INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

NOTE TO USERS

This reproduction is the best copy available

UMI

Handling Large Data Storage in Synthesis of Multiple FPGA Systems

Amal Khailtash

A Thesis

in

The Department

of

Electrical and Computer Engineering

Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montreal, Quebec, Canada

September 1999

© Amal Khailtash, 1999

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-43653-5

Canadä

Abstract

Handling Large Data Storage in Synthesis of Multiple FPGA Systems

Amal Khailtash

Implementing DSP algorithms on single or multiple FPGAs has the advantages of short time to market, non-recurring engineering, and fast prototyping. Most of today's FPGAs provide fast arithmetic operations and large enough internal RAM storage that makes them very appealing to prototyping large systems, even building DSP applications. So having a good architecture to begin with is a good asset to engineers.

This thesis investigates the issues of handling large data storage in the synthesis of multiple FPGA systems especially in digital signal/image processing applications. In these applications very simple to complex algorithms are performed on large amounts of data - an image. An efficient way to store and access these data, the storage of intermediate variables locally or on RAM, is presented. The maximum pipeline level is extracted based on this storage and access scheme. A generic architecture for execution of arbitrary DSP algorithms with multiple memory banks is proposed. An ILP formulation for assigning memory banks to variables is presented. For demonstration purposes, a pipelined complex FFT has been developed in VHDL and the efficient storage and access order for this algorithm is presented. Also, based on these storage/access orders, the generation of addresses is done using hardware address generators.

Acknowledgments

I wish to thank my supervisor, Professor Baher S. Haroun, who helped me get started on this thesis. His insights and guidance have always been to the point and very helpful in this voyage. He always has bright ideas and is passionately pursuing his goals. His persistence and knowledge has always been my inspiration.

I also thank my other supervisor, Professor Asim Al-Khalili, who has been very patient with me and has always encouraged me to finish this thesis. He is always listening and trying to be as helpful as he can. His knowledge and his experience in this field is admiring.

I dedicate this thesis to my father, my mother, and my wife. I thank my parents for bearing me a knowledge-seeking person and raising me right. I also thank my wife for being patient and encouraging me to finish my thesis.

Table of Contents

List of Figures vii

List of Tables viii

List of Acronyms ix

1. Introduction 1
   1.1. Motivation 5
   1.2. Outline 6

2. High-Level Synthesis and FPGA design Flow 8
   2.1. Electronic Design Automation and Synthesis 8
   2.2. Operations Done in High-Level Synthesis of Architectures 10
   2.3. FPGA Design Flow 13

3. Handling Memory in Synthesis of Architectures 17
   3.1. Architectural Transformations 17
   3.2. Memory and Loop Transformations 18
   3.3. Studying Different Methods 19
   3.4. Architecture Proposed for Executing DSP Algorithms 21
   3.5. Extracting the Maximum Pipeline Level 26
   3.6. Scheduling the Graph 29

4. Memory Bank Assignment 31
   4.1. Exhaustive Search of the Solution Space 32
      4.1.1. Implementation Details 34
      4.1.2. Results from the Exhaustive Search 35
   4.2. Formulating the Problem in ILP 35
      4.2.1. Automatic Generation of the ILP Source for Arbitrary FFT 40
      4.2.2. Results from the ILP Formulation 40

5. Memory Address Assignment and Generation 42
   5.1. Address Assignment 42
   5.2. Address Generation 43
      5.2.1. Software-based Address Generation 44
      5.2.2. Hardware-based Address Generation 45

6. Using the Techniques in an Example Design 47
   6.1. Which FFT Algorithm Implementation to use? 47
   6.2. An Efficient Architecture for a 1024-point Complex FFT 49
   6.3. FFT Signal-flow Graph and Memory Access Pattern 50
   6.4. Manipulating Memory Access Patterns 52

7. Detailed VHDL Design 55
   7.1. Design of the Data Path and Its Elements 55

7.1.1. Addition Schemes 56
7.1.2. Multiplication Schemes 59
7.1.3. FFT Butterfly Data Path Implementation 63
7.2. Design of the Control Logic 67
7.3. Design Synthesis 70
7.3.1. Synthesis results 71
7.3. Constructing a Testbench 71
7.4.1. Results from the simulation and the FFT benchmarks 72

8. Conclusions and Future Work 75

8.1. Conclusions 75
8.2. Suggested Directions to Continue This Work 76

Bibliography 78

Useful URL Resources 85

Appendix 87

A. Memory Bank Assignment Exhaustive Search 87
A.1. Exhaustive Search C Source Program for 16-point radix-4 FFT 87
A.2. Sample Output of the Exhaustive Search 90
B. Program to Generate the ILP Source File for Arbitrary FFT 92
B.1. Program (GILP_FFT.C) for Generating Bank Assignment ILP, arbitrary FFT 92
B.2. ILP Source (FFT_16_2.GMS) for 16-point radix-2 FFT, Two Memory Banks 96
C. Program to Generate FFT Twiddle Factors 99
C.1. C Source Program TWIDDLE.C 99
C.2. Sample Output of the Program for a 256-point FFT 100
D. C Source File Used to Design a Hardware Address Generator 101
D.1. C Source File ADDGEN.C 101
D.2. Sample #1 106
D.2. Sample #2 106
D.2. Sample #3 107
D.2. Sample #4 107
D. VHDL Source Files for 1024-point Complex FFT 108
D.1. FFT Component Hierarchy 108
D.2. addrgen_bitrev.vhd 108
D.3. addrgen_linear.vhd 109
D.4. butterfly.vhd 110
D.5. cfft1024.vhd 112
D.6. controller.vhd 118
D.7. mem_bank.vhd 123
D.8. mult.vhd 124
D.9. reg_pipe.vhd 126
D.10. reg_pipe_single.vhd 127
D.11. skew_buffer.vhd 128
D.12. twiddle_factor.vhd 129
D.13. Testbench "cfft1024_tb.vhd" 131

List of Figures

Figure 1. A reconfigurable board. 3
Figure 2. Tasks in a high-level synthesis tool 12
Figure 3. FPGA design flow 15
Figure 4. Architectural transformations. 17
Figure 5. Proposed architecture 22
Figure 6. 8-point, radix 2, in-place FFT (Cooley-Tukey) 23
Figure 7. A number of folded implementations of the FFT 24
Figure 8. Sequence of operations in execution of the graph 25
Figure 9. Sequence of operations, showing the parallelism achieved. 25
Figure 10. Different scan orders for the sample FFT graph 27
Figure 11. Retiming and pipelining illustrated 28
Figure 12. Scheduled FFT with two-stage pipelined butterfly core and variable lifetimes. 30
Figure 13. Radix-4 16-point FFT 31
Figure 14. Pictorial representation of a symbol 33
Figure 15. Software-based (microcontroller) address generator 44
Figure 16. Simple address generator 46
Figure 17. Decimation-in-time and decimation-in-frequency butterflies 48
Figure 18. Proposed architecture for complex FFT 50
Figure 19. Cooley-Tukey FFT access patterns 51
Figure 20. Modified accesses for 4 & 8-point Cooley-Tukey FFT (two memory banks) 53
Figure 21. Final FFT architecture with skew buffer registers 54
Figure 22. A few different adder structures 57
Figure 23. Areas for different adder architecture 58
Figure 24. Speed for different adder architectures 59
Figure 25. Different multiplication architectures 61
Figure 26. DIF butterfly engine data path details 64
Figure 27. Skew buffer detailed schematic 65
Figure 28. Top-level module for 1024-point complex FFT and its I/O timing 68
Figure 29. Simplified state diagram of the controller 70
Figure 30. Basic simulation testbench 72
Figure 31. FFT benchmarks results (chart) 74

List of Tables

Table 1. The set of input/output connections to the inputs & outputs the butterfly 26
Table 2. Results for radix-4, 16-point FFT and two memory banks (exhaustive search)...35
Table 3. Sample assignments of symbols to iterations and nodes' outputs 36
Table 4. Sample assignments of symbols to iterations and nodes' inputs 37
Table 5. Constraints used for the 16-point FFT memory bank assignment.........................38
Table 6. Base TotalBanks equivalent of symbols in16-point FFT and 3 memory banks....39
Table 7. Results for radix-4, 16-point FFT and two memory banks...................................40
Table 8. Results for radix-4, 16-point FFT and three memory banks.................................41
Table 9. Results for radix-8, 64-point FFT and two memory banks...................................41
Table 10. FFT benchmark results (tabulated)...73

List of Acronyms

ALAP	As Late As Possible (Scheduling)
ALU	Arithmetic Logic Unit
ASAP	As Soon As Possible (Scheduling)
ASIC	Application Specific Integrated Circuit
ATPG	Automatic Test Pattern Generator
CAD	Computer-Aided Design
CAM	Content-Addressable Memory
CFFT	Complex Fast Fourier Transform
CFG	Control Flow Graph
CDFG	Control Data Flow Graph
CIA	Carry-Increment Adder
CLA	Carry-Lookahead Adder
CLB	Configurable Logic Block
COSA	Conditional-Sum Adder
CPA	Carry-Propagate Adder
CSA	Carry-Save Adder
CSKA	Carry-Skip Adder
CSLA	Carry-Select Adder
DA	Distributed Arithmetic
DIF	Discrete In Time
DIT	Discrete In Frequency
DFG	Data Flow Graph
DFT	Design For Testability
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
EDA	Electronics Design Automation
EDIF	Electronic Design Interchange Format
FA	Full Adder
FCFS	First Come, First Served
FIFO	First In, First Out
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
GAG	Generic Address Generator
HA	Half Adder
HDL	Hardware Description Language
IP	Intellectual Property
ILP	Integer Linear Programming
LUT	Lookup Table
MTTM	Mean Time To Market
MPEG	Moving Pictures Experts Group

MUX	Multiplexer
NRE	Non-Recurring Engineering
PCI	Peripheral Component Interconnect
PDG	Polyhedral Dependence Graph
PPA	Parallel-Prefix Adder
PPA-BK	Parallel-Prefix Adder Brent-Kung implementation
PPA-KS	Parallel-Prefix Adder Kogge-Stone implementation
PPA-SK	Parallel-Prefix Adder Skansky implementation
RAM	Random Access Memory
RCA	Ripple-Carry Adder
ROM	Read Only Memory
RTL	Register Transfer Level
SDF	Standard Delay Format
SFG	Signal Flow Graph
SOC	System On a Chip
VHDL	VSHIC (Very High-Speed Integrated Circuit) High-level Description Language
VLIW	Very Large Instruction Word

Chapter 1

1. Introduction

With today's increasing need for processing power in the telecommunications and other industries, new techniques are used to accelerate the design turn around and to decrease the area/power consumption of the system, yet increase the system performance. New high-level synthesis tools should consider many factors and try to manipulate the system definition based on the designer's specification at a higher level of abstraction before going down to the RTL[1] code optimization and the final physical implementations. The need for more architectural enhancements, either manually or by a high-level architectural synthesis tool is more essential and evident.

New techniques based on hardware/software codesign, which has recently come to the attention of many researchers [1], [2], [3], [4], [5], [6] and the industry, try to merge all aspects of system design in one unified environment that can tackle the problem and do optimizations at all levels and across multiple domains. Codesign tools allow a designer to specify an algorithm at a high level of abstraction. The tool does a lot of optimizations and finally partitions the design into a software module that would reside on a general purpose processor and another module that would go into a dedicated logic such as multiple FPGA[2] or ASIC[3].

---

[1] Register Transfer Level

[2] Field Programmable Gate Array

With today's million-gate FPGAs, one can put more functional units like multiplier-accumulator blocks in parallel and achieve a higher performance. It is also possible to find a vast variety of soft and hard cores ranging from different DSP[4] algorithms, microprocessors, PCI[5] interface cores, to large cores like MPEG[6] encoder/decoder chips, network controller chips, and communications systems building blocks. There are many design houses and independent designers that work only on creating IP[7] cores, which come complete with testbenches and documentation and sometimes even the source codes, using the latest EDA[8] tools.

FPGA devices are ideal prototyping tools for small to medium size systems. The MTTM[9] for systems implemented using FPGAs is small. The NRE[10] associated with the system is also low compared to an ASIC because of the fewer number of steps needed to arrive at the final design and the chance to enhance previous designs faster and try different designs in less time. FPGAs also have the advantage of reconfigurability. The concept of on-the-fly reconfigurable boards is not new. In fact there have been many papers on this subject [7]. There are also commercial products that make use of this technique and the reconfigurability of the SRAM based FPGAs. One such product is the MEGA-OPS system that has multiple FPGAs and three memory banks on a single board.

---

[3] Application Specific Integrated Circuit

[4] Digital Signal Processing

[5] Peripheral Component Interconnect

[6] Moving Pictures Experts Group

[7] Intellectual Property

[8] Electronic Design Automation

[9] Mean Time To Market

[10] Non-Recurring Engineering

Their goal is to implement hardware accelerator boards that speed up the computations on a personal computer. They use a C-style language to specify the algorithm and a compiler that compiles it to an intermediate form and finally to a form suitable to be downloaded into the FPGAs on the board. Once the FPGAs are configured the board can execute at a much faster speed and the speed-up gained is much more than the software only implementation of the algorithm. It also has the flexibility of the software; i.e., one can change the algorithm and download a new one into the board and use the system for a different purpose.

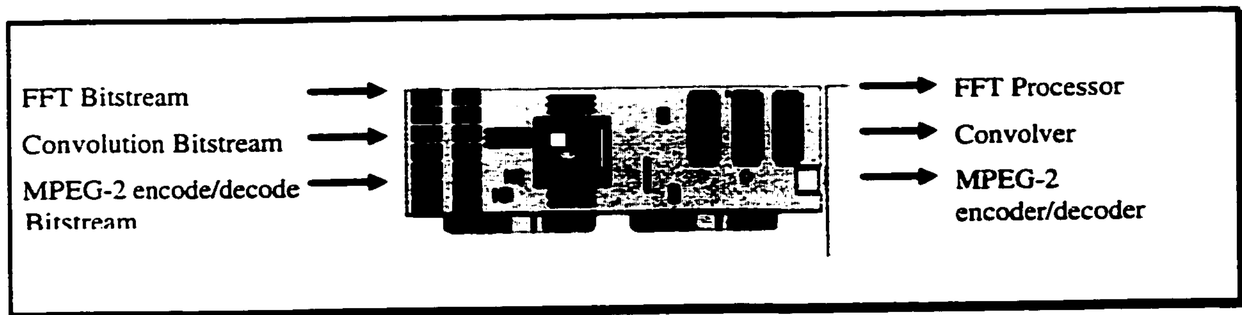


Figure 1. A reconfigurable board.

One can also achieve a higher performance by parallel implementation of algorithms on an FPGA or dedicated logic than implementing it on a general purpose DSP processor. This is true if one can convert a floating point algorithm to its fixed-point counterpart with reasonable resolution, FPGAs could have advantages over general purpose DSP processors. Otherwise floating-point operations are better done on floating-point DSP processors. Currently, the DSP processors have a few (usually one or two) built-in multiplier-accumulator units that are the essential part of most digital signal processing algorithms. New breeds of architectures from Texas Instruments, Analog

Devices, Lucent Technologies and Motorola are using VLIW[11] processors with multiple data pipelines to improve the performance and throughput of the processor for these applications.

Most signal processing applications, especially those in the field of image processing, need to access large amounts of data that are normally stored in RAM. The way in which this access is done highly affects the final architecture. Also the way one sets the constraints on the synthesis tool affects the performance and area of the final architecture obtained. The goal is to increase the memory bandwidth thus increasing the performance of the system, but this may add to the total area, which may not be very desirable in all applications. Therefore there is a trade-off between the area and the performance of the system. The area/performance factors also affect the final power consumption of the system.

The purpose of this work is to study a system architecture with multiple memory blocks that can be accessed simultaneously by the processing kernel, which will run the algorithm. These memory blocks allow the exploitation of parallelism that may increase the throughput of the system. Adding more memory blocks and more parallel data paths may not be optimal for different applications. More parallelism in an algorithm also puts constraints on the memory subsystem. One has to provide more data in parallel for alleviating the bottlenecks and not to starve the pipelines of the computing engine.

---

[11] Very Large Instruction Word

1.1. Motivation

High-level architectural synthesis tools have come a long way and have tackled different aspects of a design. There have been many studies on synthesizing and automating the generation of optimal data paths and control logic to execute a specific algorithm. In recent years, with advances in communications technology and the advent of complex DSP systems, architectural transformations and enhancements have become more important than ever. These optimizations tend to ignore the effects of auxiliary memory used in these algorithms. The way the variables are stored in memory and how they are accessed during the execution of the algorithm can dictate how the control structures work and can also affect the data path itself. The million gate era for FPGAs has arrived and as more and more functionality and architectural improvements appear in new FPGAs, the dream of millions of gates SOC[12] becomes a reality. But without proper tools and knowledge of the algorithm and different architectures, these devices may not be utilized as efficiently as possible. Architectural decisions and enhancement techniques are equally important to both FPGA and ASIC designs, but they are more important for ASIC flow with its associated NRE cost and time spent during the design.

This work emphasizes the importance of paying attention to the memory subsystem during architectural synthesis and enhancements that can be achieved by proper selection of the number of memory banks and scheduling of the read/write operations. Traditional techniques are reviewed and different views on the subject are explored.

This study tries to find answers, techniques, formulations, heuristics and arrive at a novel architecture for a multiple memory system. The main issues are choosing the right

number of memory banks for a specific algorithm, correct schedule for the memory transactions and sketching the final design.

The techniques presented will assist in arriving at a better architecture with multiple memory banks that can be used for running different DSP algorithms. The architecture presented is simple yet effective. Later chapters will show this simplicity and how it makes a design based on this architecture to run much faster than others.

1.2. Outline

Chapter 2 starts with explaining the basics of high-level synthesis, especially architectural synthesis. The fundamental processes involved in arriving at an optimal architecture that can be used to run a variety of DSP algorithms are explained. The methods described are independent of the target technology used, whether it be ASIC or FPGA. The emphasis of the following chapters would be on FPGAs.

Chapter 3 concentrates on discussing different methods and issues found in papers dealing with architectural synthesis of algorithms that use memory to carry out their task. After showing different architectural transformations, a generic architecture for running DSP algorithms is proposed. A method to find the maximum pipeline level for various accesses to the scratch-pad (temporary) memory used for storage of intermediate and final variables is presented. This chapter ends by showing the effects of retiming and pipelining on the variable life times and thus the memory used. A schedule for an FFT algorithm will also be shown.

---

[12] System On a Chip

In chapter 4 a novel approach for finding the optimum number of memory banks for a specific algorithm is presented. First, an exhaustive search scheme that has very big run times is shown, and then the same problem is formulated using the Integer Linear Programming. From this chapter on, the FFT example algorithm is used throughout the work.

Chapter 5 goes over different techniques in generating addresses for a specific algorithm and finally shows a method to build a hardware address generator.

Chapter 6 uses the methods developed in the previous chapters to implement an FFT hardware engine.

Chapter 7 presents the detailed VHDL design of a complex FFT and shows the design challenges and issues. Different aspects of VHDL design of the data path and control logic for this optimized DSP algorithm is shown.

Chapter 8 gives future directions and brings up issues to be resolved in dealing with memory in architectural synthesis.

Chapter 2

2. High-Level Synthesis and FPGA design Flow

There are many steps involved in the high-level synthesis of architectures [8]. Followed by the architectural synthesis is the actual logic synthesis or silicon compilation. The results of architectural synthesis affects the outcome of the final design after logic synthesis; i.e., the design decisions made and tradeoffs used in choosing the architecture changes the area/speed grades of the result.

Nowadays many synthesis and EDA software companies' attentions are focused on making synthesis tools more aware of and capable of making architectural decisions to improve overall system performance, power and area.

2.1. Electronic Design Automation and Synthesis

The electronic industry is a very fast, dynamic filed that is also very competitive. To reduce the amount of time spent designing a system, design automation and synthesis are introduced. Electronic design automation deals with making most of the design steps automatic and faster to complete. It covers all aspects of the design from the design entry to implementation and finally design verification. Design automation allows the designer to try out different designs and come up with a good trade-off in the shortest amount of time. This lets the designer to arrive at the most optimum design needed for a specific application.

Design entry could be schematic, block diagram, state diagram and flow charts or other means of specifying the system. Design implementation EDA tools cover the synthesis, partitioning, placement and routing of the design. Examples of the design verification tools are high-level and gate-level simulators and automatic test bench generators.

Synthesis is the action of arriving at a circuit at the finest grain after specifying the system at a higher level of abstraction. Synthesis is usually divided in three different categories:

1. High-level synthesis
2. Logic Synthesis
3. Layout and physical synthesis

The high-level synthesis transforms the specification of a design, which is at the highest level and specifies the behavior of the system, to a structural netlist of interconnected components and RTL logic. This is explained in more detail in the next section.

Logic synthesis deals with converting the structural RTL specification of the design to an optimal (simplified) combinatorial and sequential logic mapped to a specific technology and cell library. Logic synthesis is not covered in this work (refer to [10]).

Layout and physical synthesis converts the mapped structural design into the exact physical geometry or layout of the design. This includes the actual placement and routing of the components.

2.2. Operations Done in High-Level Synthesis of Architectures

The first step in high-level synthesis is the compilation of the source description, whether it be an HDL or other high-level representation of an algorithm to an intermediate format. This intermediate format is transformed into a more suitable representation for high-level synthesis that is usually a Control Data Flow Graph (CDFG). A Control Data Flow Graph is referred to two directed graphs called a Control Flow Graph (CFG) and a Data Flow Graph (DFG). A CFG contains the flow of control in the original specification with nodes being the operation and the edges being the dependencies of operations. The DFG contains the flow of information from one operational unit to the other. These operations usually encompass compiler-like and hardware-specific transformations.

Some of the transformations at this stage include: converting more complex operations to simpler ones with the same functionality, increasing the parallelism in the operations, and reducing the number of data flow levels.

After a CDFG is extracted from the high-level language specification, from this CDFG, the control circuitry and the data path are derived.

The main tasks in high-level synthesis that should be done to derive an architecture from a system specifications are: Allocation, Binding and Scheduling. After these three steps the design is written out in a structural RTL language and passed to logic synthesis. The three main steps in high-level synthesis are explained briefly.

Allocation is the assignment of different functional elements for the system, including Functional Units (FU) - adders, multipliers, ALUs, etc.- Registers, Register Files, RAMs, Interconnections, Busses, MUXes, and Bus Drivers. The selection of

different functional units is based on the constrains passed to the synthesis tool. The allocation phase tries to select operations that seem to satisfy the timing constraint by looking at the DFG.

Binding is the assignment of operations to functional units, data transfers to busses, multiplexers and interconnections, variables to registers, register files and memory blocks, addresses to memory locations. Binding tries to optimize the sharing of hardware resources. Operations done at different cycles can share the same functional unit, variables that are not alive (needed) at the same time can share the same register or memory location, and data transfers that do not occur at the same time can share the same path (bus or multiplexer).

Scheduling is the assignment of data transfers, lifetimes, operations to clock cycles in a synchronous system. Scheduling tries to optimize the number of clock cycles needed to finish the algorithm given the constraint on the hardware resources and the number of clock cycles. This operation takes into account the control relationships specified in the CFG and also should consider the data dependencies specified in the DFG. Scheduling also deals with *chaining* of operations and *multi-cycle* operations.

These three tasks and their relationships is shown in Figure 2.

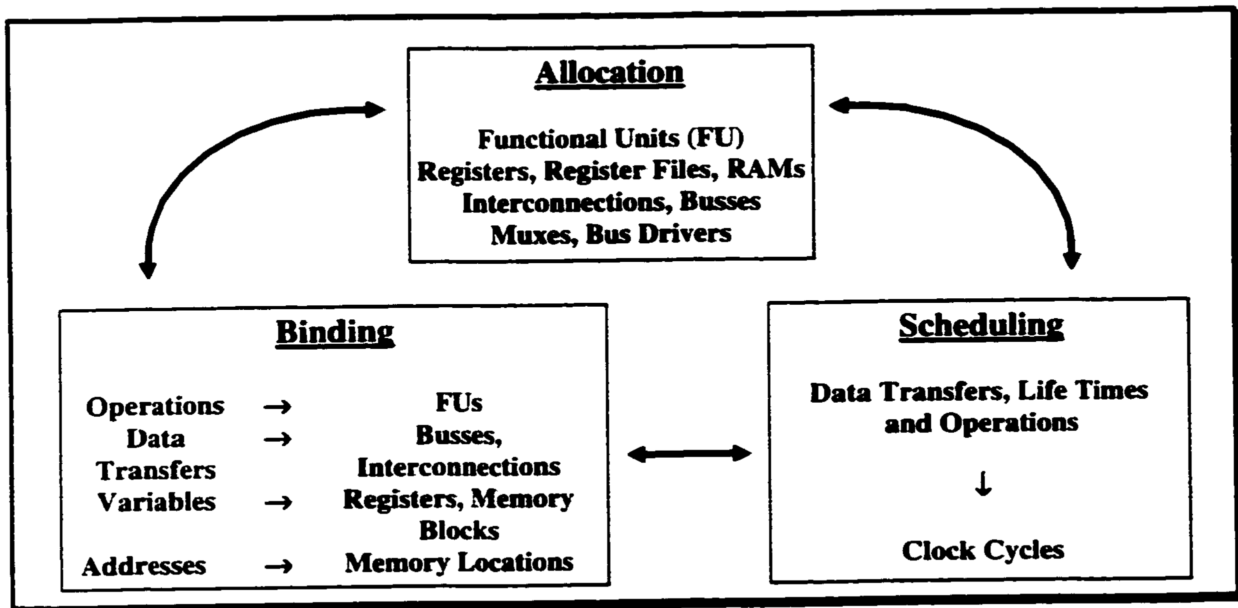


Figure 2. Tasks in a high-level synthesis tool

In a synthesis tool, these tasks are done to obtain an architecture from specification. The starting point for these tasks is usually Allocation. But there is a cycle among these three tasks that should be done a number of times to arrive at the desirable architecture based on the constraints put on the synthesis tool by user specification. Some synthesis tools break this cycle at some point or even do two or three of theses tasks together. The complexity of the tool increases as these tasks are done together, but the architecture obtained is closer to the optimal architecture because doing the processes together gives global visibility of the system to the tool.

There are different scheduling techniques. A few of them are:

1. First come, first served (FCFS) scheduling. This looks only at data dependencies and tries to schedule operations from the first to the last one whichever come first.

2. ASAP (as soon as possible) scheduling, by which, operations are scheduled as early as possible considering their dependencies.

3. ALAP (as late as possible) scheduling, by which, operations are scheduled at the latest possible maximum time allowed.

4. Critical path scheduling, also called mobility scheduling, schedules operations based on their mobilities. Mobility of an operation is the difference between its ALAP and ASAP schedule.

5. Lifetime scheduling tries to find a good schedule by minimizing the number of registers.

Other scheduling techniques are "Force Directed Scheduling", "List Scheduling", and "Look-ahead Scheduling".

In this study, work on binding variables to memory blocks and addresses to memory locations are undertaken. Other tasks are also reviewed as the final architecture is derived.

2.3. FPGA Design Flow

To arrive at the final programming bitstream for the FPGAs, a designer starts by system specification and then capturing the design with a design entry tool. Design entry can be pure schematics, pure HDL[13] code (VHDL[14]/Verilog), or mixed schematics and HDL. Following this is a simulation step, in which the functionality of the design is verified. After the verification step, is the actual synthesis of the HDL code and design optimization. The input to this step is the designer's timing and area constraints. The first step in the design implementation, is the HDL synthesis and mapping the design to the

[13] Hardware Description Language

[14] VHSIC (Very High-Speed Integrated Circuit) Hardware Description Language

target device (technology mapping). Then the mapped design is flattened and all the elements are placed considering the timing and placement constraints. After placement, is the automatic constraint-driven routing of the nets in the design and their interconnects. At the end of this step, the configuration bitstream for the FPGA is produced.

To verify the functionality at this stage, one must back-annotate the actual delays from the placed and routed design back to the flattened HDL netlist and do a timing or back-annotated simulation. The result of this simulation is to be compared with the result of the functional simulation. If the two results are within the allowable range of design specifications, the design cycle is complete. The complete flow is shown in Figure 3.

The ASIC design flow is very similar to the FPGA flow with a few more additional steps. There could be an RTL floor-planning before the actual synthesis to improve the area/performance. After the synthesis, which takes user constraints and the target technology's cell libraries, is the final floor planning and placement of the modules. For better design testability, a DFT[15] scan chain insertion step is done in which the IEEE 1149 boundary scan chain logic is inserted at the I/O boundaries.

An Automatic Test Pattern Generator (ATPG) module and a signature analyzer module could also be placed on chip to do self test and sanity check on the circuit.

