time */
printf("\nSoftware Implementation End");
CheckTimeStamp (dwStartTick, lTicksUsed, timer_overhead);

printf("\n(Are2) + i (Aim2)  is (%ld)+i(%ld)", Are2, Aim2);
printf("\n(Bre2) + i (Bim2)  is (%ld)+i(%ld)", Bre2, Bim2);

/***************************************************************/

Are = 20;
Aim = 40;

Bre = 60;
Bim = 80;

Wr = 23167; // Value of C
Wi = -23167; // Value of S
W1 = 0; // Value of C+S
W2 = 46334; // Value of C-S
our_dwStartTick=GetTickCount();  /* record start time*/

nm_bfpr_pfx(1,Are,Aim);           // Read Are and Aim
nm_bfpr_pfx(2,Bre,Bim);           // Read Bre and Bim
nm_bfpr_pfx(3,Wr,0);              // Read c
nm_bfpr_pfx(4,W1,W2);             // Read c+s and c-s
temp1 = nm_bfpr_pfx(5,0,0);       // Write the real value
temp2 = nm_bfpr_pfx(6,0,0);       // Write the imaginary value

temp3 = nm_bfpr_pfx(7,0,0);       // Write the real value
temp4 = nm_bfpr_pfx(8,0,0);       // Write the imaginary value

our_lTicksUsed=GetTickCount();    /* record end time */
printf("\nCustom Instruction Implementation End");
CheckTimeStamp (our_dwStartTick, our_lTicksUsed, timer_overhead);

printf("\n(Are) + i (Aim)  is (%ld)+i(%ld)", temp1, temp2);
printf("\n(Bre) + i (Bim)  is (%ld)+i(%ld)", temp3, temp4);
printf("\n\n");
printf("Good bye.\n\004");          // control-D tells terminal program

} // End of the Program
REFERENCES


[18] Altera Corporation, “Nios PIO Data Sheet”.

106


BIOGRAPHICAL SKETCH

_Divya Lakshmi Sunkara_

DivyaLakshmi Sunkara was born in Ongole, Andhra Pradesh, India on April 17th, 1980. She is the only daughter of Mr. Sunkara Obul Reddy and Mrs. Ramani. She did her schooling at D.A.V Higher Secondary School, Chennai. She then graduated from Madras University with BS degree (BE) in Electronics and Instrumentation Engineering with distinction. She started her Masters of Science program in Electrical and Computer Engineering at Florida State University in fall 2002 and graduated in Summer 2004. Her areas of interest are microprocessors, digital IC and ASIC design.