[15] Design For Testability

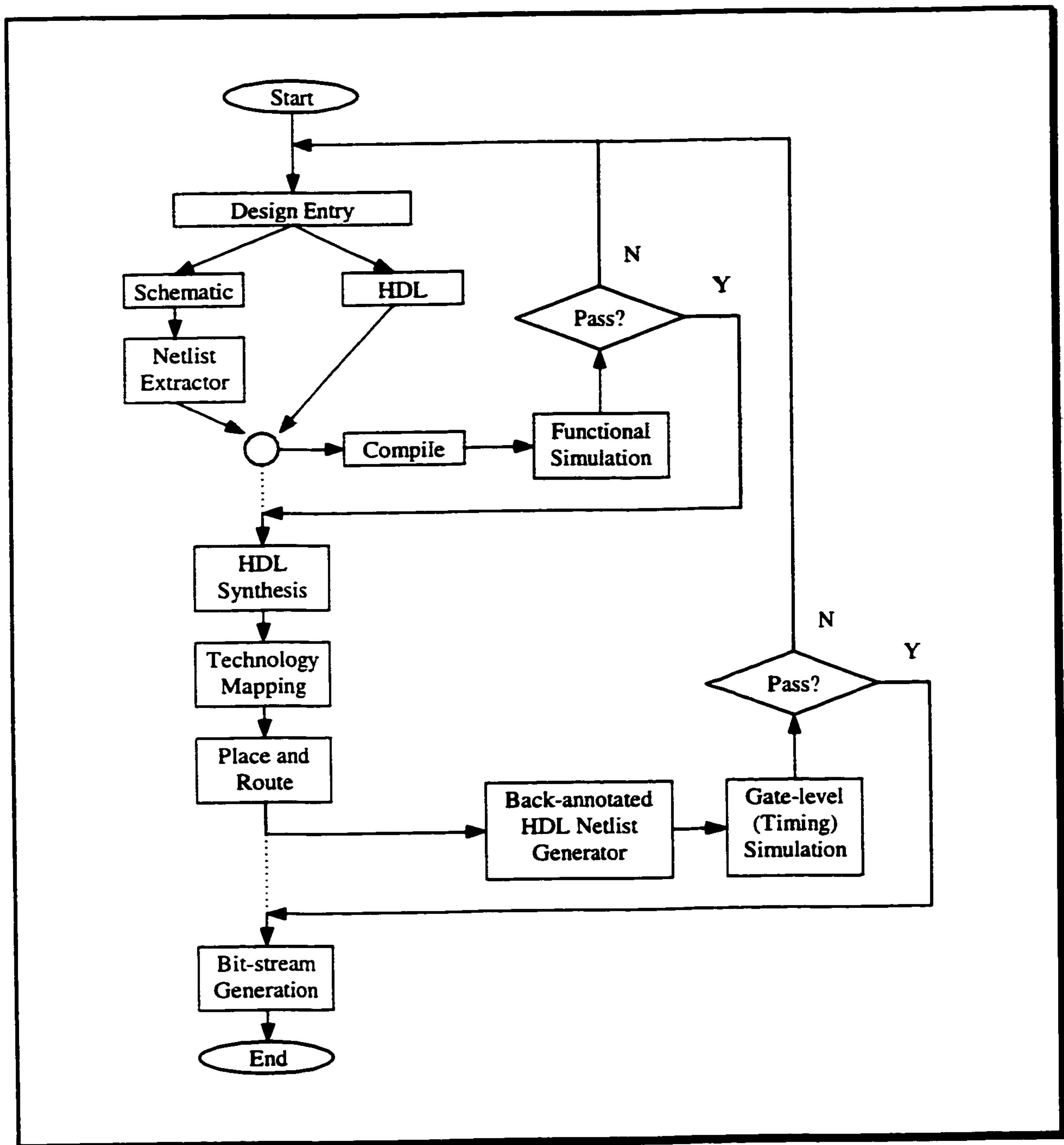


Figure 3. FPGA design flow

There could be an additional power optimization of the system in which some techniques are used to reduce the toggling rate of flip-flops thus reducing the power consumption of the system.

After the final routing is the delay extraction and generation of the back-annotated netlist that is usually in the EDIF[16] format or VHDL/Verilog netlist with associated SDF[17]. A comprehensive simulation is done at this stage and if there was a problem at this stage, the preceding steps could be repeated. After the final confirmation that the design satisfies the design specifications, the masks are generated and the chip layout is done. The masks are sent to the fabrication facilities where the chip is fabricated and packaged.

[16] Electronic Design Interchange Format

[17] Standard Delay Format

Chapter 3

3. Handling Memory in Synthesis of Architectures

In this chapter, different architectural transformations related to the synthesis are explored and then different memory access (loop) transformations are presented. Then some of the published techniques in dealing with synthesis of DSP algorithms that make use of memory as temporary storage are reviewed.

3.1. Architectural Transformations

There are very simple architectural transformations that can improve area and/or performance of a specific algorithm. Some of these transformations are results from compiler technology [9] applied to hardware synthesis [10], [11], [12].

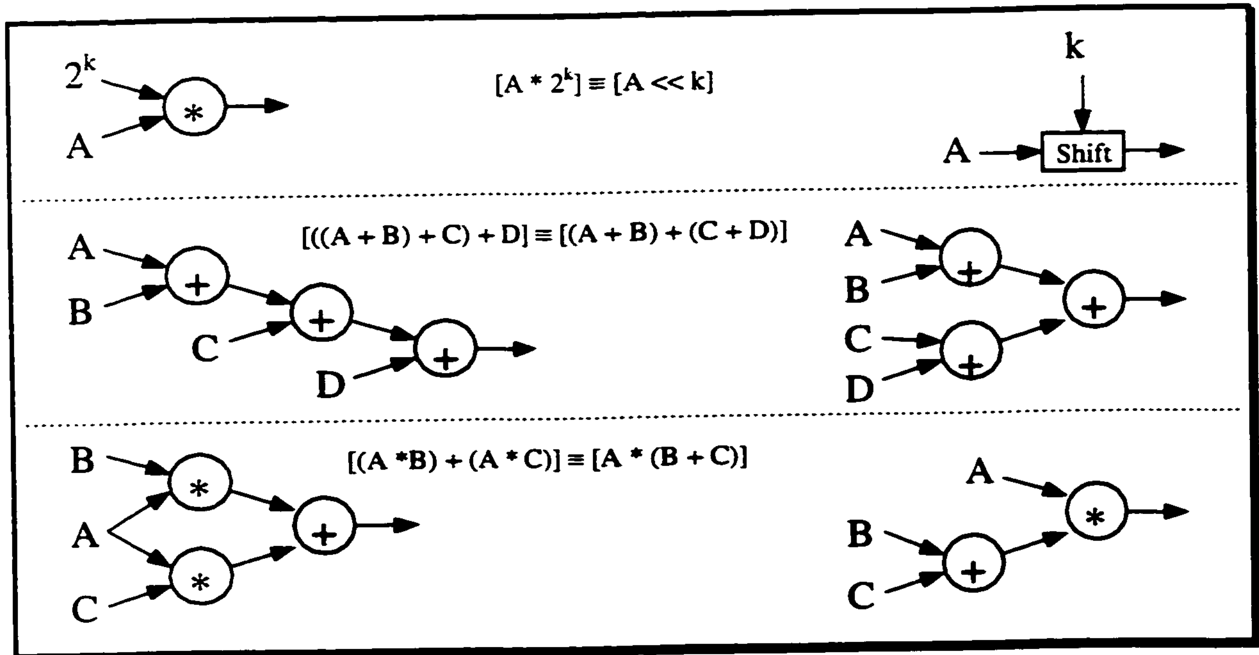


Figure 4. Architectural transformations.

As can be seen in Figure 4, one can see three different simple transformations, which improve the overall system area/performance by reducing the number of operational units and their associated delay.

The first transformation is the use of other functionally equivalent, more area/speed efficient operational units in place of more costly ones. An example would be using shifts instead of multiplication by a constant power of two number. This improves both area, speed and power consumption of the system.

The second transformation is using the association property of operations to merge and group multiple operations. This improves delay and therefore system performance.

The third transformation is distribution or what is usually called resource-sharing. And that is to factor and use the common part of multiple operations. This improves the resulting area and power consumption.

3.2. Memory and Loop Transformations

In section 3.5, the reader will see how arrangement of variables in memory (storage order) and accesses to those variables (access order) can affect the pipeline length and the variable lifetimes in memory. There are other ways to reduce the memory traffic (reads and/or writes) by using loop transformations.

One can observe the following different methods mentioned in [13]:

1. Loop-invariant removal tries to move the parts of the loop body that do not depend on the loop index to outside of the loop body.

2. Load-after-load optimization removes the second load from the loop body if the second access is to the same location as the first and the sequence of operations have not changed the value of the variable accessed.

3. Load-after-store optimization removes load if there were no other store operations to the same memory location and the internal variable is used instead of another memory access.

To improve the performance of computer systems and algorithms, one can increase the number of memory banks that provide data to a specific architecture and keep its internal pipelines fully utilized. Usually data interleaving is used for the storage of information in these multiple memory, parallel systems. But this may result in memory access contention and pipeline stalls. There are also other dynamic methods to increasing the performance of multiple memory, parallel systems by using dynamic storage schemes and address transformations [14], [15], [16].

3.3. Studying Different Methods

In this study, different approaches taken in different areas of computing applications have been looked at. One such approach is in the implementation of data structures and memory management strategy using window analysis in the Cathedral-II system [17]. It analyzes the algorithm and for a given number of memory ports reduces the total number of storage locations needed to a near minimum. First, the minimum number of locations that store each data structure separately in chunks of contiguous RAM locations, called pages, is found. Next, pages can share the same physical memory locations if their contents is not alive simultaneously. It was also observed that storage

order and access order of a data structure in memory, changes the amount of storage requirements. In this system, which is based on lifetime analysis, variables with disjoint lifetimes can share the same memory location thus reducing the memory size needed. The "window of an array" is that section of the array that should be alive in memory in order for the algorithm to run properly. With the change in the storage and access order of an array the window of the array changes and thus changing the amount of local memory required in the architecture.

Another approach taken, is using loop and control flow transformations using polyhedral dependence graphs (PDG) [18] and finding an ordering vector for optimal memory access having the bandwidth constraint as the maximum number of simultaneous memory accesses at any time point. Their approach is that all the intermediate variables that are sure to be consumed directly after their production do not have to be stored in background memory. One important point is that memory size is related to the maximum number of signal instances to be stored at any point of time for a given ordering of operations.

Another interesting paper is in the field of high-level-language compilers for parallel machines and the subject of loop transformations to increase parallelism. In this paper [19], a number of transformations are proposed to increase parallelism. The object of this paper is parallel computers that have fixed architecture and is different from what has to be done for this work; i.e.; compilation for an architecture that is unknown.

In another paper [20], a technique using Mathematics of Arrays and the ψ (PSI) calculus is used to generate addresses for data transfers that require less data transfers

than more traditional algorithms. But again this is targeted for general purpose processors with fixed architectures and single memory port that is not suitable for the purpose of this work.

In [21] a coprocessor engine using FPGAs for a general purpose DSP processor is shown that helps in the computation of a 3x3 convolution on a 2-D image data. They extract the window, or the active variables needed to compute one 3x3 convolution sum, from the algorithm and with the aid of FIFOs they supply enough data for the architecture implemented in the FPGA to compute the rest of the convolution.

It is known that how the data is stored in memory and how it is accessed can affect the memory requirements of the final architecture. In [22], [23], [24], it is shown that arranging the data in multiple memory banks for a parallel machine can change the throughput of the system. So there is a trade-off in the number of memory ports (blocks) and the amount of local storage inside an architecture for different algorithms.

3.4. Architecture Proposed for Executing DSP Algorithms

From the study of all these papers the following design is proposed (Figure 5) that is suitable to implement a number of different DSP algorithms with different degrees of parallelism. It is assumed that the algorithm is originally specified with some kind of loop structure and the inner core of the loop is specified as a signal flow graph. The memory blocks could be implemented as discrete memory or as embedded memories inside FPGA, which are abundant in today's FPGA architectures.

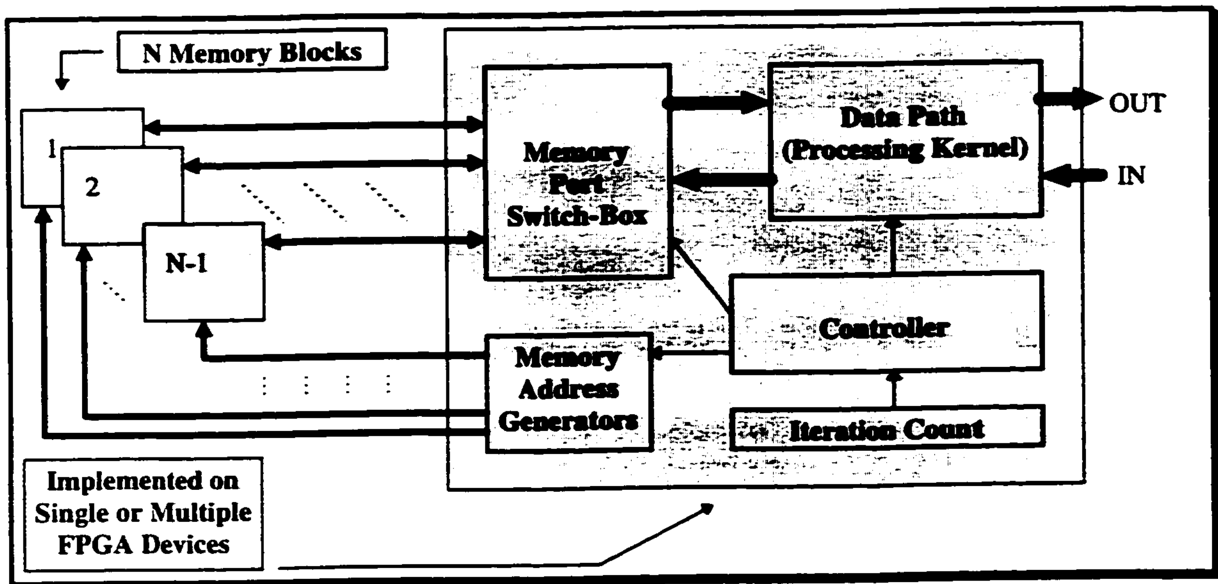


Figure 5. Proposed architecture

To illustrate this, an 8-point, radix 2, in-place FFT[18] based on this design is chosen for implementation (refer to [45] for detailed explanation of FFT). In this multiple port memory design, it is desirable to be able to pass data from each of memory blocks to the inputs of the data path kernel, which is implementing the inner core of the loop of the signal flow graph. And it should also be possible to store the outputs of the data path to any or all of the memory blocks. This is the reason for the memory port switch-box unit at the inputs and outputs of the data path to the memory blocks. This is derived from the fact that in many of the signal flow graph representation of algorithms, if the signal flow graph is repetitive (composed of similar operations), one can fold the graph and only implement the repetitive part and forward the proper data to the inputs of this folded graph. This can be seen in the signal flow graph of the FFT example; as in Figure 6.

[18] Fast Fourier Transform

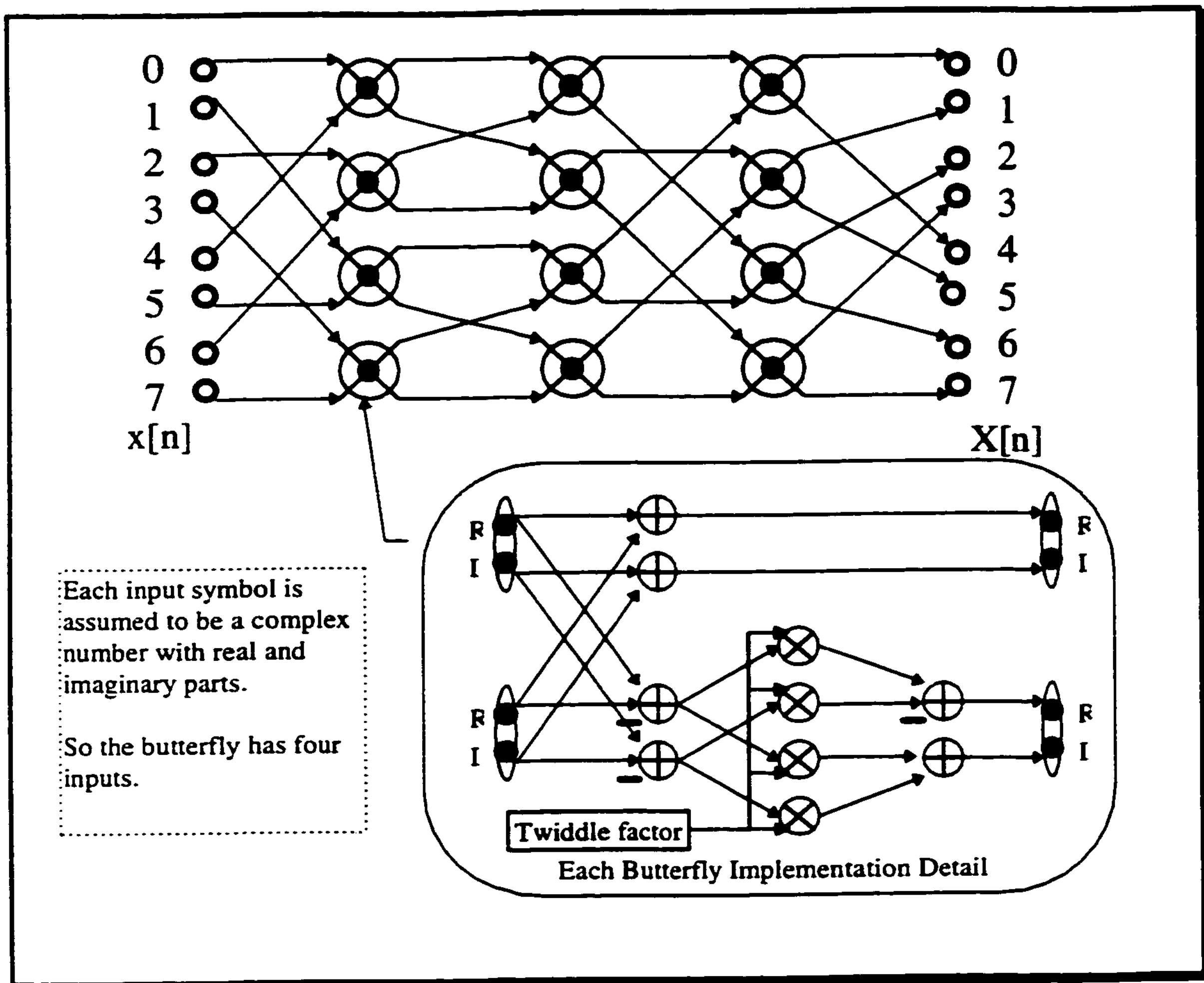


Figure 6. 8-point, radix 2, in-place FFT (Cooley-Tukey)

Assuming, all the memory accesses have been assigned to the variables that should be stored in memory, with a specific computation order, one should schedule the reads and writes of these variables and also bind them to a specific memory port.

To do this, the first task is to assume a computation order. For this purpose consider the fully folded graph of the FFT example; as in Figure 7.

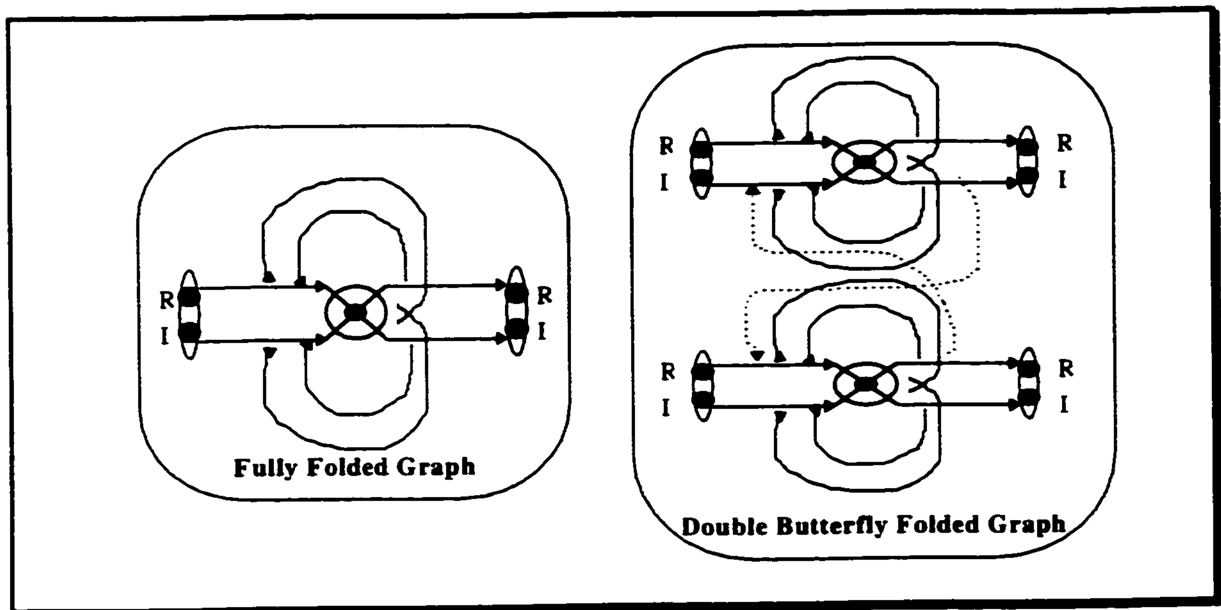


Figure 7. A number of folded implementations of the FFT

The computation order considered; having Figure 6 in mind, is a column-wise scan of the signal flow graph; i.e., the top-left butterfly is computed first, then the second top-left butterfly, then the third-top, and so on. For each step of the computation an iteration count is assigned; i.e., the first butterfly is assigned 0, the second 1, and so on. To continue the process, allocation, scheduling, and binding for the butterfly graph is done and the number of cycles needed to finish the operations is found. Then considering the architecture proposed in Figure 5, one should do the following operations one after another. In the first iteration of the loop, the necessary variables are supplied to the correct inputs of the butterfly (the data-path core in Figure 5), then it is time for the computation cycle of the butterfly itself, and then the output results are written to one or a number of memory blocks.

The memory switch-box is responsible to route the correct input to the data path and the result to each memory block. These series of operations can be seen in Figure 8.

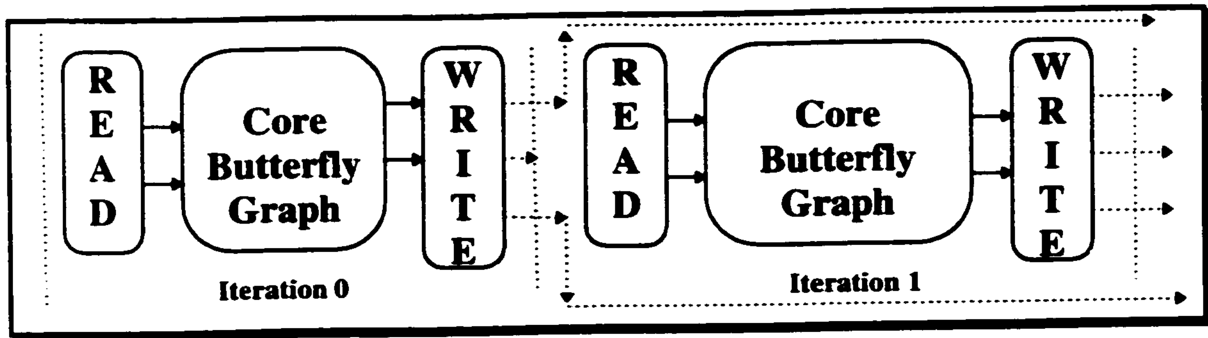


Figure 8. Sequence of operations in execution of the graph

The write operations of each iteration can be done with the read operations of the next iteration assuming there is no conflict in the memory organization; i.e., there can be simultaneous reads and writes, and also the variable produced is only consumed at least two iterations apart. If the variable is going to be used in the next iteration it can be fed back to the graph with a single delay or a recursive edge instead of being stored on the external memory to the data-path.

If the inner core graph cycle time is comparably longer than the cycle times of read and write operations, the graph computation of the third iteration can also be done in parallel with the read of the second, and write of the first iteration. The resulting order of operations can be seen in Figure 9.

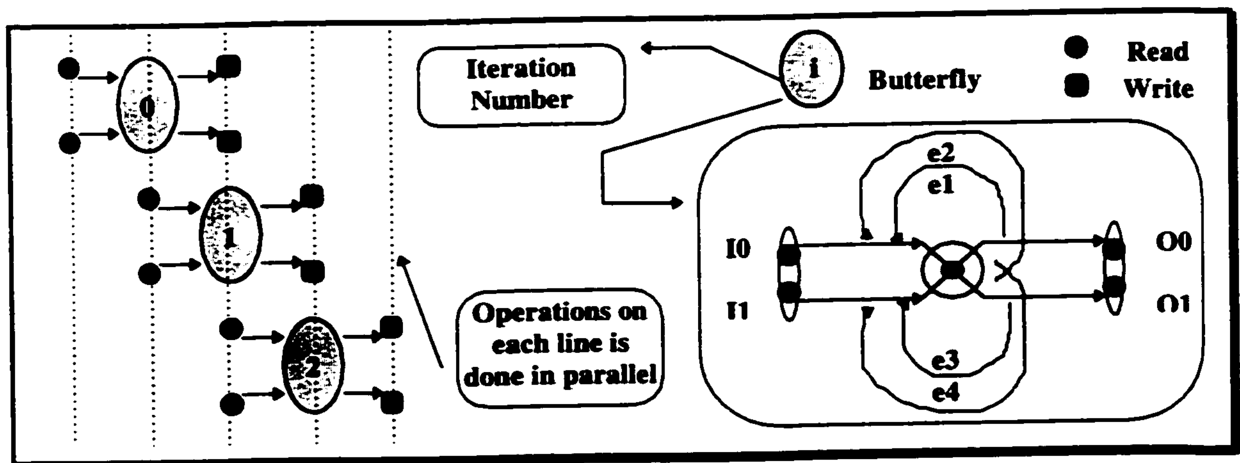


Figure 9. Sequence of operations, showing the parallelism achieved.

3.5. Extracting the Maximum Pipeline Level

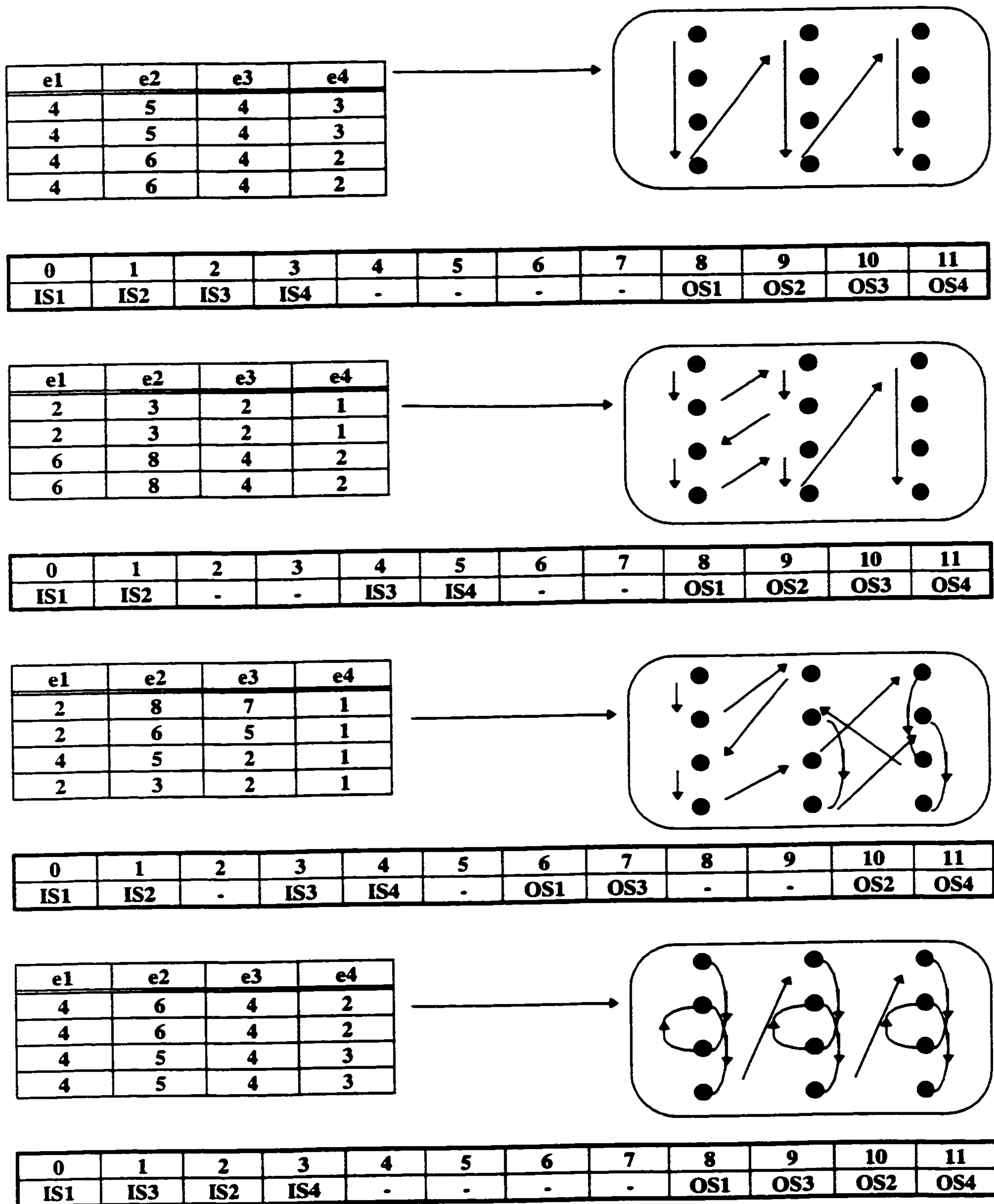
Now the weights of each edge of the folded graph are extracted. Weights are the delays that should be put at the output of one iteration so that the correct value is passed to the iteration that needs this value. For example, if a value is produced at iteration 5 and is used at iteration 9, there should be a delay of 3 (Z^{-3}) at the output or an edge with weight 3. Figure 10 shows different order of operations needed to compute the FFT in Figure 6. In all cases the precedence of operations should be preserved to guarantee the correct computation. These different orderings result in different number of delays needed for each variable, in another word, there will be less number of memory locations needed to keep the variables in between the iterations depending on this order.

With the labeling of the edges and inputs of the butterfly in Figure 9, the following tables tabulate the number of Z^{-1}s needed on each edge based on different computation order. Basically, if the delay is more than one, the variable is stored in memory, otherwise it is saved in a register that is represented by a recursive edge. These registers allow further pipelining of the core graph and data-path, thus reducing the cycle time.

I0	I1	Input Set no.	O0	O1	Output Set no.
x[0]	x[4]	IS1	X[0]	X[4]	OS1
x[2]	x[6]	IS2	X[1]	X[5]	OS2
x[1]	x[5]	IS3	X[2]	X[6]	OS3
x[3]	x[7]	IS4	X[3]	X[7]	OS4

Table 1. The set of input/output connections to the inputs & outputs the butterfly

First delays for column-wise scan is extracted, and then the same is done for other types of scan.



e1	e2	e3	e4
4	5	4	3
4	5	4	3
4	6	4	2
4	6	4	2

0	1	2	3	4	5	6	7	8	9	10	11
IS1	IS2	IS3	IS4	-	-	-	-	OS1	OS2	OS3	OS4

e1	e2	e3	e4
2	3	2	1
2	3	2	1
6	8	4	2
6	8	4	2

0	1	2	3	4	5	6	7	8	9	10	11
IS1	IS2	-	-	IS3	IS4	-	-	OS1	OS2	OS3	OS4

e1	e2	e3	e4
2	8	7	1
2	6	5	1
4	5	2	1
2	3	2	1

0	1	2	3	4	5	6	7	8	9	10	11
IS1	IS2	-	IS3	IS4	-	OS1	OS3	-	-	OS2	OS4

e1	e2	e3	e4
4	6	4	2
4	6	4	2
4	5	4	3
4	5	4	3

0	1	2	3	4	5	6	7	8	9	10	11
IS1	IS3	IS2	IS4	-	-	-	-	OS1	OS3	OS2	OS4

Figure 10. Different scan orders for the sample FFT graph

By studying these scan orders it is possible to further pipeline the data-path. If the number of Z^{-1}s are more than one in all iterations, the number of Z^{-1}s can be reduced by one and that Z^{-1} be moved inside the data-path to use it as pipeline register. This will drastically decrease the cycle time of the core data-path and the throughput of the system is increased by paralleling more operations.

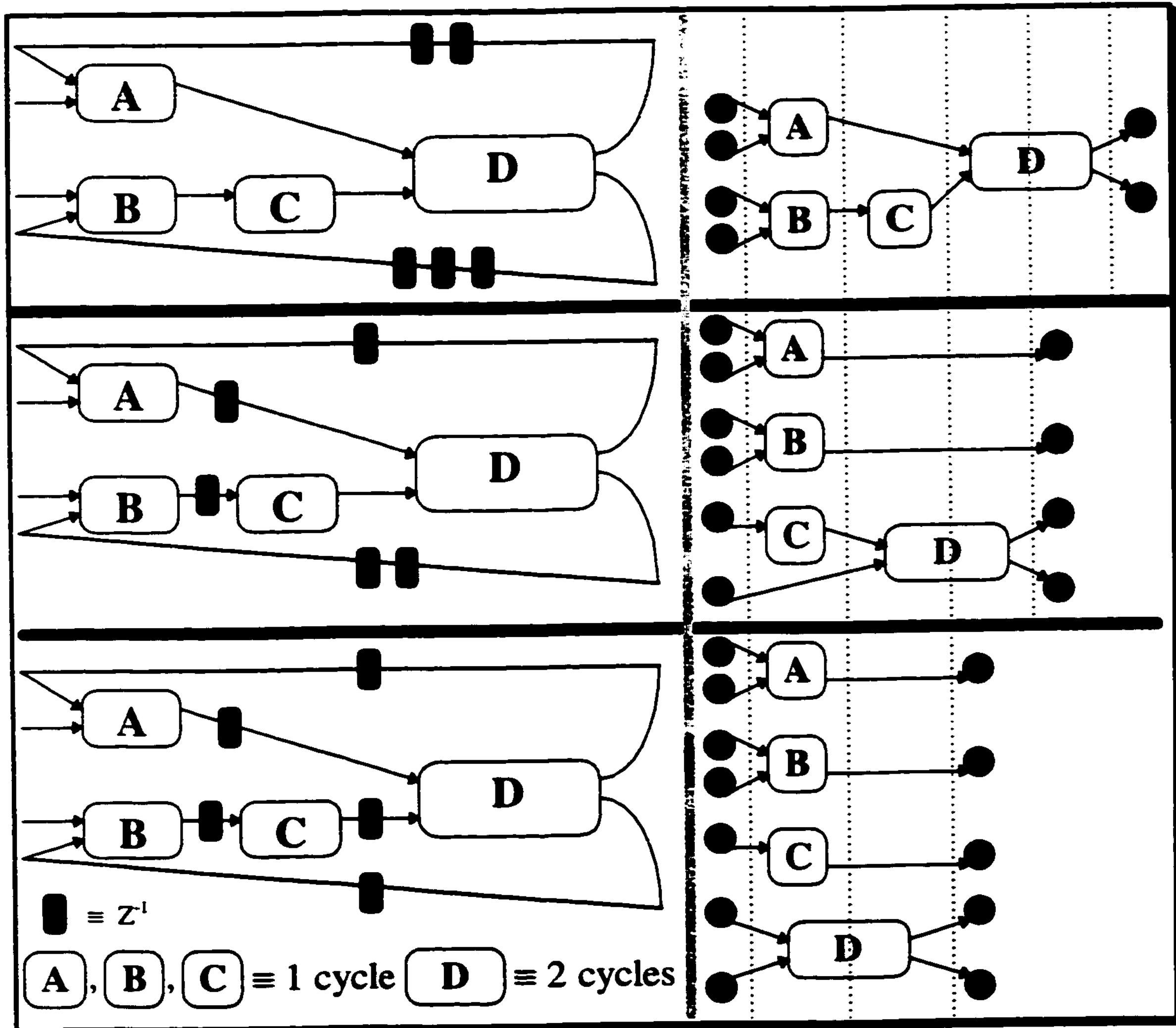


Figure 11. Retiming and pipelining illustrated

Now consider the data-path core shown in Figure 11 with two of the recursive edges with weights 2 and 3; shown as black-filled boxes and also assume that it is possible to move in as many Z^{-1}s. By moving the Z^{-1}s inside the data-path, the execution time of the

core is decreased. But it is not possible to move all of the Z^{-1}s in, otherwise the iterations' interdependency will be lost and the correct algorithm would not be implemented. This is because this data path is derived by folding the original signal flow graph. The only way to preserve the algorithm correctness is only to move in one less than the minimum number of Z^{-1}s at each iteration; i.e., to use the minimum of the weights at each column.

3.6. Scheduling the Graph

Next the FFT example is investigated and the reads and the writes to the external memory for the first type of scan (column-wise) shown in Figure 10 is extracted. With the previous discussions in mind, only one Z^{-1} can be moved from each edge in and used as pipeline register inside the data path. Having done this, the operations (reads and writes as in Figure 12) can be scheduled. It is assumed that, it is possible to have two simultaneous reads and two simultaneous writes; either by having a dual-port memory architecture or by having two memory subsystems.

This further reduces the cycle time of the execution of the whole algorithm. But if this is much too expensive, the reads and writes could be scheduled sequentially one after another.

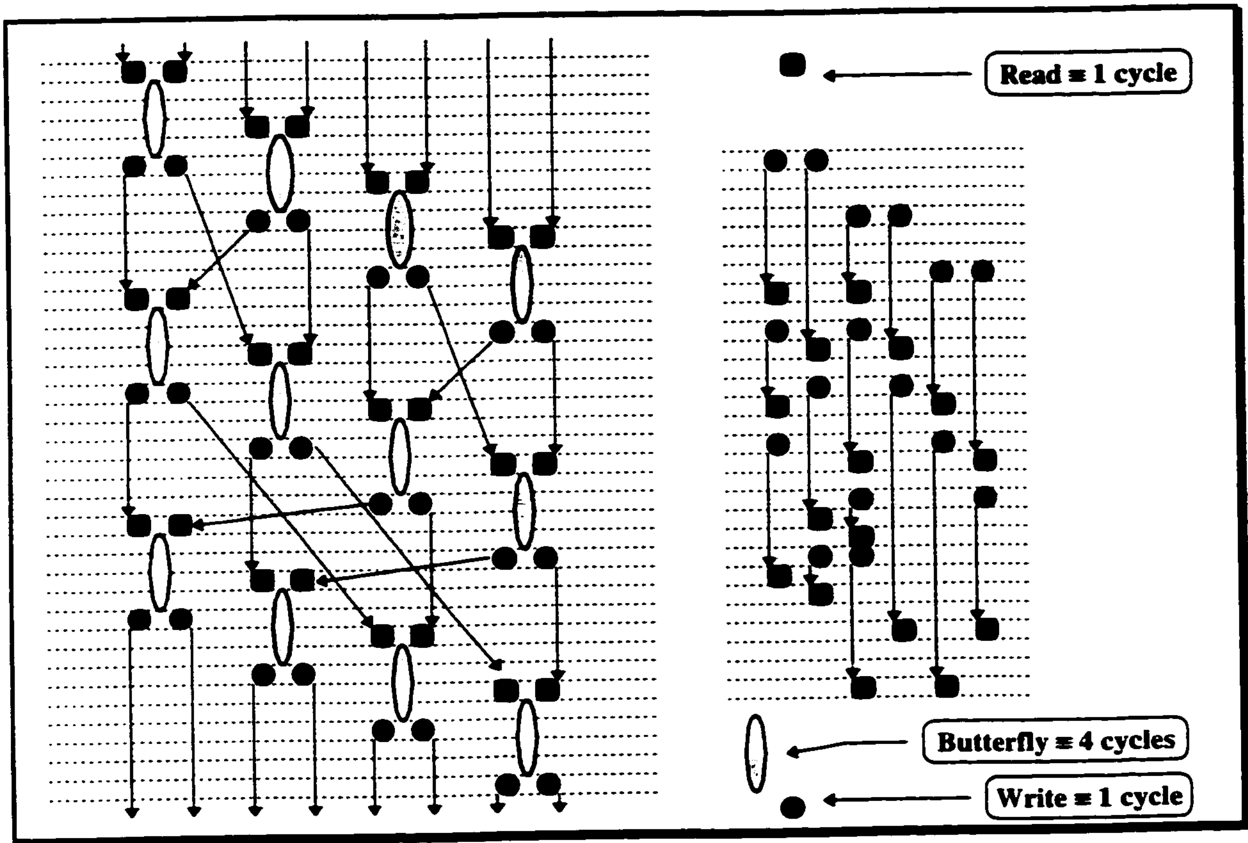


Figure 12. Scheduled FFT with two-stage pipelined butterfly core and variable lifetimes

From this discussion, it can be seen that pipelining the core shortens the total execution cycle of the algorithm. Higher levels of pipelining are also possible by introducing what is called a no-op node to the graph on edges that have the least number of Z^{-1}s. By introducing new nodes into the graph, the number of Z^{-1}s that can be moved inside the core to be used as pipelined registers could be increased. Higher levels of pipelining allow to remove the dependency among input, output operations and also the core. This simplifies the task of memory bank assignment because it is no longer needed to know the schedule of the operations in the graph and all the operations, including read and write to the memory, having the mobility of the whole execution cycle.

Chapter 4

4. Memory Bank Assignment

In this chapter, explanation is given on how to assign a memory bank number to the edges of the graph so that the memory bank usage is balanced and also on how to simplify and reduce the logic needed in the controller of the final architecture. The example graph used in this chapter is a radix-4 16-point FFT (refer to [45] for detailed explanation of FFT), assuming having only two memory banks. The graph is shown in Figure 13.

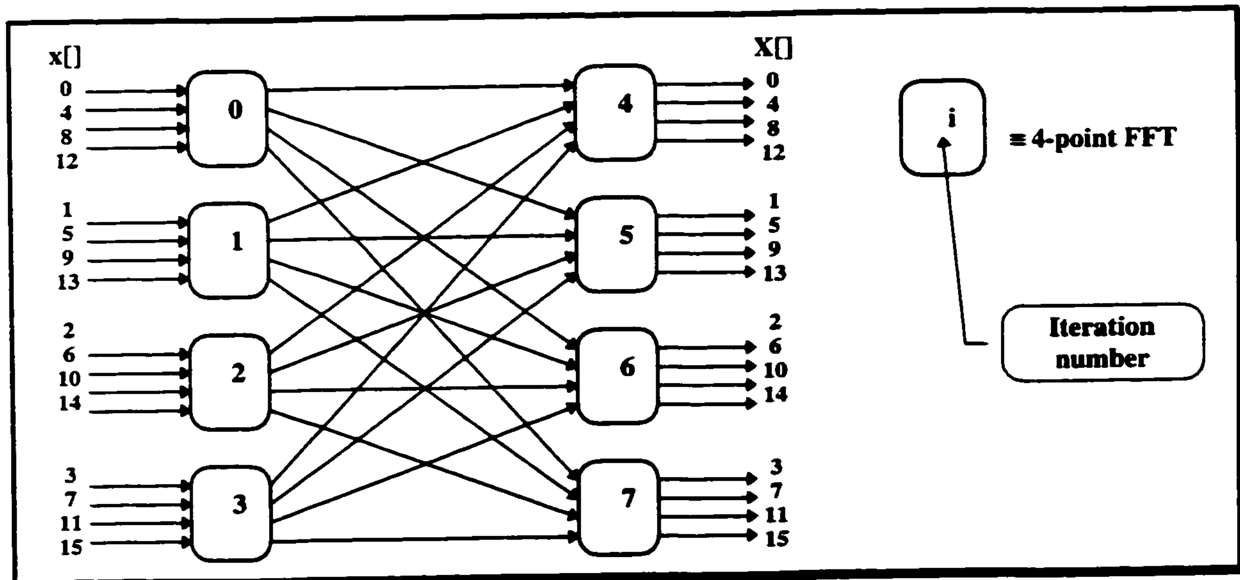


Figure 13. Radix-4 16-point FFT

An in-place storage scheme is assumed; i.e., the final result of the FFT is assumed to be stored in the same place as the original input data, but the difference with the in-place storage is that the intermediate results will not be stored in the same place as the

input data. Therefore, in the graph of Figure 13 the output edges are assumed to be wrapped around and connected to the corresponding inputs of the graph. The task is divided in two parts. One is the resource balancing, which in this case is the balancing of the memory banks usage. The second is to simplify the controller that is going to be mapped into a single or multiple FPGA system along with the data path itself. One way to simplify the controller is to reduce the number of control words used in the controller. In most of the signal processing algorithms, especially those with large storage needs and image processing applications, one can find a regularity in the usage of memory. If one can exploit and take advantage of this regularity in the access of the memory banks, the controller words that address the memory could be reduced substantially.

4.1. Exhaustive Search of the Solution Space

In the first attempt in the memory bank assignments, an exhaustive search routine was developed to do these two tasks at the same time. The assumption is that there are two memory banks and a memory bank should be assigned to each edge in the graph. With two memory banks, a binary variable is used to distinguish between the two; i.e., a '0' means the first memory bank and a '1' means the second memory bank. There are 32 edges in the graph and they are numbered from 0 to 31, and a 32-bit variable is used for the assignment of all the edges and each bit in this number represents an edge in the graph. For example a value of 0x33CC33CC means edge_0 is assigned to bank_0, edge_1 to bank_0, edge_2 to bank_1, and so on. It is assumed that the final architecture will have one processing core for the 4-point FFT, four inputs and four outputs. The binary number assigned to the edges of the graph make a 4-bit binary number at the input, and a 4-bit binary number at the output of this 4-point FFT at each iteration of the algorithm. This is

called a symbol, a write symbol for the output number and a read symbol for the input number.

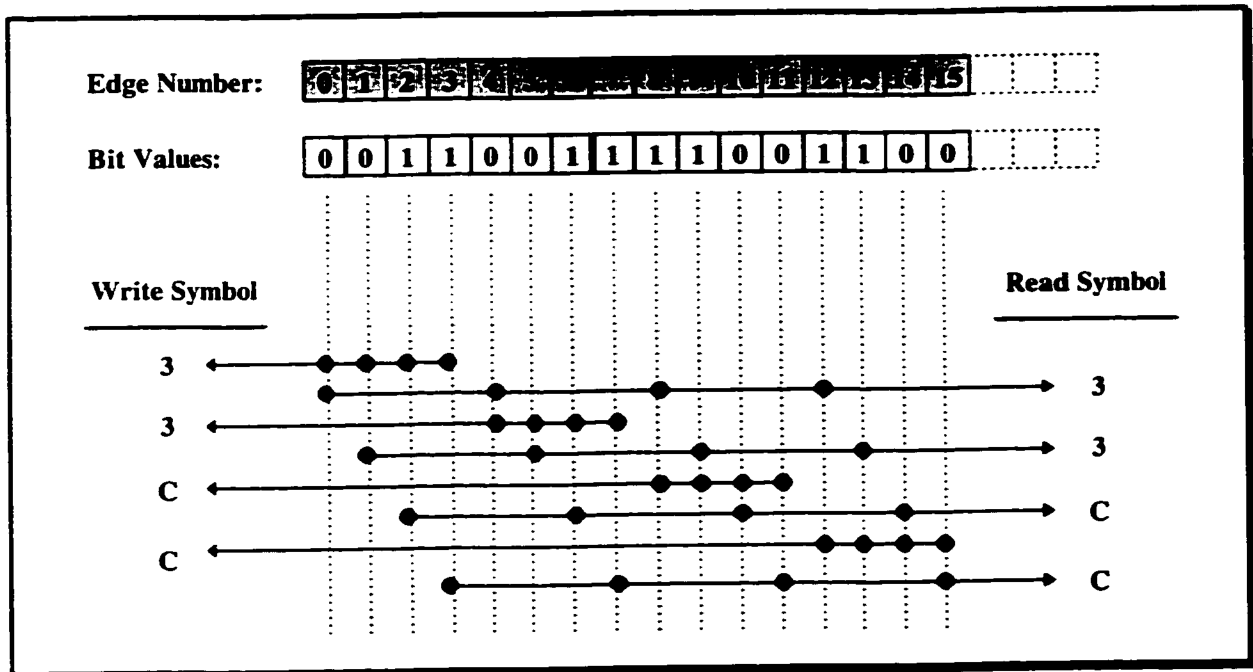


Figure 14. Pictorial representation of a symbol

The algorithm tries to assign symbols (in this case from 0x0000 to 0x1111) to the reads and the writes of each iteration, so that, first the symbol assigned is balanced in the number of 0s and 1s it has (this balances the memory bank usage) and second, the number of symbols for reads and those for the writes are minimized. The cost function used is:

$$(number_of_read_symbols) + (number_of_write_symbols) + \sum_{i=0}^{Total_number_of_symbols} (Count_of_symbol)i * (Cost_of_symbol)i$$

The exhaustive search starts counting from 0x00000000 to 0xFFFFFFFF and at each step checks the cost function and accepts the assignment only if the current cost is less than the pervious calculated cost.

4.1.1. Implementation Details

The algorithm is implemented in C and is given in Appendix A. At the beginning of the program, two arrays, a source edge and a destination edge with size of the number of edges in the graph, are declared and initialized with the node number that the edge connects to. Another array is initialized with the input and output edges that connect to a node. Two other structures are declared for an edge and a node. The edge has three fields, source node number, destination node number and the bank number assigned to it. A node has two arrays of input edge numbers and output edge numbers.

A *symbol* is defined to have a cost, a count of how many times it has been used and whether it has been used or not. Because each node has four input and four outputs, there are sixteen possible symbols whose costs are defined in the *symbol_costs* array. In the symbol's binary representation, if the number of ones and zeros are balanced (two each), the symbol cost is 0. If there are 3 ones/zeros and 1 zero/one in the symbol, the symbol cost is 1 and if there are 4 ones/zeros in the symbol, the symbol cost is 2.

There are 32 edges in the sample graph and a 32-bit number is used to represent all the memory assignments for the edges. A zero means that bank '0' is assigned to that edge and a '1' means that bank 1 is assigned to that edge. The algorithm starts by initializing the edges and nodes of the graph and then initializes the symbol table. Then the exhaustive search begins that counts from 0x00000000 to 0xFFFFFFFF and at each iteration checks the current cost. If the current cost is less than the latest calculated cost, the program reports the last cost, the current cost, the number of different words used and the current assignment.

4.1.2. Results from the Exhaustive Search

This exhaustive search was very slow and time-consuming, so a new technique based on the integer linear programming (ILP) formulation and using the GAMS solver was used and will be shown later. The results of the assignments for radix-4, 16-point FFT and two memory banks summarized in the following table.

Read Symbols (Hex)	Write Symbols (Hex)	Cost of Read symbols	Cost of Write Symbols	Total Cost
3, C	3, C	0	0	4

Table 2. Results for radix-4, 16-point FFT and two memory banks (exhaustive search).

4.2. Formulating the Problem in ILP

By formulating the problem in ILP, the search space is basically limited from all the infeasible solutions to some that may be a solution but not necessarily the best one. Search space is all the possible assignments of memory banks to the edges. In the exhaustive search, there were no means to isolate those assignments that will cost too much, long before checking all the assignments. The checking routine had also too much overhead. Once a formulation is derived, the ILP solver does a branch and bound through the bounded search space and generates a cost. The constraints written, try to minimize this cost and arrive at an optimal solution. Depending on how the constraints are written, the solver may reach the absolute best or a local optimum answer.

Now a detailed explanation of this formulation is given. In this formulation, four static sets are used. I is the set of iteration indices or the nodes that are executed at each step, in this case from 0 to 7. S is the set of symbols or the different assignments to the edges, in this case from 0 to 15. E is the set of edges, the edges are numbered from 0 to 31. The inner edges are numbered first from the output of node 0. And B is the input or

output number, a number is assigned to each input port or output port to the core; i.e., 0 to first input, 1 for the second input and so on. The same thing is true for the outputs. So in this case B is from 0 to 3, because there are four inputs and four outputs.

There are sets that define the edges of the graph using the writer's iteration number (WI), reader's iteration number (RI), writer's output (bit) number (BW), and the reader's input (bit) number (BR). There is a dynamic set called EDGE_EXTS(E, I, J, BI, BJ) that has a member for each edge defined in the graph, this dynamic set is used in the constraints. Two binary variables W_X(I, S) and R_X(I, S) are defined. Every '1' assigned to W_X means symbol 'S' is assigned to the write at iteration 'I', and a '1' assigned to R_X means symbol 'S' is assigned to the read at iteration 'I'.

As can be seen in Table 3 and Table 4, only one symbol can be assigned to each iteration whether it be a read operation or a write operation. From this, the first two constraints can be written, as will be seen later (constraints 1 and 2).

W_X(I, S)	I \ S	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0						1										
	1							1									
	2												1				
	3			1													
	4								1								
	5					1											
	6							1									
	7	1															

Table 3. Sample assignments of symbols to iterations and nodes' outputs

R_X(I, S)

I\S	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0				1												
1																1
2											1					
3			1													
4										1						
5														1		
6						1										
7					1											

Table 4. Sample assignments of symbols to iterations and nodes' inputs

The input and output symbols that are used are reflected in the W_SYM and R_SYM binary variables as a '1' (constraints 3a, 3b, 4a and 4b). If the symbol is not assigned (never used), the associated W_SYM or R_SYM will be '0'. Using these two variables, the total number of read and write symbols used (variables W_SYMS and R_SYSMS), that contribute to the final cost function (constraints 5 and 6) can be counted. Assigned to each symbol is a corresponding cost due to its distance from the average of a balanced memory access; i.e., for a two-bank memory system, writing or reading four variables into memory should send two variables to one bank and the other two to the other bank. One simplification to the problem is made by considering the nature of an FFT algorithm. It is known that one always does read or write complex variables having a real part and an imaginary part. Because these two parts are always read and written at the same time, they can be overlapped or merged, assuming only one single variable is read or written. This reduces the size of the symbols used.

The cost of every symbol is calculated and set as a constant parameter array, called SYM_COST. Constraints 7 and 8 compute the total cost of write (W_COST) and read symbols (R_COST).

1 2	$$\begin{cases} \sum_S W_X(I,S) = 1; \forall I \\ \sum_S R_X(I,S) = 1; \forall I \end{cases}$$
3a 3b 4a 4b	$$\begin{cases} I_{MAX} * W_SYM(S) - \sum_I W_X(I,S) \geq 0; \forall S \\ \sum_I W_X(I,S) - W_SYM(S) \geq 0; \forall S \\ I_{MAX} * R_SYM(S) - \sum_I R_X(I,S) \geq 0; \forall S \\ \sum_I R_X(I,S) - R_SYM(S) \geq 0; \forall S \end{cases}$$
5 6	$$\begin{cases} W_SYMS = \sum_S W_SYM(S) \\ R_SYMS = \sum_S R_SYM(S) \end{cases}$$
7 8	$$\begin{cases} W_COST = \sum_I \sum_S W_X(I,S) * SYM_COST(S) \\ R_COST = \sum_I \sum_S R_X(I,S) * SYM_COST(S) \end{cases}$$
9(x)	$$\sum_S [W_X(I,S) * BANK_IS_x(S,BI)] = \sum_S [R_X(J,S) * BANK_IS_x(S,BJ)]$$ $$; \forall E, I \to J, BI \to BJ, x = TotalBanks - 1$$
10	$$Cost = (W_SYMS + W_COST) + (R_SYMS + R_COST)$$

Table 5. Constraints used for the 16-point FFT memory bank assignment.

To summarize, constraints 1 and 2 force the assignment of at most one write or read symbol at each iteration. Constraints 3 to 8 count the total number of symbols and calculate the cost associated with them. Constraint 9, which is written for every edge, forces the source and destination of an edge to be assigned to the same memory bank. The number of constraints of the form of constraint 9 is one less than the number of memory banks used, in the case of two memory banks, one is enough. For more memory banks this constraint repeats with the difference that BANK_IS_1 is replaced by BANK_IS_2,

BANK_IS_3 and so on. These Boolean type variables are true ('1') wherever the corresponding bank in the symbol's digit is one, two, and so on.

To calculate the BANK_IS_x(S, B), there is a constant table called BITS(S,B) of the symbol S in decimal and its equivalent value in base *TotalBanks*. This is because a number is assigned to each edge that is from 0 to *TotalBanks-1*, which are digits of a number in base *TotalBanks*. Table 6 shows how the constant table of BITS(S, B) helps compute the Boolean BANK_IS_x(S, B).

BITS(S, B)

S \ B	3 (3^3)	2 (3^2)	1 (3^1)	0 (3^0)
0	0	0	0	0
1	0	0	0	1
2	0	0	0	2
3	0	0	1	0
4	0	0	1	1
5	0	0	1	2
6	0	0	2	0
...	...	...	...	...
...	...	...	...	...
27	1	0	0	0
28	1	0	0	1
29	1	0	0	2
...	...	...	...	...
...	...	...	...	...
79	2	2	2	1
80	2	2	2	2

BANK_IS_1(5, 1) = TRUE
BANK_IS_2(5, 0) = TRUE

BANK_IS_1(6, 1) = FALSE
BANK_IS_1(6, 2) = FALSE

Table 6. Base TotalBanks equivalent of symbols in16-point FFT and 3 memory banks

The total cost is calculated in the formula number 10, and it is the sum of total number of write symbols, total number of read symbols, total cost of write symbols and total cost of read symbols. This value should be minimized and this is the objective function. The ILP solver tries to minimize this and give the best assignment.

The constraints let the solver to reach an answer if it exists. Adding more constraints makes the solver arrive at an optimal answer in much less amount of time.

4.2.1. Automatic Generation of the ILP Source for Arbitrary FFT

A C program has been written that generates the ILP source file for an FFT with arbitrary number of points and radix. The input to the program is the number of points in the FFT, the FFT's radix and the number of memory banks.

The program is very helpful when dealing with higher number of points. The first part of the ILP program is very similar to the exhaustive search algorithm. The graph needs to be constructed with all the edges and nodes in it. The symbol table and their associated costs should also be constructed. The program makes writing the ILP program easier by generating all the source and destination edges and all the necessary data needed for the ILP formulation.

Code generators are very popular in software design so it is in the Electronics Design Automation. One can write a program to generate another program for another compiler or design tool. This C program did take the hassle off specifying the graph edges and data in ILP.

4.2.2. Results from the ILP Formulation

The results of the assignments for radix-4, 16-point FFT and two memory banks, radix-4 16-point FFT and three memory banks, radix-8, 64-point FFT and two memory banks are summarized in the following tables.

Read Symbols (Hex)	Write Symbols (Hex)	Cost of Read symbols	Cost of Write Symbols	Total Cost
3, C	3, C	0	0	4

Table 7. Results for radix-4, 16-point FFT and two memory banks.

Read Symbols (Hex)	Write Symbols (Hex)	Cost of Read symbols	Cost of Write Symbols	Total Cost
4, 5, 7, A, C, 1D, 33, 3B	4, 5, 7, A, 1E, 30, 37, 40	0	0	16

Table 8. Results for radix-4, 16-point FFT and three memory banks.

Read Symbols (Hex)	Write Symbols (Hex)	Cost of Read symbols	Cost of Write Symbols	Total Cost
17, E8	17, E8	0	0	4

Table 9. Results for radix-8, 64-point FFT and two memory banks.

From these tables it can be seen that, the less the number of read and write symbols, the less the complexity of the address generators and control logic. It can also be seen that balancing the memory accesses may be more costly in regards to the total cost considered here.

The ILP solver reaches the solution in much less time than the exhaustive search. The ILP formulation and the method proposed are a good start at reaching an algorithmic method to assigning banks to data flows of an algorithm. Heuristics should be used to do this at first and then come up with the proper algorithm.

In the next chapter, the process of address assignments to each memory bank is explained.

Chapter 5

5. Memory Address Assignment and Generation

In this chapter, a technique to assign addresses to intermediate variables is discussed and also different techniques to build a hardware-based address generator is explored. One can find different techniques presented in the literature. Designing a flexible and efficient address generator is very difficult. The method used may also not be very useful in generating addresses for different algorithms.

5.1. Address Assignment

There has been many studies on the assignment of memory addresses to variables (register allocation) in the filed of Computer Science. A number of algorithms have been developed mostly for use with high-level language compilers. The commonly used algorithm is the graph coloring [25], [26] and how to find the minimum number of colors needed to properly color a graph. This minimal number of colors is also called the chromatic number of the graph.

Basically, coloring of a given graph $G = (V, E)$ with K colors, where V is the set of vertices, E is the set of edges in the graph and $K \leq |V|$, is to find function $f: V \rightarrow \{1, 2, \ldots, K\}$ such that $f(u) \neq f(v)$ where $\{u, v\} \in E$. It can be said that coloring of a graph is to assign a color to each of its nodes so that the nodes connected by an edge have different colors.

Register allocation is done by first creating a *register interference graph,* which is a graph that has V nodes that represent the variables and there would be an edge between two variables that are alive at the same time during the computation. These nodes are said to interfere with each other; thus the name interference graph. After this step, for a limited number of K registers, one should find a K-colorable graph.

The graph-coloring algorithm [27] belongs to the NP-complete set of problems that may result in an impractical amount of computation that is needed to find out the number of colors. For this reason and the fact that for fairly complex DSP algorithms with large number of data stored in memory, the graph coloring algorithm would be unrealistic to be used for address assignments for large number of registers, other methods should be used. The graph-coloring algorithm is mostly used in high-level language compilers for CPU architectures with small number of registers or for register assignment in synthesis of an architecture with few number of registers.

For different algorithms, one can exploit the regularity of the access and find a good address assignment. As was seen before, the access scheme could also affect the final architecture and the maximum number of pipeline levels. There is an efficient storage scheme proposed in [15] for assignment of addresses for a radix 2 FFT. A better storage scheme will be seen later that does not have the limitations of this assignment.

5.2. Address Generation

One of the challenging tasks after register allocation and memory address assignment, is the address generation. Once all the addresses of source, intermediate and

destination variables are known, one can come up with different schemes to generate those addresses. There are two schemes for generating addresses for a specific algorithm.

5.2.1. Software-based Address Generation

In general, loop constructs or dedicated constant lookup tables can be used to generate the addresses for a specific algorithm. A dedicated microcode sequencer or small microcontroller implemented in hardware could execute the program to generate the addresses. This program could be hard coded into a ROM and even for flexibility in rewritable memory. The advantage of this scheme is that the program that generates the addresses could be modified to generate a new set of addresses for implementing another algorithm. The disadvantage is that special consideration is to be made in designing a dedicated microcontroller circuit.

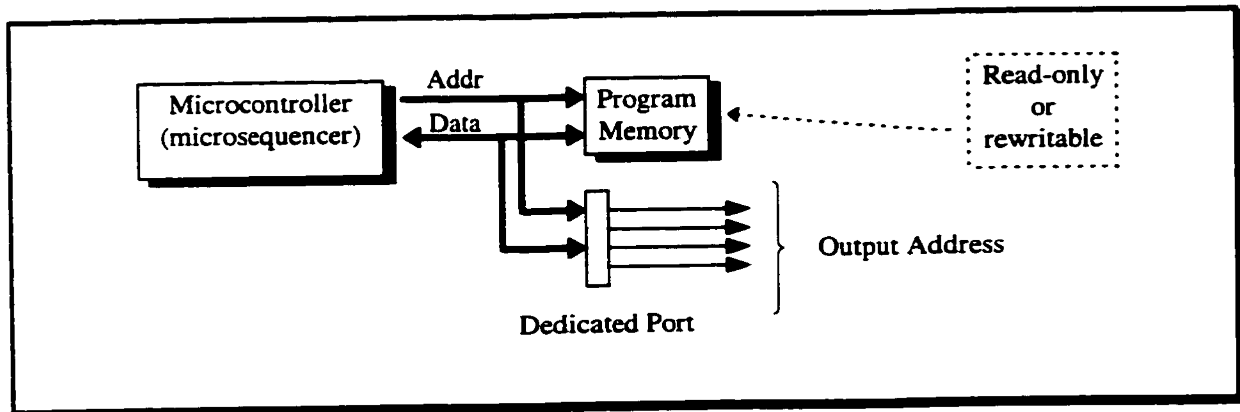


Figure 15. Software-based (microcontroller) address generator

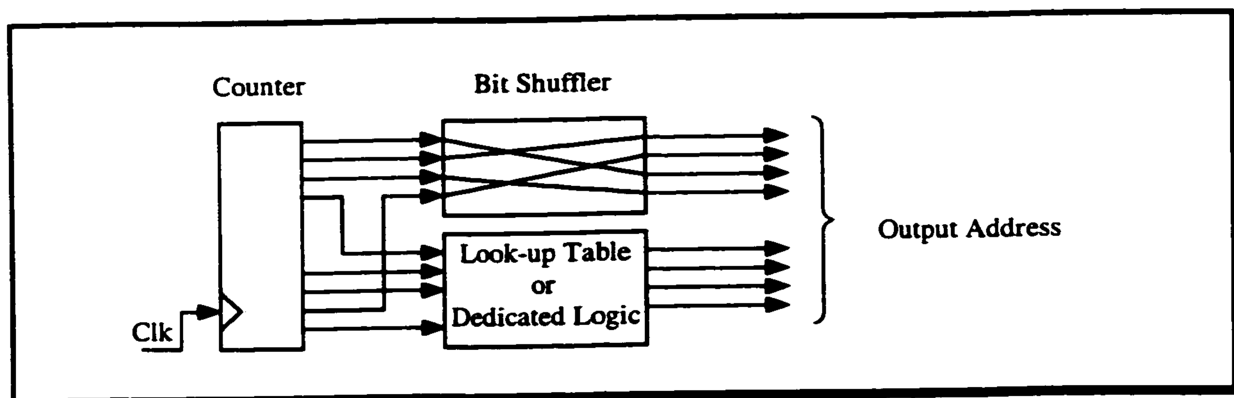
Another use of software is for analyzing the addresses and finding a regular pattern and to exploit this pattern to designing a much simpler and yet workable address generator in hardware using dedicated logic.

5.2.2. Hardware-based Address Generation

Generating addresses for a specific algorithm is very important and could become a bottleneck in execution of the algorithm. There have been many studies to come up with a scheme to generate an address of a variable in memory on the fly. For specific algorithms one can find simple methods to generating these addresses. The commonly used method is using look-up tables, which is very costly on the memory requirements and is mostly used in cases were the number of addresses are minimal.

Another method is the use of dedicated computing hardware to generate the addresses on the fly. One can construct an address generator by using counters plus additional adder/subtractor, bit-shufflers, some logic and/or look-up tables. There have been many studies in designing a GAG[19] ([28], [29], [30], [31]).

For many applications, one can exploit the access regularity of a specific algorithm, by using some transformations and changing the access order of the algorithm to take advantage of a much simpler address generators. In one study [32], by having all the addresses of the algorithm in question, one can generate them by using simple counter, bit shuffling and some logic and/or look-up table if the number of addresses is a power of 2.



[19] GAC = Generic Address Generator

Figure 16. Simple address generator

Figure 16 shows this simple address generator obtained by using this algorithm. The algorithm starts with a list of addresses (whose total number is a power of two) to generate. It then starts with the first bit of these addresses and follows these steps:

1. If the list is all zeros or all ones, the process for this bit is done and this bit is stuck and '0' or '1' whichever applies.
2. Split this list in half.
3. If the two halves are equal, go to step 2 otherwise continue.
4. If the two halves are not logical inverse of each other, the sequence is a semi-random sequence and is dealt separately. Otherwise continue.
5. If the two halves are equal, then if they are all '1' the counter bit is directly connected to the address bit, if it is '0' the counter bit is inverted and connected to the address bit.
6. If the two halves are not equal, the counter bit is ExORed with whatever bit is found by going to step 2 again.

For a semi-random sequence, basically the bits that are '1' should be decoded. The basic idea is to try to match (decode) the counter bits or a combination of them using inverters, AND, OR and XOR gates.

This algorithm has been translated to C based on the original paper [32] and is provided for use with the example design in the next chapter.

Chapter 6

6. Using the Techniques in an Example Design

In this chapter, a demonstration is made of most of the techniques discussed, to implement a 1024-point complex FFT hardware. First, the architecture to be implemented is presented, then deeper aspects of the architecture is shown, and finally different modules used and how to implement them are discussed. The design is completely done in VHDL and the results of simulation and synthesis is presented later.

6.1. Which FFT Algorithm Implementation to use?

In chapter 3, Figure 6, an implementation of the FFT algorithm called Decimation-in-frequency that is known as Cooley-Tukey implementation was seen. The FFT is break down of the DFT[20] of a finite sequence { x[n] }; $0 \le n \le N-1$ into smaller DFTs and combining them to get the final result. The DFT itself is defined as:

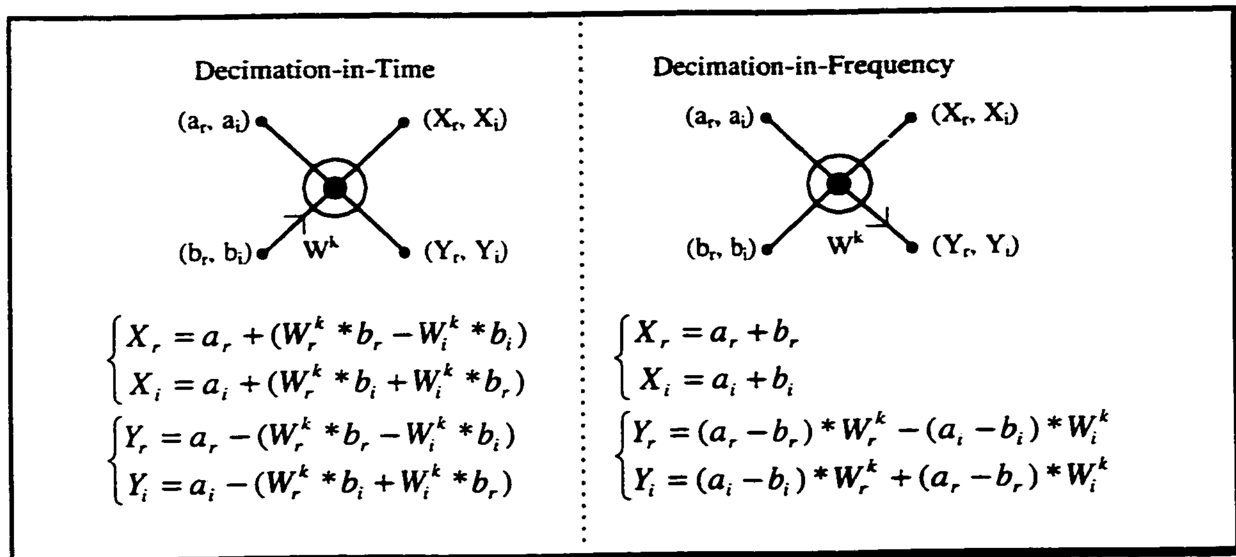
$$\begin{cases} X[k]=\sum_{n=0}^{N-1} x[n].W^{nk} ;k=0,1,\dots,N-1 \\ W^{nk}=e^{-j\left(\frac{2\Pi}{N}\right)nk} \end{cases}$$

The complexity of a DSP algorithm is determined by the number of multiplication operations to be done. The number of multiplication operations in a DFT is of $O(N^2)$ and

[20] Discrete Fourier Transform

for an FFT is of $O(N \log N)$, which makes it more suitable for implementation in hardware or software (refer to [45] for detailed explanation of DFT and FFT).

There are many ways to break down a DFT. One is called a decimation in time and the other is decimation in frequency. The two butterflies used for each of these are shown in Figure 17.



$$\begin{cases} X_r = a_r + (W_r^k * b_r - W_i^k * b_i) \\ X_i = a_i + (W_r^k * b_i + W_i^k * b_r) \end{cases} \quad \begin{cases} Y_r = a_r - (W_r^k * b_r - W_i^k * b_i) \\ Y_i = a_i - (W_r^k * b_i + W_i^k * b_r) \end{cases}$$

$$\begin{cases} X_r = a_r + b_r \\ X_i = a_i + b_i \end{cases} \quad \begin{cases} Y_r = (a_r - b_r) * W_r^k - (a_i - b_i) * W_i^k \\ Y_i = (a_i - b_i) * W_r^k + (a_r - b_r) * W_i^k \end{cases}$$

Figure 17. Decimation-in-time and decimation-in-frequency butterflies

As can be seen, the number of operations in each implementation is the same but, the decimation-in-frequency is more suitable because the multiplication is done after the additions. Usually if multiplication is done first, the results would grow in number of bits needed to represent them and because most implementations are based on fixed-point addition and multiplication, the results need to be rounded. This rounding of the results introduces error and noise in the system. So the FFT hardware based on the decimation-in-frequency FFT is selected for this demonstration.

6.2. An Efficient Architecture for a 1024-point Complex FFT

As shown in chapter 4, the optimal number of memory banks needed during the computation of a radix-2 1024-point FFT with one butterfly is two. So based on this, two distinct memory banks are needed to hold the input data, the temporary intermediate in-place results and finally for the storage of the FFT result. Because a complex FFT engine is to be implemented, twice this amount is needed to store the real and imaginary parts of each value. So the total number of memory banks needed is four.

An algorithm with a single butterfly was selected for implementation. This results in the smallest area possible for this design. If more performance is needed out of this design, more butterfly elements can be assigned that calculate more intermediate values at the same time. With careful design and scheduling, one can achieve greater performance by sacrificing more silicon area.

In chapter 3, an architecture was seen that can be used to implement most signal processing algorithms. Refining that architecture for the complex FFT, the following architecture is arrived at.

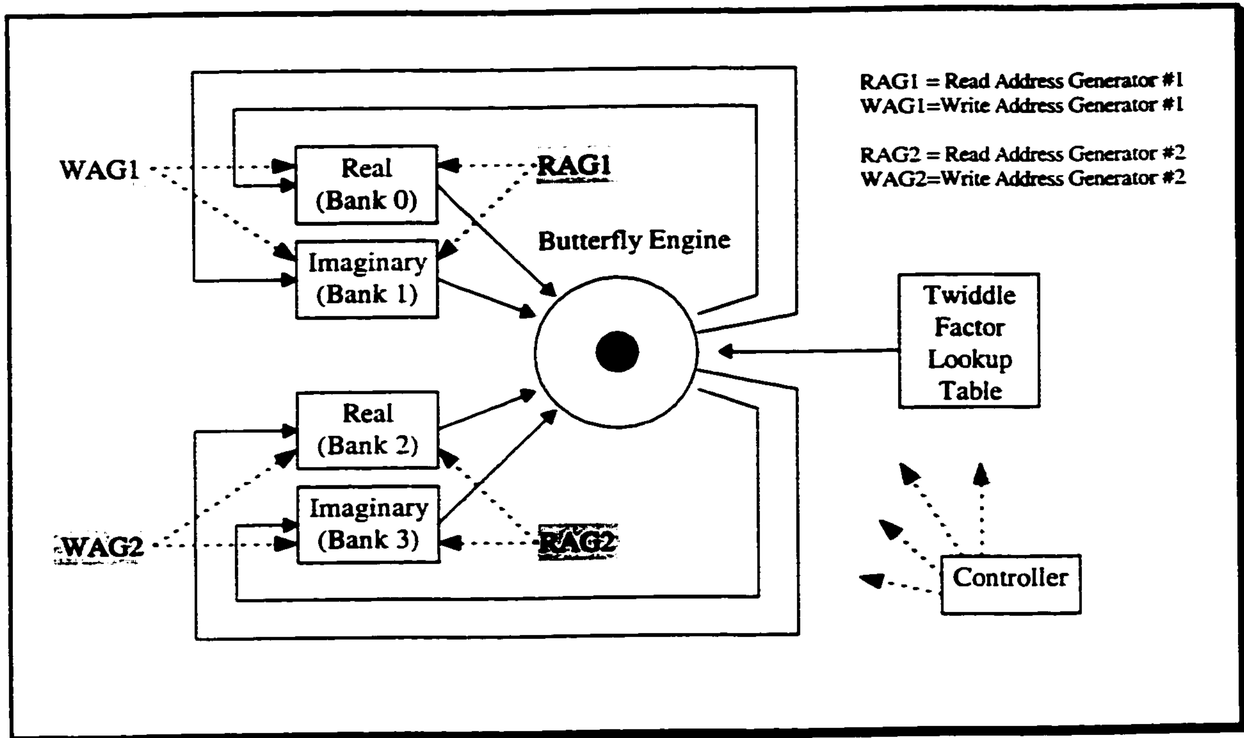


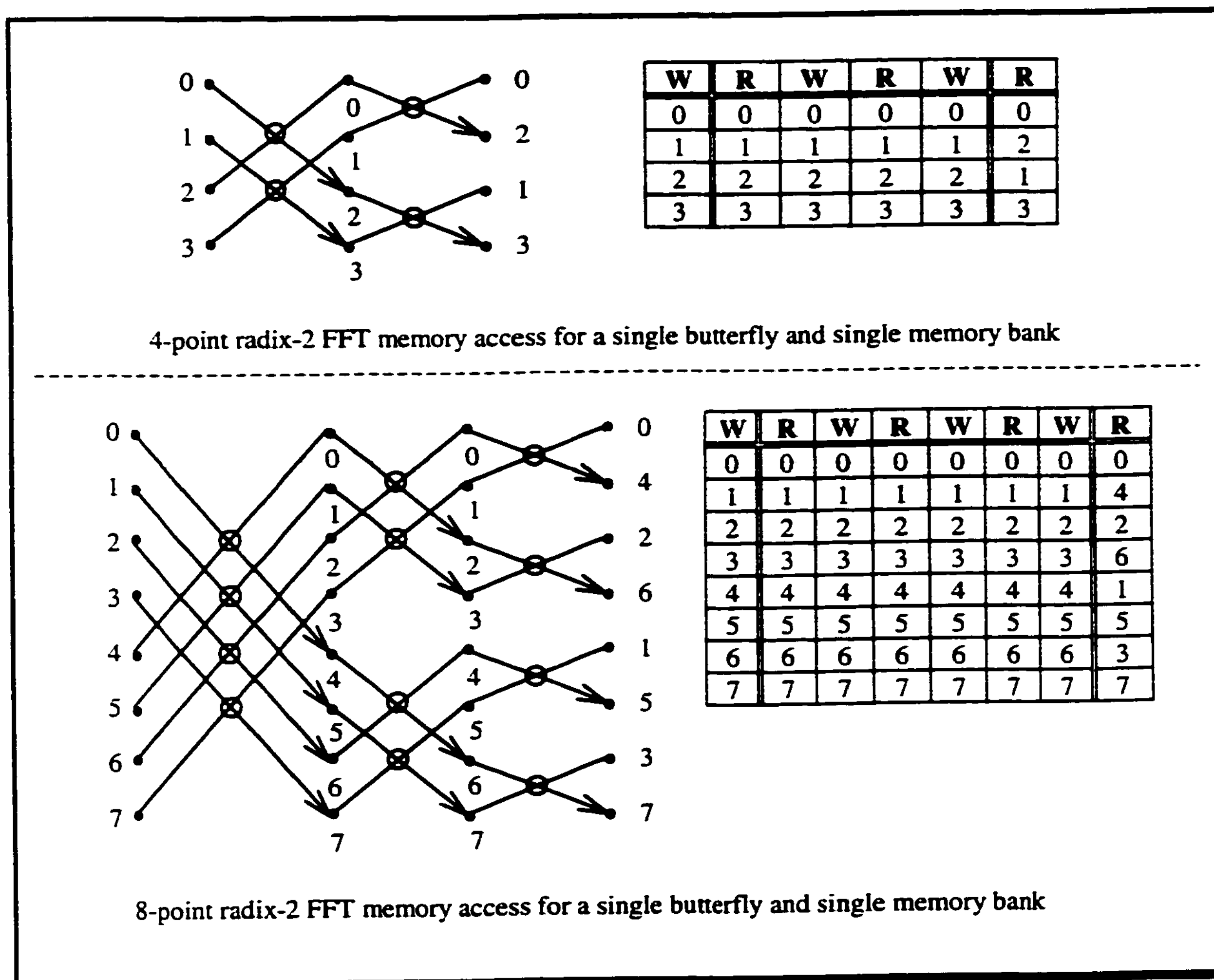
Figure 18. Proposed architecture for complex FFT

There is only one butterfly computation engine. There are also four different address generators used to address the source operands (both real and imaginary) and the destination operands (both real and imaginary). Another memory holds the twiddle factors that would be addressed with another address generator.

6.3. FFT Signal-flow Graph and Memory Access Pattern

The Cooley-Tukey implementation of an FFT, accesses the source variables in-order and stores the intermediate results in place of the source variables. The starting point is a 4-point FFT, which is increased to 32-point FFT to find a regular pattern for storage and accesses to those variables.

The arrangement of the variables of a 4-point and an 8-point decimation-in-frequency FFT and their corresponding signal flow graphs are shown in Figure 19.



W	R	W	R	W	R
0	0	0	0	0	0
1	1	1	1	1	2
2	2	2	2	2	1
3	3	3	3	3	3

4-point radix-2 FFT memory access for a single butterfly and single memory bank

W	R	W	R	W	R	W	R
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	4
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	6
4	4	4	4	4	4	4	1
5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	3
7	7	7	7	7	7	7	7

8-point radix-2 FFT memory access for a single butterfly and single memory bank

Figure 19. Cooley-Tukey FFT access patterns

As can be seen the source and all the intermediate variables are stored in increasing order from zero and the results are stored in bit-reversed order. In the corresponding signal flow graphs, the computation order is from top to bottom and from left to right. This storage and access scheme is not suitable for use with two memory banks and a single butterfly engine. The reason is that, with two storage banks, collisions should be avoided to speed up the memory access. Otherwise, when collisions or memory access conflicts occur, the memory should be accessed consecutively to retrieve/store two

source/destination variables. Now a storage and access scheme should be found that is more efficient and easy to implement in hardware.

6.4. Manipulating Memory Access Patterns

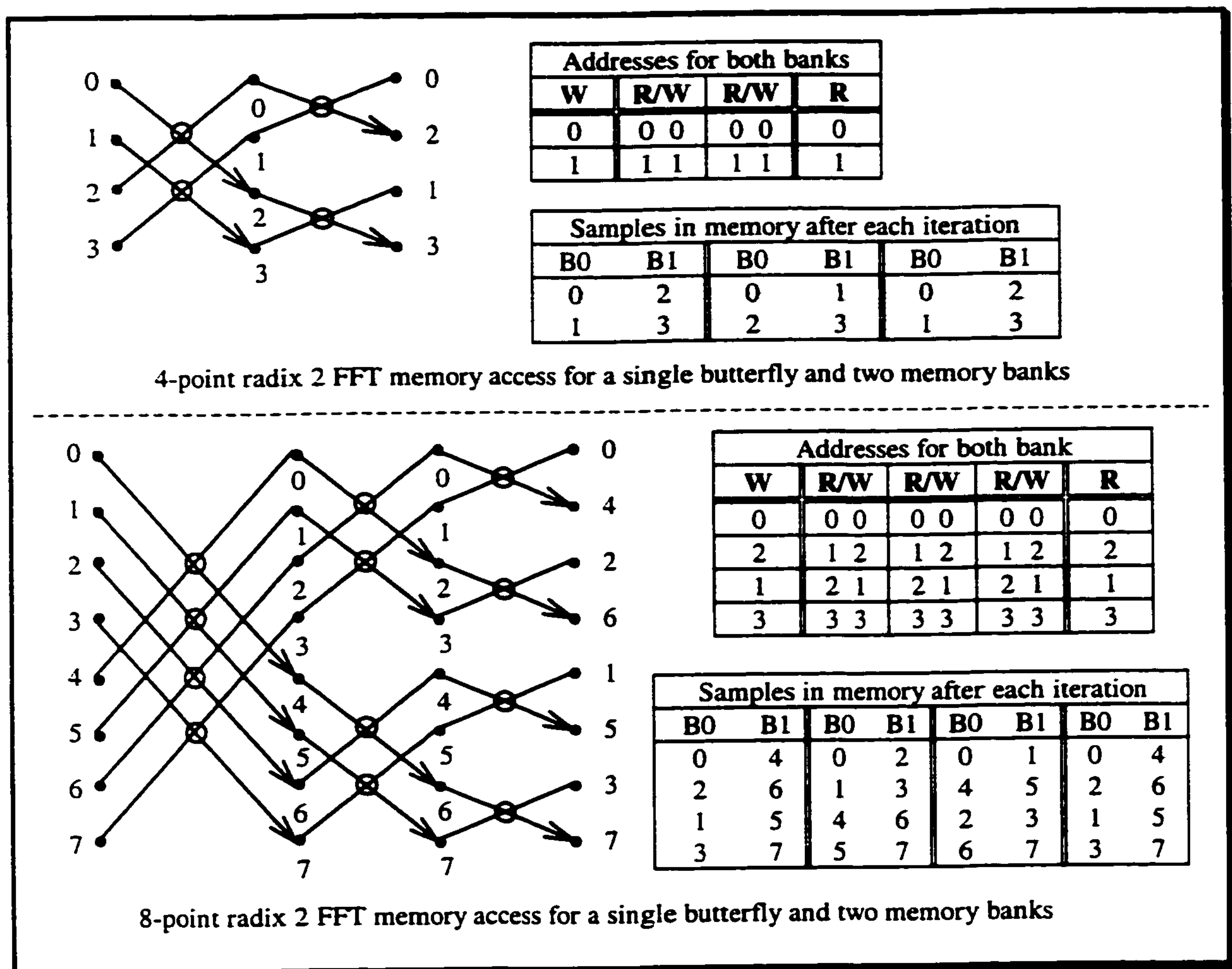
In general, to avoid collisions in multiple memory processing engines, it is best to interleave the storage of variables. By interleaving, one means storing variables accessed at consecutive points in time in different banks of memory. Interleaving does not always alleviate the memory conflicts in every algorithm and a more detailed study of a specific algorithm is needed to devise a good storage and access scheme.

In [14] and [15], an efficient way to store intermediate variables of a radix-2 FFT algorithm is proposed. In that paper, the suggested method by authors results in two different access patterns. One is a stride 1 and the other is bit-reversed. They do not show all the iterations of the computation. From the storage order they suggest, the first pass of computations is with no conflicts, but the second pass will cause some conflicts.

The storage and access schemes are refined to have zero conflicts and simple address generators. The data-path is also pipelined to achieve the fastest execution cycle. The zero conflict scheme makes sure this pipeline is not starved or stalled to get the maximum performance.

To do this, one should start from the nodes that produce the last results and start assigning addresses to those nodes keeping in mind to interleave the accesses. Then try to minimize or even remove conflicts by simple swapping using multiplexers and additional registers. This can be derived from simple observation of the access patterns. This architecture is now generalized to any number of points in the FFT as follows.

First lets look at the addresses and try to find their patterns. Figure 20 shows the addresses for a 4-point and 8-point radix-2 DIF FFT.



Addresses for both banks			
W	R/W	R/W	R
0	0 0	0 0	0
1	1 1	1 1	1

Samples in memory after each iteration					
B0	B1	B0	B1	B0	B1
0	2	0	1	0	2
1	3	2	3	1	3

Addresses for both bank				
W	R/W	R/W	R/W	R
0	0 0	0 0	0 0	0
2	1 2	1 2	1 2	2
1	2 1	2 1	2 1	1
3	3 3	3 3	3 3	3

Samples in memory after each iteration							
B0	B1	B0	B1	B0	B1	B0	B1
0	4	0	2	0	1	0	4
2	6	1	3	4	5	2	6
1	5	4	6	2	3	1	5
3	7	5	7	6	7	3	7

Figure 20. Modified accesses for 4 & 8-point Cooley-Tukey FFT (two memory banks)

From this figure and the addresses, it seems that all the writes are bit-reversed and all the reads except the last one are sequential (stride 1) and the last read is bit-reversed. To confirm this, the 16 and 32-point FFTs were tried and the same conclusion was drawn.

From these tables, it is clear that to write the results of the butterfly back to memory at the proper location, the results of two consecutive iterations need to be scheduled so that results from one iteration is sent to the same memory bank and the next

iteration to the other memory bank. The final architecture is shown in Figure 21. As can be seen, a single butterfly engine is followed by a skew buffer that routes the results to a different bank. There are four registers and two multiplexers in this skew buffer to skew the results so that they are available for writing to the address provided by the address generators based on the address assignments done in Figure 20. The details of this skew buffer is presented in the next chapter.

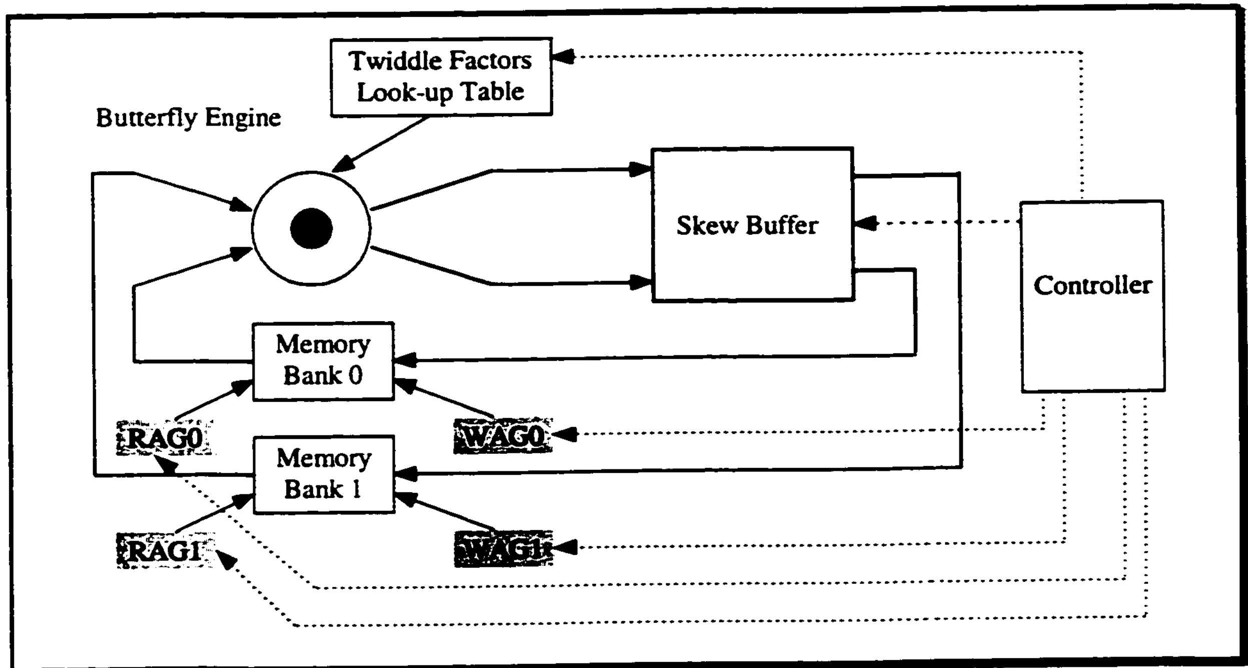


Figure 21. Final FFT architecture with skew buffer registers

The controller is responsible for orchestrating the order of operations and enabling different resisters at different cycles, controlling the multiplexer select lines, and the address generators.

Next chapter will discuss the data path and control logic in detail and present different aspects of VHDL design.

Chapter 7

7. Detailed VHDL Design

In this chapter, all the necessary steps from specification to implementation of a radix-2 1024-point Cooley-Tukey FFT engine with two memory banks is detailed. The design is completely done in VHDL and successfully fitted on a Xilinx Virtex V150PQ240-6 [33]. The synthesis is done using the Synplicity's Synplify tool and simulations are done using ModelTechnology's Modelsim VHDL simulator.

The data path design is detailed first and different tradeoffs made in the process are shown then the control logic design is explained.

7.1. Design of the Data Path and Its Elements

In most signal processing algorithms especially in Digital Signal Processing (DSP), there are many basic elements that are used to construct the data path of a system. The basic elements of a DSP system are addition, multiplication and multiply-accumulate operations. There are other operations that relate to DSP systems in general, but the ones mentioned above are the most basic and widely used in any DSP algorithm.

To improve the area/performance merit of a system, one should first do optimizations at the highest level of a design, namely: specification and architecture. After architectural optimizations, the system's building blocks or components should be improved. This improvement will, in effect, enhance the overall system operation.

The most costly operation in a DSP system is a multiplication operator. A multiplier module is both area consuming and also sluggish in the performance aspect. Therefore, multiplication is the main bottleneck in the area/performance of a data path and can change the characteristics of a system in both aspects.

Choosing the best components in general and the best multiplier for any DSP system, is the best strategy to follow for improving the system performance. In the following sections, different architectures for an adder and a multiplier, which are the basic building blocks of the FFT engine, are reviewed.

7.1.1. Addition Schemes

Adders could be categorized into the following: 1-bit adders, carry-propagate adders (CPA), carry-save adders (CSA), and multi-operand adders [34].

The 1-bit adders include Half-Adder (HA) and Full-Adder (FA). In the carry-propagate adders the carry bit to the next stage of an n-bit adder is derived from the previous stage's carry-bit and the current input bits with some additional logic. Carry-propagate adders include ripple-carry adders (RCA), carry-skip adders (CSKA), carry-select adders (CSLA), carry-increment adders (CIA), conditional-sum adders (COSA), carry-lookahead adders (CLA), and parallel-prefix adders (PPA).

The parallel-prefix adders are the most flexible ones that include a preprocessing, carry-lookahead, and postprocessing step. They can have the area and speed characteristics of all the adders mentioned above. They are basically a universal adder architecture with all the area-delay trade-offs. There are three different variations of PPAs.

They are called Kogge-Stone implementation (PPA-KS), Skansky implementation (PPA-SK) and Brent-Kung implementation (PPA-BK).

Carry-save adders (CSA) are three-operand adders that do not do any carry propagation and just save (pass) all the carry bits calculated. Multi-operand adders can be comprised of the carry-save adder stages and carry-propagate adder stages to compute the final addition. These adders can be constructed in array or tree (Wallace tree) topologies.

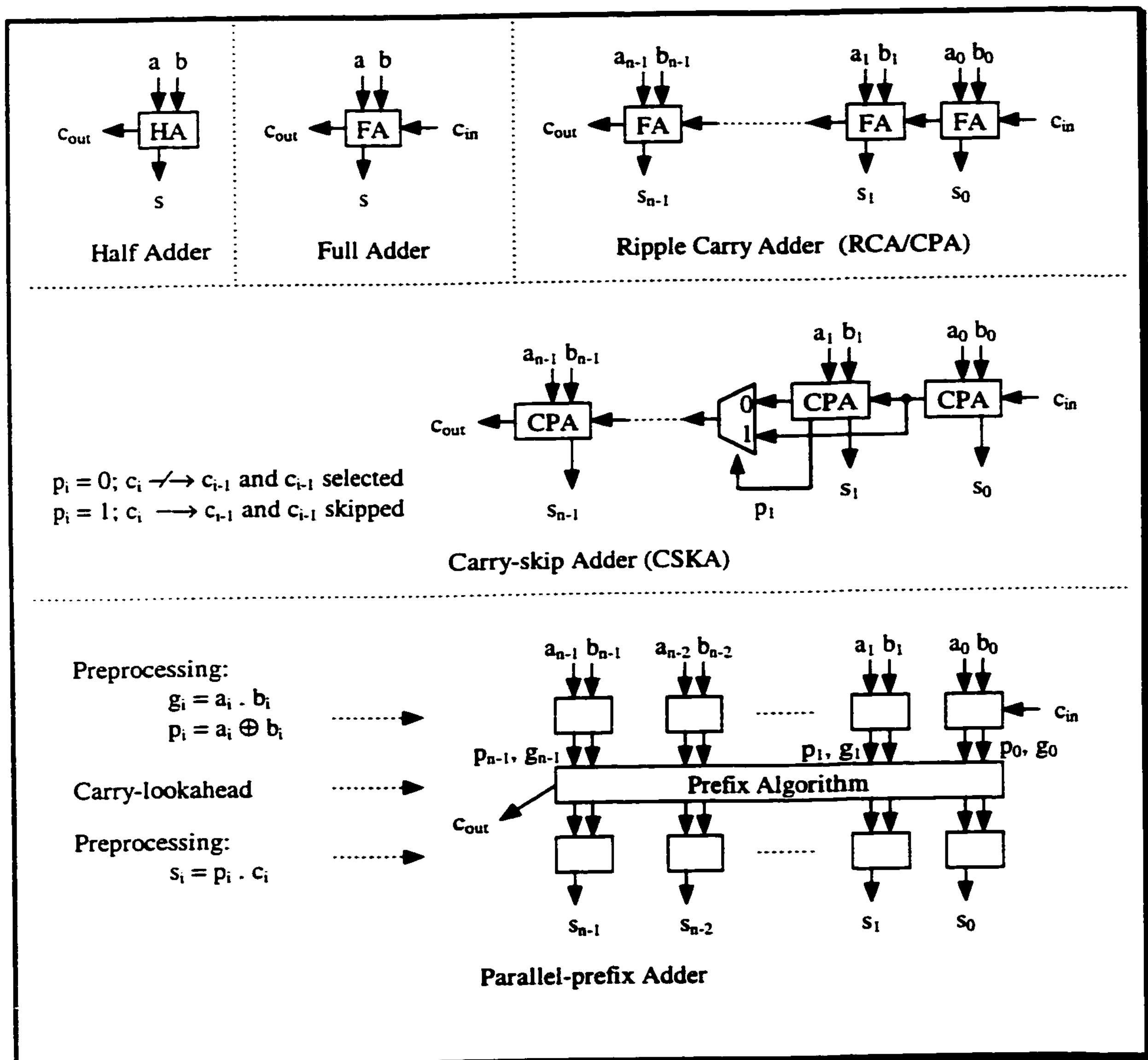


Figure 22. A few different adder structures

The ripple carry adders are almost the smallest after CSKA adders and the slowest ones, PPA-SK / PPA-KS and COSA are the fastest adders for 64-bit additions.

Exploring all these structures and choosing the optimum area/performance needed depend on the target technology that is used. For ASICs, these structures are all viable solutions and any of them can be implemented. The selection depends on the design specifications and constraints. These modules should preferably be implemented and put in a library that a high-level synthesis tool has access to. Then the area/speed selection would be the assignment part of the high-level synthesis. If the selection of the architecture is done at a higher level, the tool would also be able to insert pipeline registers to speed up the performance of the adders, yet preserve the original algorithm.

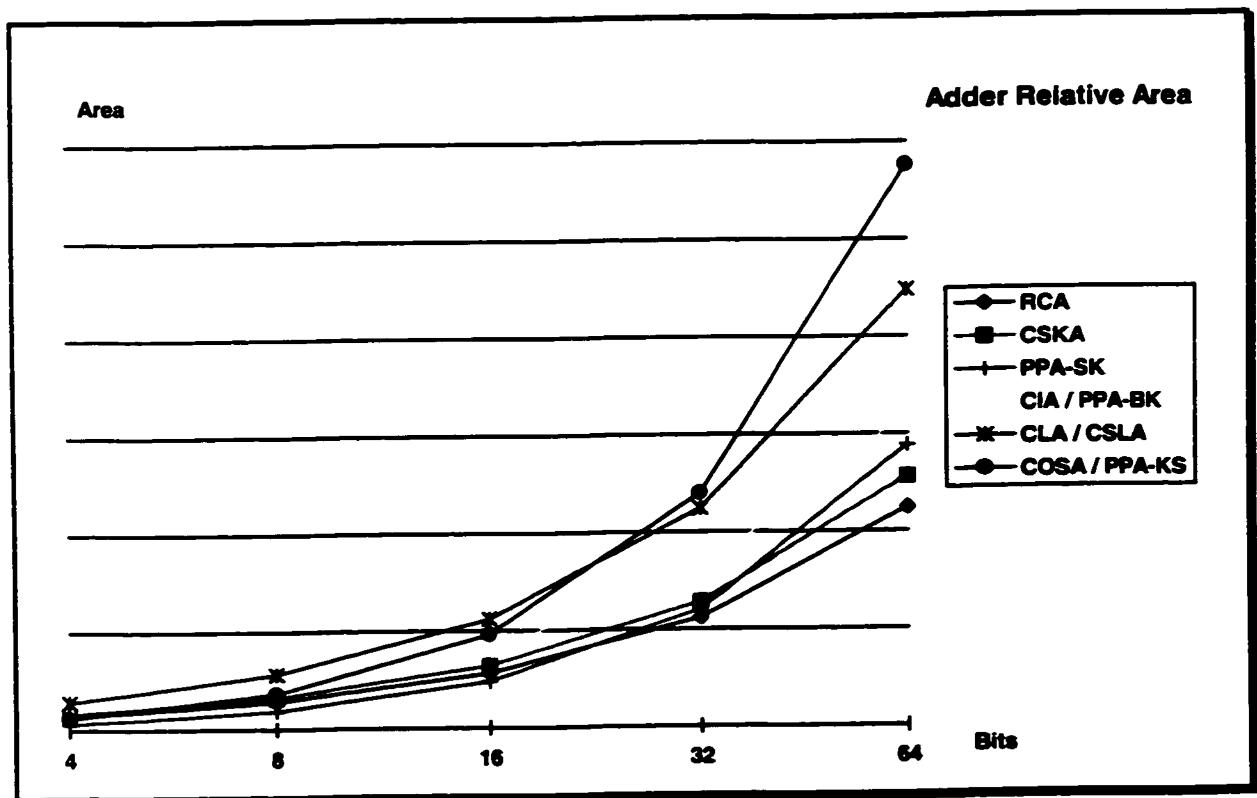


Figure 23. Areas for different adder architecture

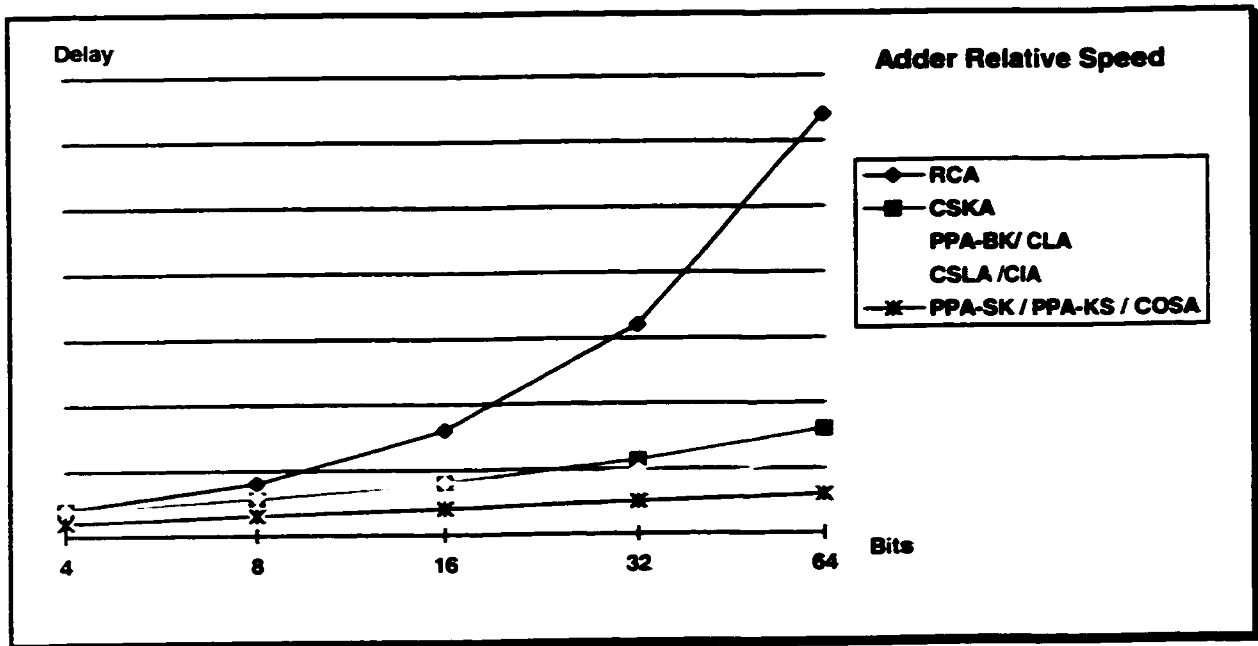


Figure 24. Speed for different adder architectures

The case is completely different for FPGAs. Almost all FPGAs have dedicated carry-logic resources to speed up the adders, subtractors, incrementers and counters. These carry chains can go up and/or down the FPGA die but not in all directions. If the regular routing had been used instead of these dedicated routes, the delay associated with arithmetic operators would be big.

All these architectures are presented to show the different trade-offs and architectures possible. One should refer to other references for complete discussion on specific algorithm.

7.1.2. Multiplication Schemes

As said before, multiplication is very costly regarding both area and speed. There are many architectures [35], [36] that help improve the speed, but at the expense of increased area.

The following are some examples of different multiplier architectures:

1. Shift and add and bit-serial multiplier
2. Booth and modified-booth algorithm
3. Wallace tree multiplier (using CAS and CLA)
4. Non-additive multiply modulus (NMM) using Wallace tree and CPAs
5. Pezaris array multiplier
6. Array (Braun) Multiplier
7. Baugh-Wooley multiplier
8. Systolic array multiplier
9. Constant coefficient multiplier
10. Distributed arithmetic multiplier (a special case of constant coefficient multiplier)
11. Partial product lookup table based multiplier

The shift and add multiplier is based on a single adder with three registers and some control circuitry. One register is used as the multiplicand and another for the multiplier, which will be shifted at each clock tick, and the last register that is an accumulator and holds the partial result and the final result of multiplication. This multiplication scheme can be done with serial input data.

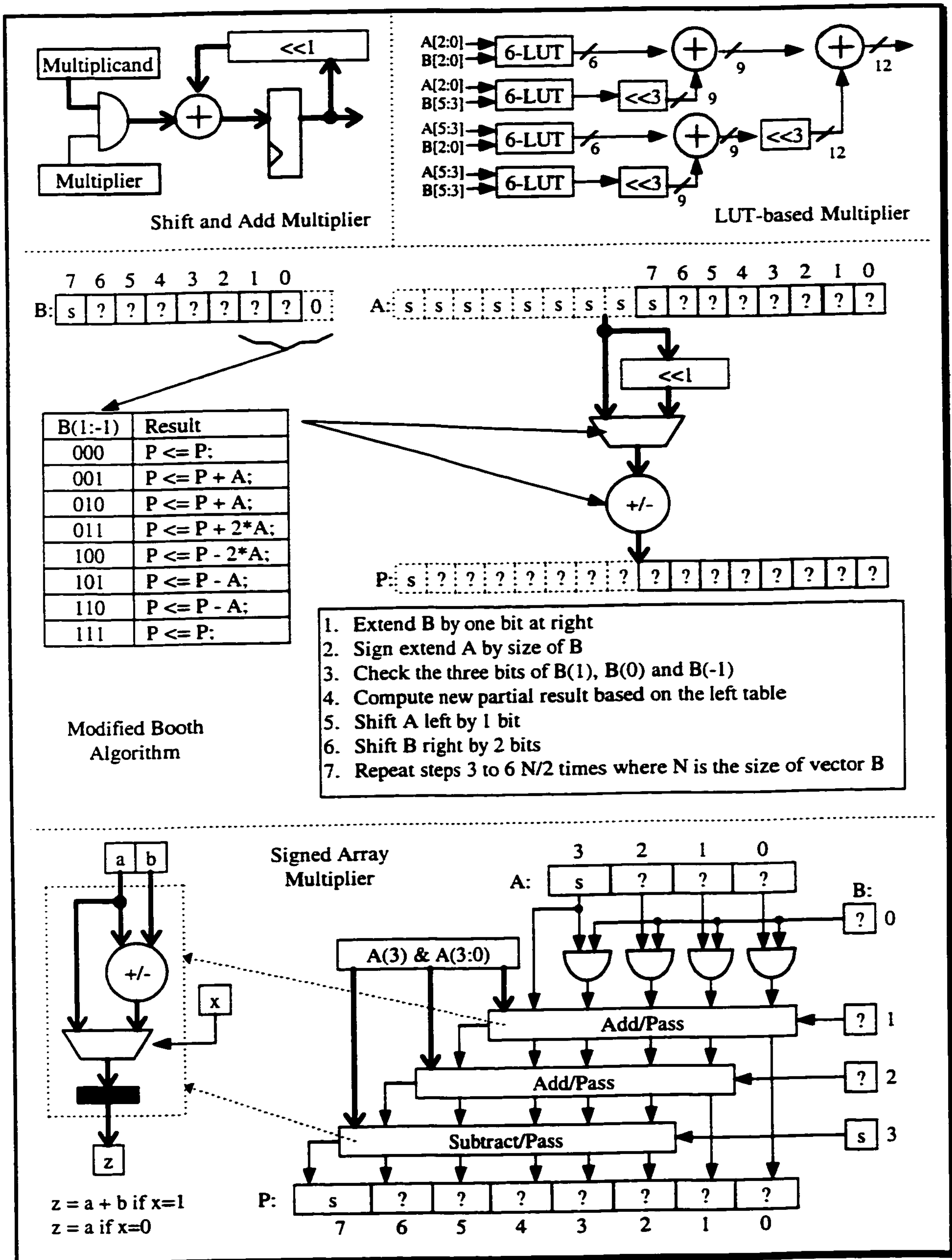


Figure 25. Different multiplication architectures

Figure 25 shows four different popular multiplier architectures. The first one is a shift and add operation that is very area efficient. The other one is a lookup table based 6x6 bit multiplier that divides the two input vectors A[5:0] and B[5:0] and forms partial multiplication results and adds them together. By using four 3x3 lookup tables that hold the values of multiplication of 3-bit by 3-bit numbers and a few shift operations, which use no logic to implement, this multiplier forms the final result.

The third multiplier structure is a modified booth multiplier, which also like the lookup table one, uses a divide and conquer scheme. This algorithm partitions the n-bit multiplier into n/2 3-bit fields with 1-bit overlap. Then based on these three bits it does an add/subtract by multiplicand, add/subtract by twice the multiplicand and no operation. After n/2 iterations the final result is ready. This multiplier can be pipelined at every stage of operation up to n/2 levels. This is a very efficient multiplier in ASIC implementations.

The last multiplier is an array (Braun) signed multiplier that is the exploitation of the multiplication operation expanded into shifts and additions. The first stage is a series of AND gates that ANDs the least significant bit of multiplier by all the bits of the multiplicand. The next stages are a series of adder-multiplexers that pass the previous stages partial result if the corresponding bit of the multiplier is zero, otherwise it is added to the multiplicand. To perform signed multiplication, adders are chosen to be one bit larger and the operands are sign extended and also the last stage should be a subtarctor-multiplexer stage. This multiplier is very easy to implement both in ASIC and FPGA. Although it has more area and it is slower than the Modified-Booth-Recoded multiplier, it is faster and more suited to FPGA implementation.

Piplining this multiplier is a bit more complicated and registers should be put at different places so that the overall timing (arrival of related data) of the multiplier does not change and the correct result is produced at the output. It is possible to do n-level pipelined array multiplier where n is the number of bits in the multiplier. If the number of bits in the multiplier and multiplicand are not equal, there is a trade-off between choosing more adder bits and more levels.

In DSP applications there is also a more dominant operation that is the multiply accumulate of a number of vectors by another constant vector or the inner product of a vector with another constant vector. This is shown by: $y = \sum_{k=1}^{M} A_k X_k$

In this equation A_k is the constant vector and X_k is the input vector. This operation is best done with what is called Distributed Arithmetic (DA). In DSP algorithms, it usually is difficult to distinguish individual operations (additions, multiplications) and hence the name Distributed Arithmetic. This method is basically a bit-serial operation with the difference that multiple vectors can be applied simultaneously. This is usually called n-bit at-a-time DA; where n is the total number of bits serially applied to the DA module. The DA module is composed of a number of lookup tables, an accumulator and a number of shifter units. For better understanding of this enabling technique refer to [37], [38], [39], [40], and [41].

7.1.3. FFT Butterfly Data Path Implementation

As was seen in the previous sections, multipliers are very costly to implement. In chapter 6 decimation-in-frequency FFT algorithm was selected for implementation. From

Figure 17, the detailed data path for the DIF FFT butterfly engine can be derived. It can be seen that there are three additions, three subtractions and four multiplications by the twiddle factors, which are to be pre-computed and stored in lookup tables.

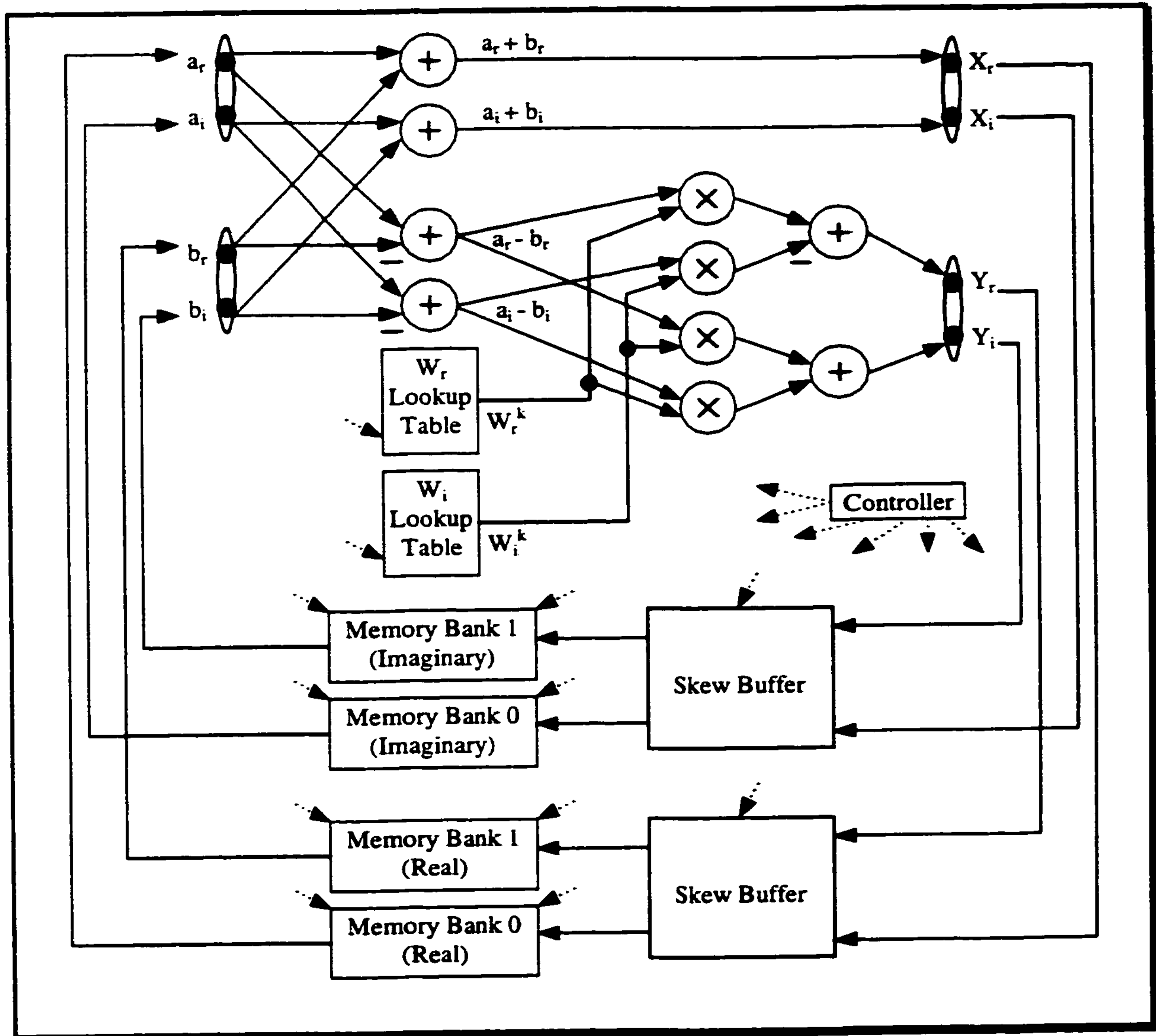


Figure 26. DIF butterfly engine data path details

In Figure 26, pipelining registers for the adders and multipliers are not shown. For increasing the computation speed of the engine, the multiplier is heavily pipelined and additional pipeline registers are inserted after the adders to balance and preserve the actual data dependencies of the data flow. One should be careful of choosing the total number of

pipeline levels. This is because, if the number of iterations in the FFT is less than the number of pipeline levels, the results of the last iteration have not yet been written back to the memories. If this is the case, the algorithm would not function properly. This is true of most algorithms, in which the retiming and the addition of pipeline registers should not affect the outcome of the algorithm.

Figure 27 shows the detailed view of the skew buffers. The inputs to each skew buffer are the two real and the two imaginary parts of the butterfly output. There is a counter that counts the number of the data input to this buffer. At each step of the count a new set of values are stored in a register pair; first R0, then R1, then R2, then R3 and the cycle repeats. The counter is delayed by two cycles and which selects a pair from the register pairs. This construct makes sure that the data has no gaps and the correct order of values are generated at the outputs.

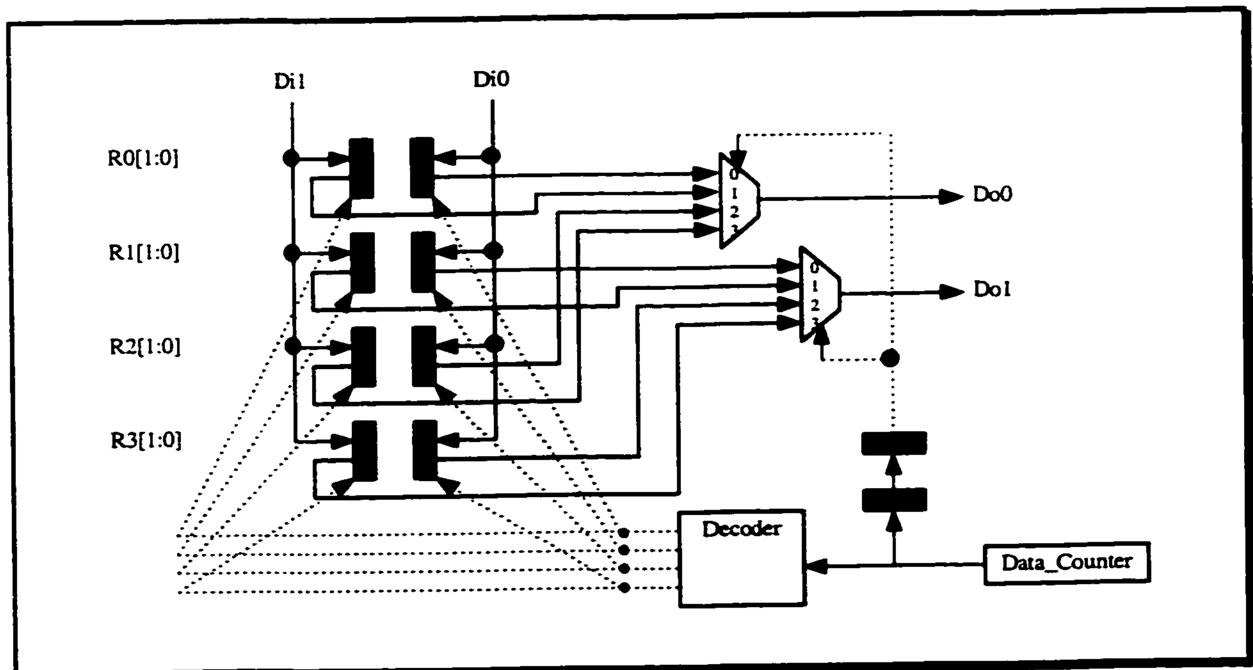


Figure 27. Skew buffer detailed schematic

The input data is assumed to be 8 bits wide for both real and imaginary parts. The memory banks are chosen to have 16-bit data busses. So, the input data is written to the memories on their least significant 8 bits and the final result is truncated to 16 bits for both real and imaginary parts (most significant bits of the final result is used). It is the responsibility of the user to make sure that the final result does not overflow.

The other components are sized based on their input values. The adders and subtractors accept 16-bits signed data. Adding or subtracting two 16-bit data results in a 17-bit data. The multipliers should multiply the output of adders (17 bits) by the 8-bit twiddle factors. This results in 17x8 signed multipliers that produce 25-bit result. The output of the butterfly are truncated to 16-bits, and written back to memories. This may result in some noise ([42], [43], [44], [45]) to be added to the computation, which is true of all fixed-point systems.

As can be seen from Figure 20, for an n-point FFT, two memory banks with n/2 words each are needed and because two banks are needed for storing the real part and imaginary parts of a complex data, there should be total of four memories of n/2 words each. For a 1024-point complex FFT with 16-bit data, four 512*16-bit memories are needed. The total number of bits used for memories is 4*512*16 that is equal to 32768 bits.

With this architecture, there could be a conflict and race to access the memories. The output of the skew registers should be written to the memories and the butterfly should be fed by new data from the memories. One could schedule the operations to be one after another and sequential. But this would increase the number of cycles and reduces

the performance. To alleviate this, one can use dual-port memories. Dual-port memories are very popular in most FPGAs and are also available in most ASIC libraries. If one wanted to use discrete memory component, this would be very costly and probably not a good choice and other schemes should be considered. Having single chip is more desirable than multiple chips in many applications.

FPGAs are very abundant in the number of registers that can also be used as memory elements. But if they are used as memory, there would not be enough registers left for implementing state machines and other functional elements that need registers. In modern FPGA architectures, other than abundant registers, there are also sparse/small flexible memory elements in each CLB[21] that can be configured as single-, dual-port or even Content-Addressable Memory (CAM). There could also be flexible block memories that are larger in size compared to the sparse memory blocks. In Xilinx Virtex FPGAs, there are enough dual-ported block memory to implement the 1024-point FFT.

CLBs could also be configured as read-only memory (ROM) or lookup tables. This is useful for implementing the lookup tables for the real and imaginary parts of the twiddle factors. The twiddle factors are computed using a C program for a specific number of points and are hard coded into the VHDL description.

7.2. Design of the Control Logic

The controller design is responsible for managing the order of operations and to provide control signals to different modules. It has to control the multiplexer select lines, the different modules' enable signals, and the memories control signals. This module is

---

[21] Configurable Logic Block

also responsible for receiving the input data and storing it in the proper order into the memories. It is also responsible for sending the result of the computation out of the module. The input data is assumed to be a stream of 2048 bytes. Each byte pair is a set of real and imaginary data samples.

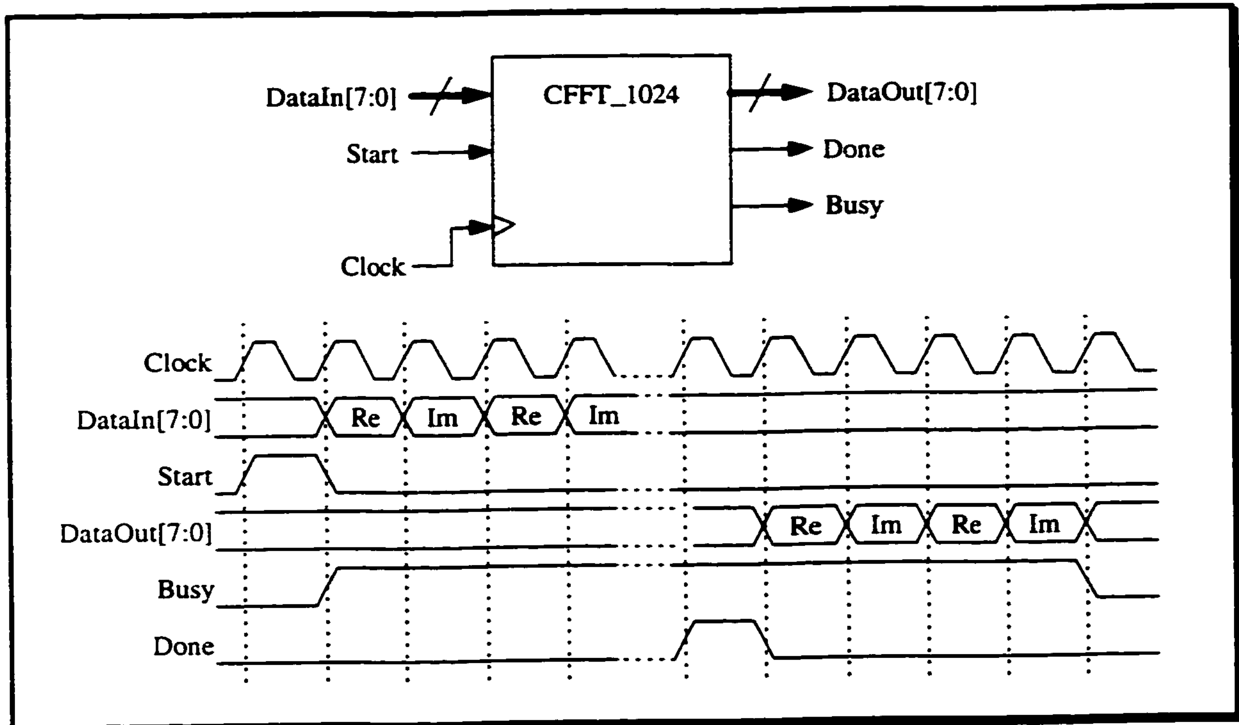


Figure 28. Top-level module for 1024-point complex FFT and its I/O timing

Figure 28 shows the top-level module for the 1024-point complex FFT with the associated input/output timing. The Start signal is asserted and then the input data is applied at the DataIn port, real followed by the imaginary part. After the assertion of the Start signal the Busy signal would go high indicating that the module is busy processing. Busy stays high until the FFT computation is done and the data is sent out on the DataOut port. The start of the output data stream is indicated by the Done signal.

There should be a way to transfer the input data to the memories through the DataIn port. The controller is a Finite State Machine (FSM) that polls the Start signal. As

soon as this signal goes high, the state machine starts one of the bit-reversed address generators, reads in the data and stores them at the proper memory bank and location that was already shown in Figure 20. This is shown as state S0 in state diagram of Figure 29 along with its detailed state names.

After all the data samples are read into the memories, the controller enables the data path, starts reading the data samples from the memory banks and sends them to the butterfly engine. The enable signal on different modules reduces the power consumption of the system and is a good design practice to minimize the amount of logic that is being switched. The controller would write the result of the computation back into the memories at the proper locations after a number of cycles after the application of data that is equal to the pipeline delay of the butterfly engine. The controller will repeat this process 5120 times, which is calculated as $(n/2)*\log_2(n)$ for an n-point FFT. This number is the number of butterflies in an n-point FFT. The number of levels in the FFT is $\log_2(n)$ and the number of nodes in each level is n/2. After the last iteration of the FFT computation the data path pipeline should be flushed to memory. This is shown as state S1 in state diagram of Figure 29 along with its detailed state names.

Finally the FSM has to read the final result out of the memories and send them out on the DataOut. Once this is complete the process is done and the controller goes into the IDLE state where it is ready to receive another set of samples. This is shown as state S2 in state diagram of Figure 29 along with its detailed state names.

The controller is responsible for generating all the control and enable signals to all the modules in the design, so it has lots of signals traveling around the chip. For a chip to

run fast, the data path should be able to run at the required speed and also the controller should be able to provide the control signals at the proper time. One-hot/cold encoding for the state machines are preferred because of the abundant registers in the FPGAs. Otherwise the decoding logic reduces the speed of the design. This encoding type could also be very useful for optimizing critical parts of an ASIC design, because of the much less complex decode logic for the state machine.

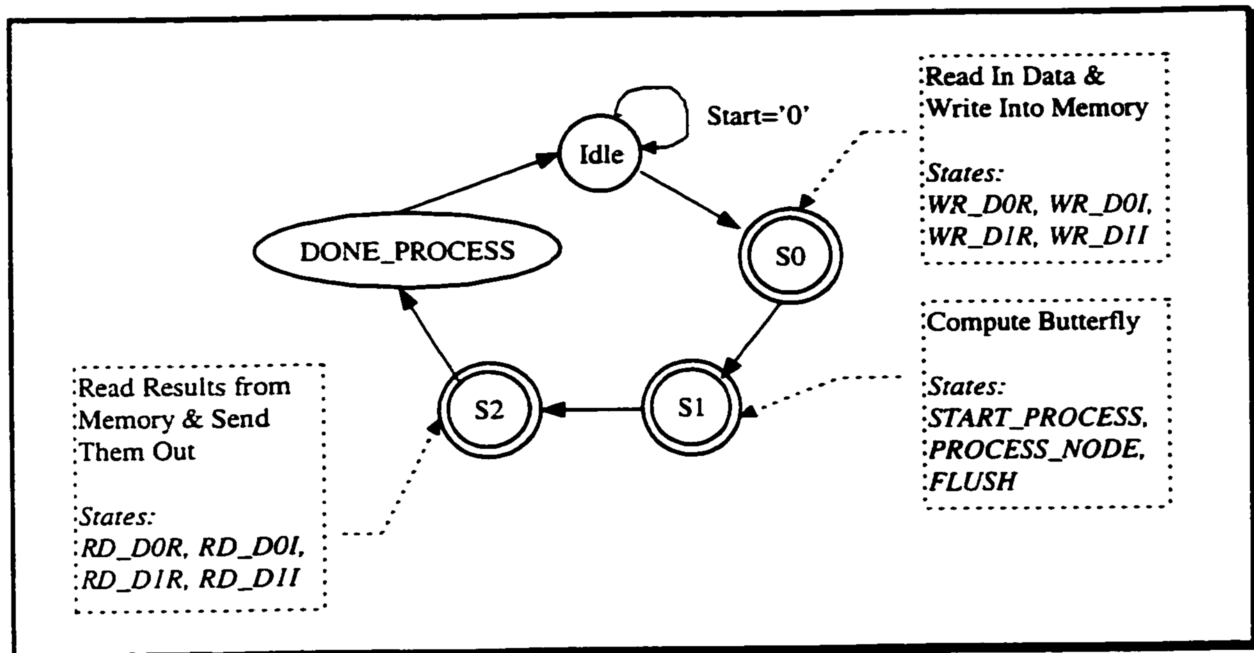


Figure 29. Simplified state diagram of the controller

7.3. Design Synthesis

The synthesis is done using the Synplicity's Synplify tool. The constraints used are only clock constraints. The goal is to run the design at a frequency of 50 MHz. Other types of constraints could be input delays (arrival times), output delays (max delay), multi-cycle paths, which are common to both FPGAs and ASICs and clock skews, output drive and load, which apply to ASICs only.

7.3.1. Synthesis results

The design is successfully placed and routed on a Xilinx Virtex V150PQ240-6 using the Xilinx Alliance v1.5i. It occupies 94% of the device and 66% of the available block RAMs. The timing reports also show that the design is able to run at 50 MHz. Total equivalent ASIC gates, reported by Xilinx Alliance, is 165896.

7.3. Constructing a Testbench

For every HDL design, there should be an associated testbench to verify the functionality of the design. This testbench could also be used to simulate the back-annotated design after the place and route in the FPGAs and after the layout, and routing in the ASICS. A testbench could be written for every single module or for the top-level module only. As a designer becomes more proficient in doing designs in HDL, there may not be a need for every single module, and the top-level simulation is enough. A good testbench should cover all possible scenarios of the unit under test (UUT). Usually, the test vectors or stimulus of the design is stored in files that are read by the VHDL testbench and are applied to the UUT.

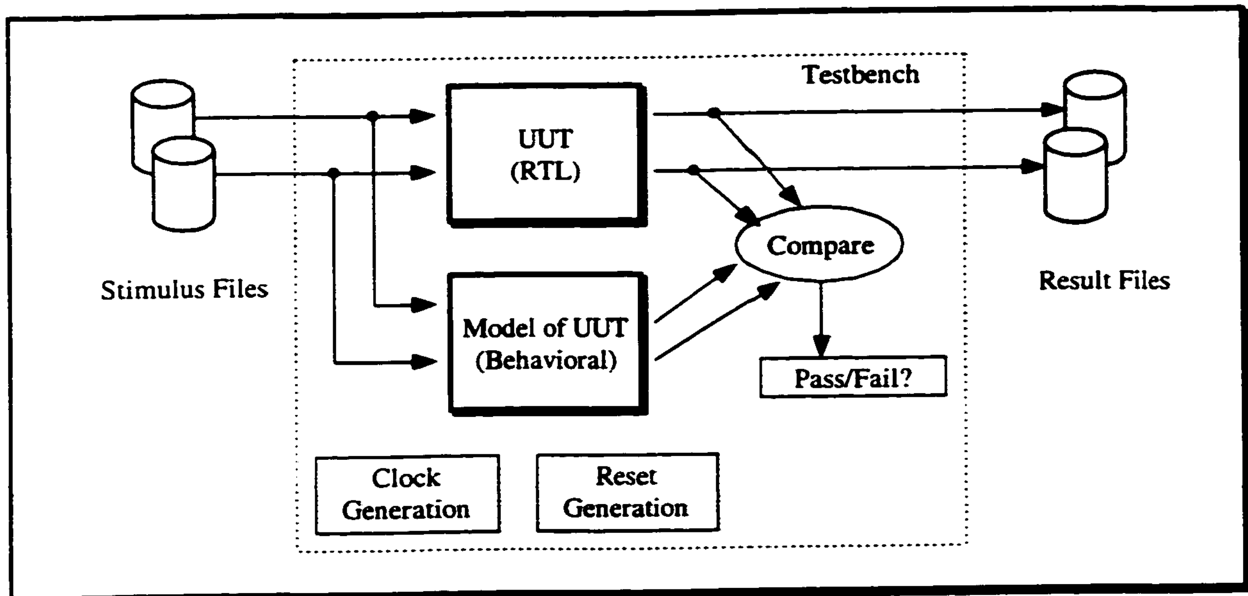


Figure 30. Basic simulation testbench

For verifying functionality, one can choose between two methods. One is that the designer should construct the behavioral model of the design and instantiate it in the testbench, along with the unit under test. Then the stimulus is applied to both the RTL design and the behavioral model. And finally the two outputs are compared in the testbench itself. The second method, which is easier to implement, is that the outputs of the unit under test are stored in files that are compared with the expected results from another source (software simulations). The second method is chosen here for the sake of simplicity and that the purpose and emphasis of this work is on showing the techniques presented.

7.4.1. Results from the simulation and the FFT benchmarks

From the simulations and the structure of this FFT, it is seen that it takes 2048 cycles to transfer the 2048 data bytes (real and imaginary) to the memories and it also takes 2048 cycles to send out the final results. The FFT computation takes 12+(1024/2)*log(1024)+12 = 5144 cycles. With 20 ns cycles time (from the synthesis

result of 50 MHz clock), for transferring data to/from the memories it takes 40.96 μs and to compute the FFT it takes 102.88 μs. If this computation is done after another process, then one can ignore the transfer of data to/from memories.

A comparison between different implementations (from custom ASIC [46], [47] to DSP processor implementations) of the 1024-point radix-2 complex FFT, can be seen in Table 10 and Figure 31 (some of the results are taken from reference [48]). As can be seen in this figure, even the single butterfly implementation of the FFT is very fast compared with most of the general purpose DSPs. The fastest (46 μs) is the Analog Devices Inc. ADSP-21160 and the second fastest (61 μs) is the custom FFT ASIC TM-66 swi-FFT from Texas Memory Systems Inc. It is seen that it is possible to add more butterflies and reduce the execution time. With two butterfly engine, the execution time goes down to 52 μs and with four butterflies down to 26 μs.

Architecture	Time (μs)
4 Butterflies FFT	[illegible]
Analog DSP211160	46
2 Butterflies FFT	52
TM-66 swiFFT	61
1 Butterfly FFT	103
TI TMS320C60xx	104
TI TMS320C80	110
TI TMS320C67xx	125
Analog ADSP2162	170
Motorola DSP56002	210
Lucent DSP1627	310
NEC UPD77015	320
Analog ADSP2171	360
TI TMS320C44	390
TI TMS320C31	410

Table 10. FFT benchmark results (tabulated)

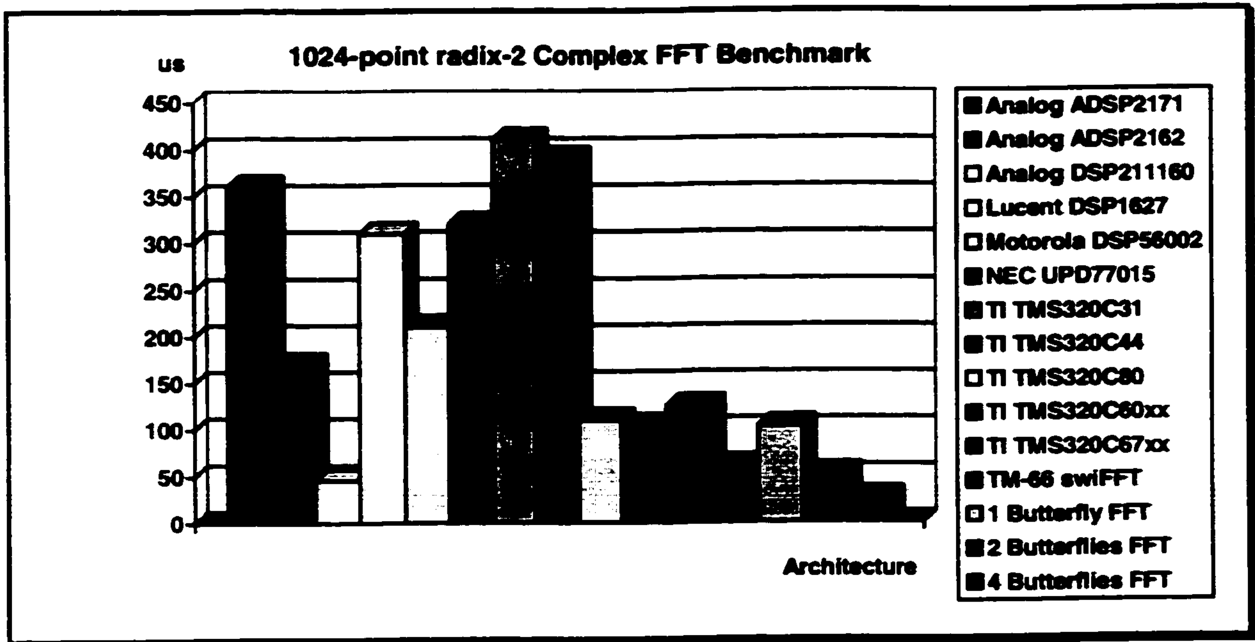


Figure 31. FFT benchmarks results (chart)

Chapter 8

8. Conclusions and Future Work

This concludes the work and provides the missing links for future researchers and interested individuals.

8.1. Conclusions

A generic architecture has been proposed that can execute a variety of digital signal processing algorithms. It consisted of a core processing engine and multiple memory banks that provide the input data to this core and are also used to store the intermediate values and the final results of the computation. A method has been proposed to extract the maximum pipeline level for a specific algorithm represented in signal flow graph form. From this signal flow graph, and by exploring different scan orders of operations, one can extract the delays on each recursive edge of the graph. If all these values are greater than one, it is possible to move all but one of them inside the data path and use them as pipeline registers to speedup the processing engine. After this step, the graph is scheduled and the edges are to be assigned to a memory bank while balancing the accesses. This problem falls into the category of NP-complete problems for a large number of edges, so an exhaustive search method has been developed in C. An ILP formulation is also presented that assists in this assignment and reduces the amount of time necessary to arrive at a reasonable assignment. An automatic ILP generation program has been written in C that works for an arbitrary radix-2 FFT algorithm.

A program has been written (based on previous work) to ease in the design of a hardware-based address generator for arbitrary addresses of size power of two. An efficient architecture for a 1024-point radix-2 FFT has been presented. For this architecture, a novel address assignment and ordering of calculations has been proposed for a two memory bank system that removes the memory address conflicts and provides the core with proper data.

Finally, the complete VHDL design of this 1024 point radix-2 FFT has been done, the design was implemented in an FPGA and simulated in a testbench. A C program has been developed for the generation of twiddle factors for this design.

8.2. Suggested Directions to Continue This Work

The architecture proposed is generalized enough to be used for different DSP algorithms. This should be verified with other types of DSP algorithms and proved efficient with those algorithms. The process of bank assignments using the exhaustive search takes unreasonable amount of time to run, even the ILP formulation has a long run time. Other procedural and formal methods should be devised that would come to a solution with less amount of time.

The heuristics to find the best order of operations and the access order to the memories and to assign addresses for each memory bank should be formalized and expanded to cover different algorithms.

There could be a lot of improvements in the address generator and its generalization. One can find an automatic processes to synthesize arbitrary hardware-based address generators for any type of access and algorithm.

The implementation of the FFT design could be improved by parallelizing the transfer of data in/out of the memories; i.e., while new data is being transferred to the memories the old results could be transferred out of the memories. This requires some modifications to the first write and last read orders; otherwise there would be conflicts and data corruption. The number of butterfly engines and the memory banks could be increased to increase the throughput and decrease the execution time of the FFT. New address assignment and access order should be devised to alleviate the conflicts.

Bibliography

[1] Donald E. Thomas, Jay K. Adams, Herman Schmit, "A Model and Methodology for Hardware-Software Codesign," IEEE Design & Test of Computers, pp. 6-15, 1993

[2] Sanjaya Kumar, James H. Aylor, Barry W. Johnson, Wm. A. Wulf, "A Framework for Hardware/Software Codesign," IEEE Computer, pp. 39-45, Dec. 1993

[3] Alan S. Wenban, John W. O'Leary, Geoffrey M. Brown, "Codesign of Communication Protocols," IEEE Computer, pp. 46-52, Dec. 1993

[4] Nam S. Woo, Alfred E. Dunlop, Wayne Wolf, "Codesign from Cospecification," IEEE Computer, pp. 42-47, Jan. 1994

[5] Rajesh K. Gupta, Giovanni De Micheli, "Hardware-Software Cosynthesis for Digital Systems," IEEE Design & Test of Computers, pp. 29-41, Sep. 1993

[6] Asawaree Kalavade, Edward A. Lee, "A Hardware-Software Codesign Methodlogy for DSP Applications," IEEE Design & Test of Computers, pp. 16-28, Sep. 1993

[7] David E.Van Den Bout, Joseph N. Morris, Douglas Thomae, Scott Labrozzi, Scot Wingo, Dean Hallman, "AnyBoard: An FPGA-Based Reconfigurable System," IEEE Design & Test of Computers, pp. 21-30, Sep. 1992

[8] Robert A. Walker and Raul Composano, "A Survey of High-Level Synthesis Systems," Kulwer Academics Publishing, 1991

[9] A. Aho, R. Sethi and J. Ulman, "Compilers," Addison-Wesley, 1986

[10] H. Lipp, "Methodical Aspects of Logic Synthesis," Proceedings of IEEE, vol. 71, pp. 88-97, Jan. 1983.

[11] H. Trickey, "Flamel: A High-level Hardware Compiler," IEEE Transactions on Computer-Aided Design, vol. CAD-6, pp. 259-269, Mar. 1987.

[12] Baher S. Haroun and Mohamed I. Elmasry, "Architectural Synthesis for DSP Silicon Compilers," IEEE Transactions on Computer-Aided Design, vol. 8, no. 4, Apr. 1989.

[13] David J. Kolson, Alexandru Nicolau, and Nikil Dutt, "Elimination of Redundant Memory Traffic in High-Level Synsthesis," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 15, no. 11, pp. 1354-1364, Nov. 1996

[14] David T. Harper III, "Block, Multistride Vector, and FFT Access in Parallel Memory Systems," IEEE Transactions on Parallel and Distributed Systems, vol. 2, no. 1, pp. 43-51, Jan. 1991.

[15] David T. Harper III and D. A. Linebarger, "Storage Schemes for Efficient Computation of Radix 2 FFT in a Machine with Parallel memories," Proceedings 1988 International Conference on Parallel Processing, pp. 422-425, 1988.

[16] David T. Harper III, "Address Transformations to Increase Memory Performance," Proceedings 1989 International Conference on Parallel Processing, pp. I237- I241,

1989.

[17] Jan Vanhoof, Karl Van Rompaey, Ivao Bolsens, Gert Goosens, Hugo De Man, "High-level Synthesis for Real-time Digital Signal Processing," "Implementation of data structures," pp. 59-115.

[18] Michael F.X.B. van Swaaij, Frank H.M. Franssen, Francky V.M. Cathoor, Hugo J. De Man, "Modeling Data Flow and Control Flow for DSP System Synthesis", pp. 219-259.

[19] Michael E. Wolf and Monica S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," IEEE Transactions on Parallel and Distributed Systems, vol. 2, no. 4, pp. 452-471, Oct. 1991.

[20] L. Mullin and S. Thibault, "A reduction semantics for array expressions: the PSI compiler". Technical Report CSC-94-05, Computer Science Department, University of Missouri-Rolla, 1994.

[21] Yvon Savaria, ..., "A 2D 3x3 Convolusion Engine in FPGA," Polytechnique University of Montreal, Electrical Engineering Department, 1995

[22] W. Eatherton, J. Kelly, T. Schiefelbein, H. Pottinger, L. R. Mullin and R. Ziegler, "An FPGA Based Reconfigurable Coprocessor Board Utilizing Mathematics of Arrays," Computer Science Department, University of Missouri-Rolla, 1994.

[23] H. Pottinger, W. Eatherton, J. Kelly, T. Schiefelbein, L. R. Mullin and R. Ziegler, "An FPGA Based Reconfigurable Coprocessor Board Utilizing Inteligent Data

Prefetching," Computer Science Department, University of Missouri-Rolla, 1994.

[24] H. Pottinger, W. Eatherton, J. Kelly, T. Schiefelbein, L. R. Mullin and R. Ziegler, "Hardware Assists for High Performance Computing Using a Mathematics of Arrays," Computer Science Department, University of Missouri-Rolla, pp39-45, 1994

[25] G. Chaitin, M.Auslander, A. Chandra, J. Coocke, M.Hopkins and P. Markstein, "Register allocation via coloring," Computer Languages, Vol. 6, pp. 47-57, Jan. 1981.

[26] F. Chow and J.Henessy, "The Priority-based Coloring approach to register allocation," ACM Transactions on Programming Languages and systems, vol. 12, no. 4, pp. 501-536, Oct. 1990.

[27] Thomas Lengauer, "Combinatorial Algorithms for Circuit Layout," John Wiley & Sons, 1990

[28] Reiner W. Hartenstein, Helmut Reining and Markus Weber, "Design of an Address Generator," Proceedings 3rd Eurochip Workshop on VLSI Design Training, Grenoble, Sep. 1992

[29] Reiner W. Hartenstein and Helmut Reining, "Novel Sequencer Hardware for High-Speed Signal Processing," Workshop on Design Methologies for Microelectronics, Smolenice Castle, Slovakia, Sep. 1995

[30] Reiner W. Hartenstein, Jürgen Becker, Michael Hertz and Ulrich Nageldinger, "A

Novel Sequencer Hardware for Application Specific Computing," Proceedings of 11th International Conference on Application-specific Systems, Architectures and Processors, ASAP'97, Zurich, Switzerland, Jul. 1997

[31] Reiner W. Hartenstein, Jürgen Becker, Michael Hertz and Ulrich Nageldinger, "A Novel Universal Sequencer Hardware," Proceedings of Fachtagung Architekturen Von Rechenstemen ARCS'97, Rostock, Germany, Sep. 1997

[32] D. Grant, P. B. Denyer and I. Finlay, "Synthesis of Address Generators," Proceedings IEEE International Conference on Computer Aided Design, Santa Clara CA, pp. 116-119, Nov. 1989

[33] Xilinx, "The Programmable Logic Data Book", 1998.

[34] Reto Zimmermann, "Computer Arithmetic: Principles, Architectures, and VLSI Design," Integrated Systems Laboratory, Swiss Federal Institute of Technology (ETH), Mar. 16, 1999

[35] Abdelkrim Kamel Oudjida, "High Speed and Very Compact Two's Complement Serial/Parallel Multipliers using Xilinx's FPGA," CDTA/Microelectronics Laboratory, 1994

[36] Gin-Kou Ma and Fred J. Taylor, "Multiplier Policies For Digital Signal Processing," IEEE ASSP Magazine, pp.6-20, Jan. 1990

[37] Kamal Nourji and Nicolas Demassieux, "Optimal VLSI Architectures for Distributed Arithmetic-based Algorithms," ICASSP, 1994

[38] Stanley A. White, "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Overview," IEEE ASSP Magazine, pp. 4-19, Jul. 1989

[39] C. Sidney Burrus, "Digital Filter Structures Described by Distributed Arithmetic," IEEE Transactions on Circuits and Systems, vol. CAS-24, no. 12, pp. 674-680, Dec. 1977

[40] Abraham Peled and Bede Liu, "A New Hardware realization of Digital Filters," IEEE Transactions on Acoustics, Speech and Signal Processing, vol. ASSP-22, no. 6, pp. 456-462, Dec. 1974

[41] Shalhav Zohar, "New Hardware Realization of Nonrecursive Digital Filters," IEEE Transactions on Computers, vol. C-22, no. 4, pp. 328-338, Apr. 1973

[42] Bede Liu, "Effect of Finite Word Length on the Accuracy of Digital Filters - A Review," IEEE Trans. On Circuit Theory, vol CT-18, pp.670-677, Nov. 1974

[43] T. Kaneko, B. Liu, "Accumulation of Round-off Error in Fast Fourier Transfomrs," Journal of Ass. Comput. Mach., vol 17., pp. 637-654, Oct. 1970

[44] David C. Munson Jr., Bede Liu, "Low-Noise Realizations for Narrow-Band Recursive Digital Filters," IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-28, no. 1, pp. 41-54, Feb. 1980

[45] Alan V. Oppenheim, Ronald W. Schafer, "Discrete-Time Signal Processing," Prentice Hall, 1989

[46] S. Y. Kung, H. J. Whitehouse and T. Kailath, "VLSI and Modern Signal

Processing," 1985

[47] Earl E. Swartzlander, Jr., George Hallnor, "High Speed FFT Processor Implementation," VLSI Signal Processing, IEEE press, pp.27-34, 1984

[48] Mintzer, L., "Large FFTs in a single FPGA," Proceedings of ICSPAT 1996

Useful URL Resources

ASICs... the website	
	http://www-ee.eng.hawaii.edu/~msmith/ASICs/HTML/ASICs.htm
Altera Home Page	
	http://www.altera.com
Cryptography Research Home Page	
	http://www.cryptography.com/
Data Compression Pointers	
	http://www.internz.com/compression-pointers.html
Don Lancaster's GURU'S LAIR home page	
	http://www.tinaja.com/
EDIF Home Page	
	http://www.edif.org/
Hamburg VHDL Archive	
	http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html
Hardware Compilation Home Page	
	http://www.comlab.ox.ac.uk/oucl/hwcomp.html
Library of Arithmetic Modules	
	http://www.iis.ee.ethz.ch/zimmi
Mathematics of Arrays and PSI Compiler	
	http://www.cs.umr.edu/~ryep/moa/moacc.html *http://www.cs.albany.edu/~psi/efforts/compiler/compiler.html* *http://www.cs.albany.edu/~psi/research_efforts.html*
Model Technology	
	http://www.model.com
OptiMagic's Programmable Logic (FPGA, CPLD) Jump Station	
	http://www.optimagic.com/
Reconfigurable Cryptography (A Hardware Compiler for Cryptographic Applications)	
	http://www.pdos.lcs.mit.edu/~cananian/Projects/ele580a/writeup.html
(All the Best of) Spread Spectrum Scene Online	
	http://sss-mag.com/index.html

Synplicity HomePage
http://www.synplicity.com
TechOnLine
http://www.techonline.com
VHDL International Home Page
http://www.vhdl.org/
VIUF comp.lang.vhdl Archive
http://vhdl.org/vi/comp.lang.vhdl/
Xilinx Hompage
http://www.xilinx.com
Xputer Page
http://xputers.informatik.uni-kl.de/xputer/index_xputer.html

Appendix

A. Memory Bank Assignment Exhaustive Search

The program is called BANKS, is written in C and is included on the accompanying diskette in the BANKS folder. The source is called BANKS.C and the executable is BANKS.EXE. It has been compiled using Microsoft Visual C++ 5.0. All the project files necessary files is also included.

A.1. Exhaustive Search C Source Program for 16-point radix-4 FFT

```
#include <conio.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define FALSE (0 == 1)
#define TRUE  (1 == 1)

char *Copyright = "Memory Bank Assignment Exhaustive Search for 16-point radix-4 FFT\n"
                  "Copyright (c) 1999 Amal Khailtash (akhailtash@spacebridge.com)\n\n";

#define NO_OF_EDGES  32
#define NO_OF_NODES  8

int src[NO_OF_EDGES] = {
  0, 0, 0, 0,
  1, 1, 1, 1,
  2, 2, 2, 2,
  3, 3, 3, 3,
  4, 4, 4, 4,
  5, 5, 5, 5,
  6, 6, 6, 6,
  7, 7, 7, 7
};

int dst[NO_OF_EDGES] = {
  4, 5, 6, 7,
  4, 5, 6, 7,
  4, 5, 6, 7,
  4, 5, 6, 7,
  0, 0, 0, 0,
  1, 1, 1, 1,
  2, 2, 2, 2,
  3, 3, 3, 3
};

int node_i_o[NO_OF_NODES][2][4] = {
  { { 16, 17, 18, 19 }, {  0,  1,  2,  3 } },
  { { 20, 21, 22, 23 }, {  4,  5,  6,  7 } },
  { { 24, 25, 26, 27 }, {  8,  9, 10, 11 } },
  { { 28, 29, 30, 31 }, { 12, 13, 14, 15 } },
  { {  0,  4,  8, 12 }, { 16, 17, 18, 19 } },
  { {  1,  5,  9, 13 }, { 20, 21, 22, 23 } },
  { {  2,  6, 10, 14 }, { 24, 25, 26, 27 } },
  { {  3,  7, 11, 15 }, { 28, 29, 30, 31 } }
};

struct edge {
  int src_node;
```

```
  int dst_node;
  int bank;
} edges[NO_OF_EDGES];

struct node {
  int inputs[4];
  int outputs[4];
} nodes[NO_OF_NODES];

void read_edges_and_nodes( void )
{
  int i, j;

  for( i=0; i<NO_OF_EDGES; i++ )
  {
    edges[i].src_node = src[i];
    edges[i].dst_node = dst[i];
  }
  for( i=0; i<NO_OF_NODES; i++ )
  {
    for( j=0; j<4; j++ )
    {
      nodes[i].inputs[j]  = node_i_o[i][0][j];
      nodes[i].outputs[j] = node_i_o[i][1][j];
    }
  }
}

void assign_banks( unsigned long b )
{
  int i;

  for( i=0; i<NO_OF_EDGES; i++ )
  {
    edges[i].bank = (int)((b >> i) & 0x1L);
  }
}

struct symbol {
  int cost;
  int count;
  int used;
} symbols[16];

symbol_costs[] = { 2, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 2 };

void init_symbols_costs( void )
{
  int i;

  for (i=0; i<16; i++)
  {
    symbols[i].cost  = symbol_costs[i];
    symbols[i].count = 0;
    symbols[i].used  = FALSE;
  }
}

int calculate_cost( int *symbols_used )
{
  int  i, symbol_i, symbol_o;
  int  cost;

  for (i=0; i<NO_OF_NODES; i++)
  {
    symbol_i = edges[nodes[i].inputs[0]].bank +
               edges[nodes[i].inputs[1]].bank * 2 +
               edges[nodes[i].inputs[2]].bank * 4 +
               edges[nodes[i].inputs[3]].bank * 8;

    symbol_o = edges[nodes[i].outputs[0]].bank +
               edges[nodes[i].outputs[1]].bank * 2 +
               edges[nodes[i].outputs[2]].bank * 4 +
               edges[nodes[i].outputs[3]].bank * 8;
```

```
    symbols[symbol_i].count++;
    symbols[symbol_i].used = TRUE;
    symbols[symbol_o].count++;
    symbols[symbol_o].used = TRUE;
  }
  *symbols_used = 0; cost = 0;
  for (i=0; i<16; i++)
  {
    if (symbols[i].used)
    {
      (*symbols_used)++;
      cost += symbols[i].cost * symbols[i].count;
      symbols[i].count = 0;  /* reset the counter */
      symbols[i].used  = FALSE;
    }
  }
/*  printf("%d\t", cost);*/

  return cost;
}

void exit_if_ESC_pressed( void )
{
  if ( kbhit() )
    if ( getch()==27 )
      exit(0);
}

int SPACE_pressed( void )
{
  int ch;

  if ( kbhit() )
  {
    if ( (ch=getch())==' ' )
      return TRUE;
    else if ( ch==27 )
      exit(1);
    else
      return FALSE;
  } else
    return FALSE;
}

void wait_for_SPACE( char *msg )
{
  printf( "%s\n", msg );
  while( getch()!=' ' );
}

void report_time( char *msg )
{
  struct tm   *tm;
  time_t      current_time;
  static char time_now[80];

  time( &current_time );
  tm = localtime( &current_time );
  sprintf( time_now, "%02d:%02d:%02d", tm->tm_hour, tm->tm_min, tm->tm_sec );
  printf( "%s%s\n", msg, time_now );
}

void get_time( char *now )
{
//  struct tm   *tm;
  time_t      current_time;

  time( &current_time );
//  tm = localtime( &current_time );
//  sprintf( now, "%02d:%02d:%02d", tm->tm_hour, tm->tm_min, tm->tm_sec );
  sprintf( now, asctime(localtime(&current_time)) );
}

void main()
{
```

```
  unsigned long  i, start;
  int        current_cost, last_cost = 99999;
  int        words;
  FILE       *fp;
  int        show = TRUE;
  char       time_now[80];

  printf( Copyright );
  printf( "Press ESC to exit program.\n"
          "      SPACE to stop/restart displaying the current cost...\n\n" );

  do {
    printf( "Enter starting cost in hex (0 if exploring all): " );
  } while( scanf( "%x", &start )!= 1 );

  fp = fopen( "banks.dat", "w" );
  if ( fp==NULL )
  {
    fprintf(stderr, "Unable to open 'bank.dat'.\n");
    exit(1);
  }
  fprintf( fp, Copyright );

//  report_time( "Started at " );
  get_time( time_now );
  printf( "\nStarted at %s\n", time_now );
  fprintf( fp, "Started at %s\n", time_now );

  read_edges_and_nodes();
  init_symbols_costs();

  for( i=start; i<0xFFFFFFFFL; i++ )
  {
/*    exit_if_ESC_pressed();*/
    if ( SPACE_pressed() )
    {
      if ( show ) show = FALSE;
      else        show = TRUE;
    }

    assign_banks( i );

    current_cost = calculate_cost( &words );
    current_cost += words;

    if ( show )
    {
/*      printf( "%08lX\t%d\t%d", i, current_cost, words );*/
      printf( "\r%08lX", i );
    }

    if ( current_cost<=last_cost )
    {
      printf( "\rLast = %6d\tCurrent = %6d\tWords = %d\tSymbol = %08lX\n",
              last_cost, current_cost, words, i );
      fprintf( fp, "Last = %6d\tCurrent = %6d\tWords = %d\t%08lX\n",
        last_cost, current_cost, words, i );
      last_cost = current_cost;
    }
/*    wait_for_SPACE();*/
  }

//  report_time( "Finished at " );
  get_time( time_now );
  printf( "Finished at  %s\n", time_now );
  fprintf( fp, "Finished at %s\n", time_now );
  fclose( fp );

  wait_for_SPACE( "Press SPACE to exit." );
}
```

A.2. Sample Output of the Exhaustive Search

This is a shortened version of the actual file that is included on the accompanied disk.

```
Memory Bank Assignment Exhaustive Search for 16-point radix-4 FFT
Copyright (c) 1999 Amal Khailtash (akhailtash@spacebridge.com)

Started at Tue May 13 09:05:43 1997
Last = 99999  Current =    33    Words = 1    Symbol = 00000000
Last =    33  Current =    32    Words = 2    Symbol = 00000001
Last =    32  Current =    31    Words = 3    Symbol = 00000003
Last =    31  Current =    31    Words = 3    Symbol = 00000005
Last =    31  Current =    31    Words = 3    Symbol = 00000006
Last =    31  Current =    31    Words = 3    Symbol = 00000007
Last =    31  Current =    31    Words = 3    Symbol = 00000009
Last =    31  Current =    31    Words = 3    Symbol = 0000000A
Last =    31  Current =    31    Words = 3    Symbol = 0000000B
Last =    31  Current =    31    Words = 3    Symbol = 0000000C
Last =    31  Current =    31    Words = 3    Symbol = 0000000D
Last =    31  Current =    31    Words = 3    Symbol = 0000000E
Last =    31  Current =    31    Words = 3    Symbol = 0000000F
Last =    31  Current =    31    Words = 3    Symbol = 00000011
Last =    31  Current =    31    Words = 3    Symbol = 00000012
Last =    31  Current =    29    Words = 3    Symbol = 00000013
Last =    29  Current =    29    Words = 3    Symbol = 00000032
Last =    29  Current =    26    Words = 2    Symbol = 00000033
....
Last =    26  Current =    26    Words = 4    Symbol = 00000136
Last =    26  Current =    26    Words = 4    Symbol = 00000253
Last =    26  Current =    25    Words = 5    Symbol = 0000033C
....
Last =    24  Current =    24    Words = 4    Symbol = 00000536
Last =    24  Current =    24    Words = 4    Symbol = 00000669
Last =    24  Current =    24    Words = 4    Symbol = 00000996
Last =    24  Current =    24    Words = 4    Symbol = 00000A5A
Last =    24  Current =    24    Words = 4    Symbol = 00000C33
Last =    24  Current =    24    Words = 4    Symbol = 000011CC
Last =    24  Current =    23    Words = 5    Symbol = 0000136C
Last =    23  Current =    23    Words = 5    Symbol = 000013CC
Last =    23  Current =    23    Words = 5    Symbol = 0000165A
Last =    23  Current =    22    Words = 4    Symbol = 00001669
Last =    22  Current =    22    Words = 4    Symbol = 000025A5
Last =    22  Current =    19    Words = 3    Symbol = 000033CC
....
Last =    19  Current =    19    Words = 3    Symbol = 00005A5A
....
Last =     2  Current =     2    Words = 2    Symbol = 99669669
Last =     2  Current =     2    Words = 2    Symbol = 99696996
Last =     2  Current =     2    Words = 2    Symbol = 99699669
Last =     2  Current =     2    Words = 2    Symbol = 99966996
Last =     2  Current =     2    Words = 2    Symbol = 99969669
Last =     2  Current =     2    Words = 2    Symbol = 99996996
Last =     2  Current =     2    Words = 2    Symbol = 99999669
Last =     2  Current =     2    Words = 2    Symbol = A5555A5A
Last =     2  Current =     2    Words = 2    Symbol = A555A5A5
Last =     2  Current =     2    Words = 2    Symbol = A55A5A5A
Last =     2  Current =     2    Words = 2    Symbol = A55AA5A5
Last =     2  Current =     2    Words = 2    Symbol = AA5A5A5A
Last =     2  Current =     2    Words = 2    Symbol = AA5AA5A5
Last =     2  Current =     2    Words = 2    Symbol = AAA55A5A
Last =     2  Current =     2    Words = 2    Symbol = AAA5A5A5
Last =     2  Current =     2    Words = 2    Symbol = AAAA5A5A
Last =     2  Current =     2    Words = 2    Symbol = AAAAA5A5
Last =     2  Current =     2    Words = 2    Symbol = C33333CC
Last =     2  Current =     2    Words = 2    Symbol = C3C3CC33
Last =     2  Current =     2    Words = 2    Symbol = C3CC33CC
Last =     2  Current =     2    Words = 2    Symbol = C3CCCC33
Last =     2  Current =     2    Words = 2    Symbol = CC3333CC
Last =     2  Current =     2    Words = 2    Symbol = CC33CC33
Last =     2  Current =     2    Words = 2    Symbol = CC3C33CC
Last =     2  Current =     2    Words = 2    Symbol = CC3CCC33
Last =     2  Current =     2    Words = 2    Symbol = CCC333CC
Last =     2  Current =     2    Words = 2    Symbol = CCC3CC33
Last =     2  Current =     2    Words = 2    Symbol = CCCC33CC
Last =     2  Current =     2    Words = 2    Symbol = CCCCCC33
Finished at Thu May 15 15:03:18 1997
```

B. Program to Generate the ILP Source File for Arbitrary FFT

The program is compiled using the Microsoft Visual C++ v5.0.

B.1. Program (GILP_FFT.C) for Generating Bank Assignment ILP, arbitrary FFT

```
#include <conio.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *Copyright = "ILP Generator for radix-2 FFT\n"
                  "Copyright (c) 1999 Amal Khailtash (akhailtash@spacebridge.com)\n\n";

char *UsageMsg  = "Usage: GILP_FFT <number_of_points> <radix> <levels>\n"
                  "       number_of_points : is the number of points in the FFT.\n"
                  "       radix            : is the FFT radix.\n"
                  "       levels           : is the number of levels in the graph.\n";

//#define DEBUG

int N,  /* Number of points */
    R,  /* Radix */
    L,  /* Levels */
    B,  /* Number of memory banks */
    I,  /* Number of iteretions */
    E,  /* Number of edges */
    S,  /* Number of symbols */
    P;  /* */

void print_header( void )
{
  printf( "$TITLE Assignment of memory banks to variables\n"
          "$OFFUPPER\n\n"
          "*************************************************************\n"
          "* Copyright (c) 1999 Amal Khailtash\n"
          "*   (akhailtash@spacebridge.com)\n"
          "*************************************************************\n" );
  printf( "* %d-point radix-%d FFT with\n"
          "* %d levels and %d memory banks\n", N, R, L, B );
  printf( "*************************************************************\n"
          "* Indices (sets)\n"
          "*************************************************************\n"
          "SETS\n" );
  printf( "  I Iterration number / 0 * %d /\n"
          "  S Symbol            / 0 * %d /\n"
          "  B Bit index         / 0 * %d /\n"
          "  E Edge index        / 0 * %d /\n", I-1, S-1, R-1, E-1 );
  printf( "  ;\n\n"
          "ALIAS (I, J) ;\n"
          "ALIAS (B, BI, BJ) ;\n\n" );
}

void print_fft_data( void )
{
  int i, r, e;
//  int **wi;

//  wi = (int **)malloc( I*sizeof(int**) );
//  for( i=0; i<I; i++ )
//    wi[i] = (int *)malloc( R*sizeof(int*) );

  printf( "SETS\n"
          "  WI(I, E) Writer's Iteration number\n    /\n" );
  for( i=0; i<I; i++ )
  {
    printf( "      %2d.    (", i);
    for( r=0; r<R; r++ )
    {
```

```
      e = i * R + r;
      printf( "%4d%s", e, r<R-1 ? "," : " )" );
//      wi[i][r] = e;
    }
    printf( "\n" );
  }
  printf( "    /\n" );
#ifdef DEBUG
//  for( i=0; i<I; i++ )
//  {
//    for( r=0; r<R; r++ )
//    {
//      printf( "WI(%2d, %2d) = %2d   ", i, r, wi[i][r] );
//    }
//    printf( "\n" );
//  }
#endif
  printf( "  RI(J, E) Reader's Iteration number\n    /\n" );
  for( i=0; i<I; i++ )
  {
    printf( "      %2d.    (", i );
    for( r=0; r<R; r++ )
    {
      if ( i<(N/R) )
        e = i * R + r + E/2;
      else
        e = (I/2) * r + i - (I/2);
      printf( "%4d%s", e, r<R-1 ? "," : " )" );
    }
    printf( "\n" );
  }
  printf( "    /\n" );

  printf( "  BW(BI, E) Writer's bit number\n    /\n" );
  for( i=0; i<R; i++ )
  {
    printf( "      %2d.    (", i );
    for( r=0; r<2*R; r++ )
    {
      e = r * R + i;
      printf( "%4d%s", e, r<2*R-1 ? "," : " )" );
    }
    printf( "\n" );
  }
  printf( "    /\n" );

  printf( "  BR(BJ, E) Reader's bit number\n    /\n" );
  for( i=0; i<R; i++ )
  {
    printf( "      %2d.    (", i );
    for( r=0; r<2*R; r++ )
    {
      if ( r<R )
        e = i * R + r;
      else
        e = r * R + i;
      printf( "%4d%s", e, r<2*R-1 ? "," : " )" );
    }
    printf( "\n" );
  }
  printf( "    /\n"
          "  EDGE_EXTS(E, I, J, BI, BJ) Edge exists\n"
          "  ;\n\n" );

//  for( i=0; i<I; i++ ) {
//    free( wi[i] );
//    printf( "Here\n" ); }
//  free( wi );
//  printf( "Here\n" );
}

void print_table( void )
{
  int  r, s, i, b;
  char buf[16], sym[16], str[2] = "?";
```

```
  char **sym_tab;
  int  *sym_cost, *banks, cost;

  sym_tab = (char **)malloc( S*sizeof(char **) );
  for( s=0; s<S; s++ )
    sym_tab[s] = (char *)malloc( 16*sizeof(char *) );

  sym_cost = (int *)malloc( S*sizeof(int*) );
  banks = (int *)malloc( B*sizeof(int*) );

#ifdef DEBUG
  printf( "%d\n", S );
  for( s=0; s<S; s++ ) { fprintf( stderr, "%d ", s); strcpy( sym_tab[s], "1234" ); }
  for( s=0; s<S; s++ ) printf( " ------> %s\n", sym_tab[s] );
#endif

  printf( "************************************************************\n"
          "** Given data (parameters, tables, scalars)\n"
          "************************************************************\n"
          "TABLE\n"
          "  BITS(S, B) binary equivalents of symbol S\n"
          "            " );

  for( r=R-1; r>=0; r-- )
    printf( "%d    ", r );
  printf( "\n" );
  for( s=0; s<S; s++ )
  {
    printf( "  %3d     ", s );
    itoa( s, buf, B );
    i = 0;
    strcpy( sym, "" );
    for( r=R-1; r>=0; r-- )
    {
      if ( r<(int)strlen(buf) )
      {
        printf( "%c    ", buf[i] );
        str[0] = buf[i];
        strcat( sym, str );
        i++;
      } else {
        printf( "0    " );
        strcat( sym, "0" );
      }
    }
    strcpy( sym_tab[s], sym );
//    printf( " --------> %s", sym_tab[s] );
    printf( "\n" );
  }
  printf( "  ;\n\n" );

//  for( s=0; s<S; s++ ) printf( " ------> %s\n", sym_tab[s] );

  printf( "PARAMETERS\n"
          "  SYM_COST(S) Cost of each symbol\n" );

  printf( "    /" );
  for( s=0; s<S; s++ )
  {
    if ( s%8==0 ) printf( "\n      " );
    printf( "%4d=", s );
    for( b=0; b<B; b++ )
      banks[b] = 0;
    for( r=R-1; r>=0; r-- )
      banks[sym_tab[s][r] - '0']++;
    cost = 0;
    for( b=0; b<B; b++ )
      cost += abs(banks[b] - R/2);
    cost /= B;
    printf( "%d", cost );
    if ( s<S-1 ) printf( "," );
  }
  printf( "\n    /\n" );

  for( b=B-1; b>0; b-- )
```

```
    printf( "  BANK_IS_%d(S, B) Is one for bank %d\n", b, b );
  printf( "  ;\n\n" );

  for( b=B-1; b>0; b-- )
    printf( "  BANK_IS_%d(S, B) = 1 $ (BITS(S, B) EQ %d) ;\n", b, b );

  free( banks );
  free( sym_cost );
  for( s=0; s<S; s++ )
    free( sym_tab[s] );
  free( sym_tab );
}

void print_trailer( void )
{
  int b;

  printf( "\n***********************************************************\n"
          "* Decision variables (variables)\n"
          "***********************************************************\n"
          "VARIABLES\n"
          "  W_X(I, S) Write at iteration I is assigned symbol S\n"
          "  R_X(I, S) Read at iteration I is assigned symbol S\n"
          "  W_SYM(S)  Total number of each symbol for writes\n"
          "  R_SYM(S)  Total number of each symbol for reads\n"
          "  W_SYMS    Total write symbols\n"
          "  R_SYMS    Total read symbols\n"
          "  W_COST    Cost of write symbols\n"
          "  R_COST    Cost of read symbols\n"
          "  COST      Total cost\n"
          "  ;\n\n"
          "BINARY VARIABLES W_X, R_X, W_SYM, R_SYM ;\n"
          "INTEGER VARIABLES W_SYMS, R_SYMS ;\n\n"
          "EDGE_EXTS(E, I, J, BI, BJ) = YES $ WI(I, E) $ RI(J, E) $ "
          "BW(BI, E) $ BR(BJ, E) ;\n\n"
          "***********************************************************\n"
          "* Constraints & objective function (Equations)\n"
          "***********************************************************\n"
          "EQUATIONS\n"
          "  CONS1(I)    Allow only one write symbol at iteration I\n"
          "  CONS2(I)    Allow only one read symbol at iteration I\n"
          "  CONS3a(S)   Calculate total number of each symbol for writes\n"
          "  CONS3b(S)   Calculate total number of each symbol for writes\n"
          "  CONS4a(S)   Calculate total number of each symbol for reads\n"
          "  CONS4b(S)   Calculate total number of each symbol for reads\n"
          "  CONS5       Calculate total number of write symbols\n"
          "  CONS6       Calculate total number of read symbols\n"
          "  CONS7       Calculate cost of write symbols\n"
          "  CONS8       Calculate cost of read symbols\n" );

  for ( b=1; b<B; b++ )
    printf( "  CONS%d%s (E, I, J, BI, BJ) Force the bit to be '%d' on "
            "corresponding read of a write\n", b+8, (b==1?" ":""), b );

  printf( "  OBJECT      Our objective (cost) function\n"
          "  ;\n\n"
          "CONS1(I)   .. SUM(S, W_X(I, S)) =E= 1 ;\n"
          "CONS2(I)   .. SUM(S, R_X(I, S)) =E= 1 ;\n"
          "CONS3a(S)  .. 8 * W_SYM(S) - SUM(I, W_X(I, S)) =G= 0;\n"
          "CONS3b(S)  .. SUM(I, W_X(I, S)) - W_SYM(S) =G= 0;\n"
          "CONS4a(S)  .. 8 * R_SYM(S) - SUM(I, R_X(I, S)) =G= 0;\n"
          "CONS4b(S)  .. SUM(I, R_X(I, S)) - R_SYM(S) =G= 0;\n"
          "CONS5      .. W_SYMS =E= SUM(S, W_SYM(S)) ;\n"
          "CONS6      .. R_SYMS =E= SUM(S, R_SYM(S)) ;\n"
          "CONS7      .. W_COST =E= SUM((I, S), W_X(I, S) * SYM_COST(S)) ;\n"
          "CONS8      .. R_COST =E= SUM((I, S), R_X(I, S) * SYM_COST(S)) ;\n\n" );

  for( b=1; b<B; b++ )
  {
    printf( "CONS%d(E, I, J, BI, BJ) $ ( EDGE_EXTS(E, I, J, BI, BJ) ) ..\n", b+8 );
    printf( "  SUM(S, W_X(I, S)*BANK_IS_%d(S, BI)) =E= SUM(S, R_X(J, S)*"
            "BANK_IS_%d(S, BJ)) ;\n\n", b, b );
  }

  printf( "OBJECT .. COST =E= (W_SYMS + W_COST) + (R_SYMS + R_COST) ;\n\n"
```

```
          "*********************************************************\n"
          "*\n"
          "*********************************************************\n"
          "MODEL Banks / ALL / ;\n"
          "OPTIONS LIMROW=10000, LIMCOL=100, RESLIM=90000000, ITERLIM=10000000 ;\n"
          "SOLVE Banks USING MIP MINIMIZING COST ;\n" );
}

void usage( void )
{
  fprintf( stderr, UsageMsg );
  exit( 1 );
}

int main( int argc, char *argv[] )
{
  if ( argc==1 ) {
    fprintf( stderr, "Number of points --> " );
    scanf( "%d", &N );
    fprintf( stderr, "FFT Radix        --> " );
    scanf( "%d", &R );
    fprintf( stderr, "Number of levels --> " );
    scanf( "%d", &L );
    fprintf( stderr, "Memory Banks     --> " );
    scanf( "%d", &B );
  } else if (argc != 5) {
    usage();
  } else {
    N = atoi(argv[1]);
    R = atoi(argv[2]);
    L = atoi(argv[3]);
    B = atoi(argv[4]);
  }

  if (N != (int)pow(R, L)) {
    fprintf( stderr, "Invalid numbers, impossible!\n" );
    exit(1);
  }

  I = (N / R) * L;
  E = I * R;
  S = (int)pow( B, R );

  print_header();
  print_fft_data();
  print_table();
  print_trailer();

  return 0;
}
```

B.2. ILP Source (FFT_16_2.GMS) for 16-point radix-2 FFT, Two Memory Banks

```
$TITLE Assignment of memory banks to variables
$OFFUPPER

*********************************************************
* Copyright (c) 1999 Amal Khailtash
*   (akhailtash@spacebridge.com)
*********************************************************
* 16-point radix-4 FFT with
* 2 levels and 2 memory banks
*********************************************************
* Indices (sets)
*********************************************************
SETS
  I Iterration number / 0 * 7 /
  S Symbol            / 0 * 15 /
  B Bit index         / 0 * 3 /
  E Edge index        / 0 * 31 /
  ;

ALIAS (I, J) ;
ALIAS (B, BI, BJ) ;
```

```
SETS
  WI(I, E) Writer's Iteration number
    /
      0.     (  0,   1,   2,   3 )
      1.     (  4,   5,   6,   7 )
      2.     (  8,   9,  10,  11 )
      3.     ( 12,  13,  14,  15 )
      4.     ( 16,  17,  18,  19 )
      5.     ( 20,  21,  22,  23 )
      6.     ( 24,  25,  26,  27 )
      7.     ( 28,  29,  30,  31 )
    /
  RI(J, E) Reader's Iteration number
    /
      0.     ( 16,  17,  18,  19 )
      1.     ( 20,  21,  22,  23 )
      2.     ( 24,  25,  26,  27 )
      3.     ( 28,  29,  30,  31 )
      4.     (  0,   4,   8,  12 )
      5.     (  1,   5,   9,  13 )
      6.     (  2,   6,  10,  14 )
      7.     (  3,   7,  11,  15 )
    /
  BW(BI, E) Writer's bit number
    /
      0.     (  0,   4,   8,  12,  16,  20,  24,  28 )
      1.     (  1,   5,   9,  13,  17,  21,  25,  29 )
      2.     (  2,   6,  10,  14,  18,  22,  26,  30 )
      3.     (  3,   7,  11,  15,  19,  23,  27,  31 )
    /
  BR(BJ, E) Reader's bit number
    /
      0.     (  0,   1,   2,   3,  16,  20,  24,  28 )
      1.     (  4,   5,   6,   7,  17,  21,  25,  29 )
      2.     (  8,   9,  10,  11,  18,  22,  26,  30 )
      3.     ( 12,  13,  14,  15,  19,  23,  27,  31 )
    /
  EDGE_EXTS(E, I, J, BI, BJ) Edge exists
  ;

*************************************************************
* Given data (parameters, tables, scalars)
*************************************************************
TABLE
  BITS(S, B) binary equivalents of symbol S
          3    2    1    0
     0    0    0    0    0
     1    0    0    0    1
     2    0    0    1    0
     3    0    0    1    1
     4    0    1    0    0
     5    0    1    0    1
     6    0    1    1    0
     7    0    1    1    1
     8    1    0    0    0
     9    1    0    0    1
    10    1    0    1    0
    11    1    0    1    1
    12    1    1    0    0
    13    1    1    0    1
    14    1    1    1    0
    15    1    1    1    1
  ;

PARAMETERS
  SYM_COST(S) Cost of each symbol
    /
        0=2,   1=1,   2=1,   3=0,   4=1,   5=0,   6=0,   7=1,
        8=1,   9=0,  10=0,  11=1,  12=0,  13=1,  14=1,  15=2
    /
  BANK_IS_1(S, B) Is one for bank 1
  ;

  BANK_IS_1(S, B) = 1 $ (BITS(S, B) EQ 1) ;
```

```
***********************************************************
* Decision variables (variables)
***********************************************************
VARIABLES
  W_X(I, S) Write at iteration I is assigned symbol S
  R_X(I, S) Read at iteration I is assigned symbol S
  W_SYM(S)  Total number of each symbol for writes
  R_SYM(S)  Total number of each symbol for reads
  W_SYMS    Total write symbols
  R_SYMS    Total read symbols
  W_COST    Cost of write symbols
  R_COST    Cost of read symbols
  COST      Total cost
  ;

BINARY VARIABLES W_X, R_X, W_SYM, R_SYM ;
INTEGER VARIABLES W_SYMS, R_SYMS ;

EDGE_EXTS(E, I, J, BI, BJ) = YES $ WI(I, E) $ RI(J, E) $ BW(BI, E) $ BR(BJ, E) ;

***********************************************************
* Constraints & objective function (Equations)
***********************************************************
EQUATIONS
  CONS1(I)    Allow only one write symbol at iteration I
  CONS2(I)    Allow only one read symbol at iteration I
  CONS3a(S)   Calculate total number of each symbol for writes
  CONS3b(S)   Calculate total number of each symbol for writes
  CONS4a(S)   Calculate total number of each symbol for reads
  CONS4b(S)   Calculate total number of each symbol for reads
  CONS5       Calculate total number of write symbols
  CONS6       Calculate total number of read symbols
  CONS7       Calculate cost of write symbols
  CONS8       Calculate cost of read symbols
  CONS9  (E, I, J, BI, BJ) Force the bit to be '1' on corresponding read of a write
  OBJECT      Our objective (cost) function
  ;

CONS1(I)   .. SUM(S, W_X(I, S)) =E= 1 ;
CONS2(I)   .. SUM(S, R_X(I, S)) =E= 1 ;
CONS3a(S)  .. 8 * W_SYM(S) - SUM(I, W_X(I, S)) =G= 0;
CONS3b(S)  .. SUM(I, W_X(I, S)) - W_SYM(S) =G= 0;
CONS4a(S)  .. 8 * R_SYM(S) - SUM(I, R_X(I, S)) =G= 0;
CONS4b(S)  .. SUM(I, R_X(I, S)) - R_SYM(S) =G= 0;
CONS5      .. W_SYMS =E= SUM(S, W_SYM(S)) ;
CONS6      .. R_SYMS =E= SUM(S, R_SYM(S)) ;
CONS7      .. W_COST =E= SUM((I, S), W_X(I, S) * SYM_COST(S)) ;
CONS8      .. R_COST =E= SUM((I, S), R_X(I, S) * SYM_COST(S)) ;

CONS9(E, I, J, BI, BJ) $ ( EDGE_EXTS(E, I, J, BI, BJ) ) ..
  SUM(S, W_X(I, S)*BANK_IS_1(S, BI)) =E= SUM(S, R_X(J, S)*BANK_IS_1(S, BJ)) ;

OBJECT .. COST =E= (W_SYMS + W_COST) + (R_SYMS + R_COST) ;

***********************************************************
*
***********************************************************
MODEL Banks / ALL / ;
OPTIONS LIMROW=10000, LIMCOL=100, RESLIM=90000000, ITERLIM=10000000 ;
SOLVE Banks USING MIP MINIMIZING COST ;
```

C. Program to Generate FFT Twiddle Factors

The program is compiled using the Microsoft Visual C++ v5.0.

C.1. C Source Program TWIDDLE.C

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

//#define DEBUG

char *Copyright = "Twiddle Factor VHDL Generator for radix-2 FFT\n"
                  "Copyright (c) 1999 Amal Khailtash (akhailtash@spacebridge.com)\n\n";

char *UsageMsg  = "Usage: TWIDDLE <n>\n"
                  "       n: number of FFT points (power of 2)\n";

#define pi (acos(-1))

void usage( void )
{
  fprintf( stderr, UsageMsg );
  exit( 1 );
}

int main( int argc, char *argv[] )
{
  int    k, n, m;
  double w_real, w_imag;
  int    w_real_scaled, w_imag_scaled;
  int    *wr, *wi;

  fprintf( stderr, Copyright );

  if ( argc!=2 )
    usage();

  n = atoi( argv[1] );
  if ( (log(n)/log(2)) != (int)(log(n)/log(2)) )
    usage();

  m = n / 2;
  wr = malloc( m * sizeof(int) );
  wi = malloc( m * sizeof(int) );

  for( k=0; k<m; k++ ) {
    w_real        =  cos( 2*k*pi/n );
    w_imag        = -sin( 2*k*pi/n );
    w_real_scaled = (int)(127 * w_real);
    w_imag_scaled = (int)(127 * w_imag);
    wr[k]         = w_real_scaled & 0xFF;
    wi[k]         = w_imag_scaled & 0xFF;

#ifdef DEBUG
    printf( "%02d: w_r: %9.6f (%4d) (%02X)\tw_i: %9.6f (%4d) (%02X)\n", k,
            w_real, w_real_scaled, wr[k],
            w_imag, w_imag_scaled, wi[k] );
#endif
  }

  printf( "  ---------------------------------------------------------------------
\n" );
  printf( "  -- Constant Twiddle Factors\n" );
  printf( "  ---------------------------------------------------------------------
\n" );
  printf( "  type LookupTable is array(0 to %d) of std_logic_vector(7 downto 0);\n", m-1
);
  printf( "  constant WR : LookupTable := (\n" );
```

```
  for( k=0; k<m; k++ ) {
    if ( k%8==0 ) printf( "    " );
    printf( "X\"%02X\"", wr[k] );
    if ( k<m-1 ) {
      printf( ", " );
      if ( (k+1)%8==0 ) printf( "\n" );
    }
  }
  printf( "\n  );\n\n" );
  printf( "  constant WI : LookupTable := (\n" );
  for( k=0; k<m; k++ ) {
    if ( k%8==0 ) printf( "    " );
    printf( "X\"%02X\"", wi[k] );
    if ( k<m-1 ) {
      printf( ", " );
      if ( (k+1)%8==0 ) printf( "\n" );
    }
  }
  printf( "\n  );\n\n" );

  free( wr );
  free( wi );

  return 0;
}
```

C.2. Sample Output of the Program for a 256-point FFT

```
Twiddle Factor VHDL Generator for radix-2 FFT
Copyright (c) 1999 Amal Khailtash (akhailtash@spacebridge.com)

  ----------------------------------------------------------------------
  -- Constant Twiddle Factors
  ----------------------------------------------------------------------
  type LookupTable is array(0 to 127) of std_logic_vector(7 downto 0);
  constant WR : LookupTable := (
    X"7F", X"7E", X"7E", X"7E", X"7E", X"7E", X"7D", X"7D",
    X"7C", X"7B", X"7B", X"7A", X"79", X"78", X"77", X"76",
    X"75", X"74", X"72", X"71", X"70", X"6E", X"6C", X"6B",
    X"69", X"67", X"66", X"64", X"62", X"60", X"5E", X"5B",
    X"59", X"57", X"55", X"52", X"50", X"4E", X"4B", X"49",
    X"46", X"43", X"41", X"3E", X"3B", X"39", X"36", X"33",
    X"30", X"2D", X"2A", X"27", X"24", X"21", X"1E", X"1B",
    X"18", X"15", X"12", X"0F", X"0C", X"09", X"06", X"03",
    X"00", X"FD", X"FA", X"F7", X"F4", X"F1", X"EE", X"EB",
    X"E8", X"E5", X"E2", X"DF", X"DC", X"D9", X"D6", X"D3",
    X"D0", X"CD", X"CA", X"C7", X"C5", X"C2", X"BF", X"BD",
    X"BA", X"B7", X"B5", X"B2", X"B0", X"AE", X"AB", X"A9",
    X"A7", X"A5", X"A2", X"A0", X"9E", X"9C", X"9A", X"99",
    X"97", X"95", X"94", X"92", X"90", X"8F", X"8E", X"8C",
    X"8B", X"8A", X"89", X"88", X"87", X"86", X"85", X"85",
    X"84", X"83", X"83", X"82", X"82", X"82", X"82", X"82"
  );

  constant WI : LookupTable := (
    X"00", X"FD", X"FA", X"F7", X"F4", X"F1", X"EE", X"EB",
    X"E8", X"E5", X"E2", X"DF", X"DC", X"D9", X"D6", X"D3",
    X"D0", X"CD", X"CA", X"C7", X"C5", X"C2", X"BF", X"BD",
    X"BA", X"B7", X"B5", X"B2", X"B0", X"AE", X"AB", X"A9",
    X"A7", X"A5", X"A2", X"A0", X"9E", X"9C", X"9A", X"99",
    X"97", X"95", X"94", X"92", X"90", X"8F", X"8E", X"8C",
    X"8B", X"8A", X"89", X"88", X"87", X"86", X"85", X"85",
    X"84", X"83", X"83", X"82", X"82", X"82", X"82", X"82",
    X"81", X"82", X"82", X"82", X"82", X"82", X"83", X"83",
    X"84", X"85", X"85", X"86", X"87", X"88", X"89", X"8A",
    X"8B", X"8C", X"8E", X"8F", X"90", X"92", X"94", X"95",
    X"97", X"99", X"9A", X"9C", X"9E", X"A0", X"A2", X"A5",
    X"A7", X"A9", X"AB", X"AE", X"B0", X"B2", X"B5", X"B7",
    X"BA", X"BD", X"BF", X"C2", X"C5", X"C7", X"CA", X"CD",
    X"D0", X"D3", X"D6", X"D9", X"DC", X"DF", X"E2", X"E5",
    X"E8", X"EB", X"EE", X"F1", X"F4", X"F7", X"FA", X"FD"
  );
```

D. C Source File Used to Design a Hardware Address Generator

The program is compiled using the Microsoft Visual C++ v5.0.

D.1. C Source File ADDGEN.C

```
#include <conio.h>
#include <math.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TEST_1
//#define TEST_2
//#define TEST_3
//#define TEST_4

#ifdef TEST_1
#  define SIZE  (256*256)
#endif
#ifdef TEST_2
#  define SIZE  (256*256)
#endif
#ifdef TEST_3
#  define SIZE  (12)
#endif

#define FALSE (0==1)
#define TRUE  (1==1)

typedef unsigned char BYTE;

// -------------------------------------------------------------------
#ifdef TEST_4
#define SAMPLES   8
#define POWER     ((double)log10((double)SAMPLES)/log10((double)2.0))

#define SIZE SAMPLES
int address1[SIZE*SIZE];
int address2[SIZE*SIZE];
int address3[SIZE*SIZE];
int address4[SIZE*SIZE];
#endif
// -------------------------------------------------------------------

int address[SIZE];
BYTE add_bit[SIZE];

#ifdef TEST_1
void gen_addresses1( void )
{
  int x, y, i, j, X, Y;

  j = 0;
  for( Y=0; Y<65536; Y+=4096 )            // block height = 16 rows
  {
    for( X=0; X<256; X+=16 )              // block width = 16 columns
    {
      for( i=0; i<4; i++ )                // do 4 times
      {
        for( y=0; y<4096; y+=512 )        // every 2nd line
        {
          for( x=(y/512)%2; x<16; x+=2 )  // every 2nd pixel
          {
//            printf( "Y=%d\tX=%d\ti=%d\ty=%d\tx=%d", Y, X, i, y, x );
            address[j] = x + y + X + Y;
//            printf( "\tj=%d, add=%d\n", j, address[j] );
            j++;
//            getch();
```

```
            }
          }
        }
      }
    }
}
#endif

#ifdef TEST_2
void gen_addresses2( void )
{
    int x, y, j, X, Y;
    int rand[] = { 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1 };

    j = 0;
    for( Y=0; Y<65536; Y+=4096 )          //
    {
      for( X=0; X<256; X+=16 )            //
      {
        for( y=0; y<4096; y+=256 )        //
        {
          for( x=0; x<16; x++ )           //
          {
            address[j] = rand[x]*x + y + X + Y;
            j++;
          }
        }
      }
    }
}
#endif

void print_addresses( void )
{
    int i;

    for( i=0; i<SIZE; i++ )
    {
      if ( i%12==0 )
        printf( "%07d:", i );

      if ( (i+1)%(12*25)==0 ) {
//        printf( "\n" );
        if ( getch()==27 ) break;
      }
      printf( "%6d", address[i] );
    }
    printf( "\n" );
}

void get_address_bits( int bit )
{
    int i;

    for( i=0; i<SIZE; i++ )
    {
      add_bit[i] = (address[i] >> bit) & 0x1;
    }
}

int bits_equal( int *bit, int last )
{
    int i;
    int equal;

    *bit = add_bit[0];
    equal = TRUE;
    for( i=1; i<last; i++ )
    {
      if ( add_bit[i-1]!=add_bit[i] ) {
        *bit = -1;
        equal = FALSE;
        break;
      }
    }
```

```
  return equal;
}

int halves_equal( int first, int last )
{
  int i;
  int equal;

  equal = TRUE;
  for( i=first; i<last; i++ )
  {
    if ( add_bit[i]!=add_bit[last+i] ) {
      equal = FALSE;
      break;
    }
  }

  return equal;
}

int halves_inverse( int first, int last )
{
  int i;
  int inverse;

  inverse = TRUE;
  for( i=first; i<last; i++ )
  {
    if ( (add_bit[i] + add_bit[last+i])!=1 ) {
      inverse = FALSE;
      break;
    }
  }

  return inverse;
}

void semi_random_sequence( BYTE *list, int size, char *mapping )
{
  int i, j, c;
  char buffer[1024];
  int first1;
  int first2;

//  printf( "Semi-random Sequence...\n" );

  mapping[0] = '\0';
  first1 = TRUE;
  c = 0;
  for( i=0; i<size; i++ ) {
    if ( list[i]==1 ) {
      c++;
//      printf( "list[%d]=%d\n", i, list[i] );

      if ( !first1 ) strcat( mapping, " +\n     " );
      first2 = TRUE;
      for( j=0; j<(int)(log10(size)/log10(2)); j++ ) {
        if ( !first2 ) strcat( mapping, "." );
        sprintf( buffer, "C[%d]", j );
        strcat( mapping, buffer );
        if ( (i & (0x1 << j))==0 ) strcat( mapping, "^" );
        if ( first2 ) first2 = FALSE;
      }
      if ( first1 ) first1 = FALSE;
    }
  }
//  printf( "%d\n", c );
}

void synth_address( BYTE *list, int size, char *mapping )
{
  int bit, last, m;
//  int bit, equal, last;
  char new_mapping[10*1024];
```

```
//  printf( "Synth Address...\n" );
  last = size;
  do {
    if ( bits_equal( &bit, last ) ) {
      sprintf( mapping, "%d", bit );
      return;
    }
    last /= 2;
  } while( halves_equal( 0, last ) );
//  } while( halves_equal( 0, last ) && last>0 );

  if ( halves_inverse( 0, last ) ) {
    m = (int)(log10(last)/log10(2));
    if ( halves_equal( 0, last/2 ) ) {
      sprintf( mapping, "%sC[%d]", (list[0]==0)?"":"not ", m );
    } else {
      synth_address( list, last, new_mapping );
      sprintf( mapping, "C[%d] xor (%s)", m, new_mapping );
    }
  } else {
    semi_random_sequence( list, last*2, new_mapping );
//    sprintf( mapping, "???" );
    sprintf( mapping, new_mapping );
  }
}

/*********************************************************************
 *
 *********************************************************************/
void trace( char *s, ... )
{
#ifdef _DEBUG
  va_list args;

  va_start( args, s );
  vprintf( s, args );
  va_end( args );
#endif
}

#ifdef TEST_4
/*********************************************************************
 * Bit reverse the number
 * Change 11100000b to 00000111b or vice-versa
 *********************************************************************/
int permute( int index )
{
  int n1, result, loop;

  n1 = SAMPLES;
  result = 0;

  for( loop=0; loop<POWER; loop++ )
  {
    n1 >>= 1;      /* n1 / 2.0 */
    if (index < n1)
      continue;

    result += (int)pow( (double)2.0, (double)loop );
    index -= n1;
  }

  return result;
}

/*********************************************************************
 *
 *********************************************************************/
void fft_dif()
{
  int l, i, j, k;
  int m, n, o, p;
  int x;
//  double w;
```

```
//  double z1, w1, z2, w2;

  x = 0;
  m = SAMPLES / 2;
  n = 1;

  for( l=0; l<POWER; l++ )
  {
    o = 0;
    p = m;

    for( i=0; i<n; i++ )
    {
      for( j=o; j<p; j++ )
      {
        k = (j - o) * permute(m);
        if ( l<POWER-1 )
        {
          address1[x] = j;
          address2[x] = j+m;
          address3[x] = j;
          address4[x] = j+m;
          trace( "%d: (%d %d) -> (%d %d)\n", k, j, j+m, j, j+m );
        } else {
          address1[x] = j;
          address2[x] = j+m;
          address3[x] = permute(j);
          address4[x] = permute(j+m);
          trace( "%d: (%d %d) -> (%d %d)\n", k, j, j+m, permute(j), permute(j+m) );
        }
        x++;
      }
      trace( "\n" );
      o += (m * 2);
      p += (m * 2);
    }
    m /= 2;
    n *= 2;
  }
}
#endif

int main( int argc, char *argv[] )
{
  int i;
  char transform[1024];

#ifdef TEST_1
  gen_addresses1();
//  print_addresses();
  for( i=0; i<16; i++ )
  {
    get_address_bits( i );
    synth_address( add_bit, SIZE, transform );
    printf( "    '%s' \t==> adbit %d\n", transform, i );
  }
  printf( "\n" );
#endif

#ifdef TEST_2
  gen_addresses2();
  for( i=0; i<16; i++ )
  {
    get_address_bits( i );
    synth_address( add_bit, SIZE, transform );
    printf( "    '%s' \t==> adbit %d\n", transform, i );
  }

  printf( "\n" );
#endif

#ifdef TEST_3
  address[ 0] = 0;  address[ 1] = 2;  address[ 2] = 1;  address[ 3] = 3;
  address[ 4] = 0;  address[ 5] = 2;  address[ 6] = 4;  address[ 7] = 6;
  address[ 8] = 0;  address[ 9] = 2;  address[10] = 4;  address[11] = 6;
```

```
  for( i=0; i<3; i++ )
  {
    get_address_bits( i );
    synth_address( add_bit, 8, transform );
    printf( "    '%s' \t==> adbit %d\n", transform, i );
  }
#endif

#ifdef TEST_4
  //-----------------------------------------------------------------
  fft_dif();
//  printf( "-\n" );
  for( i=0; i<SAMPLES; i++ )
    address[i] = address1[i];
//  printf( "--\n" );
  for( i=0; i<16; i++ )
  {
    get_address_bits( i );
//     printf( "---\n" );
    synth_address( add_bit, SIZE, transform );
    printf( "    '%s' \t==> adbit %d\n", transform, i );
  }
#endif

  return 0;
}
```

D.2. Sample #1

```
'C[3]'      ==> adbit 0
'C[0]'      ==> adbit 1
'C[1]'      ==> adbit 2
'C[2]'      ==> adbit 3
'C[8]'      ==> adbit 4
'C[9]'      ==> adbit 5
'C[10]'     ==> adbit 6
'C[11]'     ==> adbit 7
'0'         ==> adbit 8
'C[3]'      ==> adbit 9
'C[4]'      ==> adbit 10
'C[5]'      ==> adbit 11
'C[12]'     ==> adbit 12
'C[13]'     ==> adbit 13
'C[14]'     ==> adbit 14
'C[15]'     ==> adbit 15
```

D.2. Sample #2

```
'C[0].C[1]^.C[2].C[3]^ +
 C[0].C[1]^.C[2].C[3] +
 C[0].C[1].C[2].C[3]'          ==> adbit 0
'C[0]^.C[1].C[2]^.C[3]^ +
 C[0]^.C[1].C[2]^.C[3] +
 C[0].C[1].C[2].C[3]'          ==> adbit 1
'C[0]^.C[1]^.C[2].C[3]^ +
 C[0].C[1]^.C[2].C[3]^ +
 C[0]^.C[1]^.C[2].C[3] +
 C[0].C[1]^.C[2].C[3] +
 C[0].C[1].C[2].C[3]'          ==> adbit 2
'C[0]^.C[1].C[2]^.C[3] +
 C[0]^.C[1]^.C[2].C[3] +
 C[0].C[1]^.C[2].C[3] +
 C[0].C[1].C[2].C[3]'          ==> adbit 3
'C[8]'                         ==> adbit 4
'C[9]'                         ==> adbit 5
'C[10]'                        ==> adbit 6
'C[11]'                        ==> adbit 7
'C[4]'                         ==> adbit 8
'C[5]'                         ==> adbit 9
'C[6]'                         ==> adbit 10
'C[7]'                         ==> adbit 11
'C[12]'                        ==> adbit 12
'C[13]'                        ==> adbit 13
'C[14]'                        ==> adbit 14
```

```
'C[15]'                              ==> adbit 15
```

D.2. Sample #3

```
'C[0]^.C[1].C[2]^ +
 C[0].C[1].C[2]^'    ==> adbit 0
'C[0]'               ==> adbit 1
'C[0]^.C[1].C[2] +
 C[0].C[1].C[2]'     ==> adbit 2
```

D.2. Sample #4

```
'C[0]'               ==> adbit 0
'C[0]^.C[1].C[2]^ +
 C[0].C[1].C[2]^'    ==> adbit 1
'C[0]^.C[1].C[2] +
 C[0].C[1].C[2]'     ==> adbit 2
'0'                  ==> adbit 3
'0'                  ==> adbit 4
'0'                  ==> adbit 5
'0'                  ==> adbit 6
'0'                  ==> adbit 7
'0'                  ==> adbit 8
'0'                  ==> adbit 9
'0'                  ==> adbit 10
'0'                  ==> adbit 11
'0'                  ==> adbit 12
'0'                  ==> adbit 13
'0'                  ==> adbit 14
'0'                  ==> adbit 15
```

D. VHDL Source Files for 1024-point Complex FFT

```
-------------------------------------------------------------------------------
-- Author     : Amal Khailtash
-- Design     : 1024-point Complex FFT
-- ---------------------------------------------------------------------------
-- >>>>>>>>>>>>>>>>>>>>>>>>>>>> COPYRIGHT NOTICE <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
-- ---------------------------------------------------------------------------
-- Copyright 1999 (c) Amal Khailtash
--
-- This is a public domain code.  You may use or modify it as long as you
-- mention my name as the original author.
-- ---------------------------------------------------------------------------
-- >>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Warrantee <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
-- ---------------------------------------------------------------------------
-- I MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE USE OF THIS CODE, EITHER
-- EXPRESSED OR IMPLIED.  I ALSO TAKE NO RESPONSIBILITY OF ANY DAMAGES, THE USE
-- OF THIS CODE MAY CAUSE.
-- ---------------------------------------------------------------------------
```

D.1. FFT Component Hierarchy

```
+- CFFT1024
   |
   +- addrgen_bitrev
   |
   +- addrgen_linear
   |
   +- butterfly
   |  |
   |  +- mult
   |  |  |
   |  |  +- reg_pipe_single
   |  |
   |  +- reg_pipe
   |
   +- controller
   |  |
   |  +- reg_pipe_single
   |
   +- mem_bank
   |
   +- skew_buffer
   |
   +- twiddle_factors
```

D.2. addrgen_bitrev.vhd

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

entity addrgen_bitrev is
  generic ( WIDTH : positive );
  port (
    reset_n : in  std_logic;
    clock   : in  std_logic;
    enable  : in  std_logic;

    addr    : out std_logic_vector(WIDTH-1 downto 0)
  );
end entity addrgen_bitrev;

architecture rtl of addrgen_bitrev is

  -- -------------------------------------------------------------------------
  -- - Registered Signals
  -- -------------------------------------------------------------------------
```

```
  signal addr_int : std_logic_vector(WIDTH-1 downto 0);

begin

  -- ********************************************************************
  -- * Combinational Assignments
  -- ********************************************************************
  g_bit_rev: for i in addr_int'range generate
    addr(i) <= addr_int(addr'length-i-1);
  end generate;

  -- ********************************************************************
  -- *
  -- ********************************************************************
  sync: process( reset_n, clock )
  begin
    if ( reset_n='0' ) then
      addr_int <= (others=>'0');
    elsif ( rising_edge(clock) ) then
      if ( enable='1' ) then
        addr_int <= addr_int + '1';
      end if;
    end if;
  end process sync;

end architecture rtl;
```

D.3. addrgen_linear.vhd

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

entity addrgen_linear is
  generic ( WIDTH : positive );
  port (
    reset_n : in  std_logic;
    clock   : in  std_logic;
    enable  : in  std_logic;

    addr    : out std_logic_vector(WIDTH-1 downto 0)
  );
end entity addrgen_linear;

architecture rtl of addrgen_linear is

  -- ---------------------------------------------------------------------------
  -- - Registered Signals
  -- ---------------------------------------------------------------------------
  signal addr_int : std_logic_vector(WIDTH-1 downto 0);

begin

  -- ********************************************************************
  -- * Combinational Assignments
  -- ********************************************************************
  addr <= addr_int;

  -- ********************************************************************
  -- *
  -- ********************************************************************
  sync: process( reset_n, clock )
  begin
    if ( reset_n='0' ) then
      addr_int <= (others=>'0');
    elsif ( rising_edge(clock) ) then
      if ( enable='1' ) then
        addr_int <= addr_int + '1';
      end if;
    end if;
  end process sync;
```

```
end architecture rtl;
```

D.4. butterfly.vhd

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

entity butterfly is
  port (
    reset_n : in  std_logic;
    clock   : in  std_logic;
    enable  : in  std_logic;

    w_r     : in  std_logic_vector( 7 downto 0);
    w_i     : in  std_logic_vector( 7 downto 0);

    a_r     : in  std_logic_vector(15 downto 0);
    a_i     : in  std_logic_vector(15 downto 0);
    b_r     : in  std_logic_vector(15 downto 0);
    b_i     : in  std_logic_vector(15 downto 0);

    x_r     : out std_logic_vector(15 downto 0);
    x_i     : out std_logic_vector(15 downto 0);
    y_r     : out std_logic_vector(15 downto 0);
    y_i     : out std_logic_vector(15 downto 0)
  );
end entity butterfly;

architecture rtl of butterfly is

  -- ---------------------------------------------------------------------------
  -- - Componenet Declarations
  -- ---------------------------------------------------------------------------
  component reg_pipe
    generic (
      DEPTH : positive;
      WIDTH : positive
    );
    port (
      reset_n : in  std_logic;
      clock   : in  std_logic;
      enable  : in  std_logic;
      i       : in  std_logic_vector(WIDTH-1 downto 0);
      o       : out std_logic_vector(WIDTH-1 downto 0)
    );
  end component;

  component mult
    generic (
      A_WIDTH : positive;
      B_WIDTH : positive
    );
    port (
      reset_n : in  std_logic;
      clock   : in  std_logic;
      enable  : in  std_logic;
      a       : in  std_logic_vector(A_WIDTH-1           downto 0);
      b       : in  std_logic_vector(B_WIDTH-1           downto 0);
      p       : out std_logic_vector((A_WIDTH+B_WIDTH-1) downto 0)
    );
  end component;

  -- ---------------------------------------------------------------------------
  -- - Registered Signals
  -- ---------------------------------------------------------------------------
  signal ar_plus_br  : std_logic_vector(16 downto 0);
  signal ai_plus_bi  : std_logic_vector(16 downto 0);
  signal ar_minus_br : std_logic_vector(16 downto 0);
  signal ai_minus_bi : std_logic_vector(16 downto 0);
```

```
  signal p0           : std_logic_vector(24 downto 0);
  signal p1           : std_logic_vector(24 downto 0);
  signal p2           : std_logic_vector(24 downto 0);
  signal p3           : std_logic_vector(24 downto 0);

  signal p0_minus_p1  : std_logic_vector(25 downto 0);
  signal p2_plus_p3   : std_logic_vector(25 downto 0);

  signal w_r_del      : std_logic_vector( 7 downto 0);
  signal w_i_del      : std_logic_vector( 7 downto 0);

begin

  -- -------------------------------------------------------------------------
  -- The following should be calculated:
  --   x_r = (a_r + b_r);
  --   x_i = (a_i + b_i);
  --   y_r = (a_r - b_r)*W_r(k) - (a_i - b_i)*W_i(k)
  --   y_r = (a_i - b_i)*W_r(k) + (a_r - b_r)*W_i(k)
  -- -------------------------------------------------------------------------

  -- ************************************************************************
  -- * Component Instantiations
  -- ************************************************************************
  i_pipe0: reg_pipe
    generic map( DEPTH=>9, WIDTH=>16 )
    port map(
      reset_n => reset_n,
      clock   => clock,
      enable  => enable,
      i       => ar_plus_br(15 downto 0),
      o       => x_r
    );

  i_pipe1: reg_pipe
    generic map( DEPTH=>9, WIDTH=>16 )
    port map(
      reset_n => reset_n,
      clock   => clock,
      enable  => enable,
      i       => ai_plus_bi(15 downto 0),
      o       => x_i
    );

  i_mult0: mult
    generic map( A_WIDTH=>17, B_WIDTH=>8 )
    port map(
      reset_n => reset_n,
      clock   => clock,
      enable  => enable,
      a       => ar_minus_br,
      b       => w_r_del,
      p       => p0
    );

  i_mult1: mult
    generic map( A_WIDTH=>17, B_WIDTH=>8 )
    port map(
      reset_n => reset_n,
      clock   => clock,
      enable  => enable,
      a       => ai_minus_bi,
      b       => w_i_del,
      p       => p1
    );

  i_mult2: mult
    generic map( A_WIDTH=>17, B_WIDTH=>8 )
    port map(
      reset_n => reset_n,
      clock   => clock,
      enable  => enable,
      a       => ai_minus_bi,
      b       => w_r_del,
```

```
      p        => p2
    );

  i_mult3: mult
    generic map( A_WIDTH=>17, B_WIDTH=>8 )
    port map(
      reset_n => reset_n,
      clock   => clock,
      enable  => enable,
      a       => ar_minus_br,
      b       => w_i_del,
      p       => p3
    );

  -- ************************************************************************
  -- *
  -- ************************************************************************
  process( reset_n, clock )
  begin
    if ( reset_n='0' ) then
      ar_plus_br  <= (others=>'0');
      ai_plus_bi  <= (others=>'0');
      ar_minus_br <= (others=>'0');
      ai_minus_bi <= (others=>'0');
      w_r_del     <= (others=>'0');
      w_i_del     <= (others=>'0');
      p0_minus_p1 <= (others=>'0');
      p2_plus_p3  <= (others=>'0');
      y_r         <= (others=>'0');
      y_i         <= (others=>'0');
    elsif ( rising_edge(clock) ) then
      if ( enable='1' ) then
        ar_plus_br  <= sxt( a_r, 17 ) + sxt( b_r, 17 );
        ai_plus_bi  <= sxt( a_i, 17 ) + sxt( b_i, 17 );

        ar_minus_br <= sxt( a_r, 17 ) - sxt( b_r, 17 );
        ai_minus_bi <= sxt( a_i, 17 ) - sxt( b_i, 17 );

        w_r_del     <= w_r;
        w_i_del     <= w_i;

        p0_minus_p1 <= sxt( p0, 26) - sxt( p1, 26 );
        p2_plus_p3  <= sxt( p2, 26) + sxt( p3, 26 );

        y_r         <= p0_minus_p1(25 downto 10);
        y_i         <= p2_plus_p3 (25 downto 10);
      end if;
    end if;
  end process;

end architecture rtl;
```

D.5. cfft1024.vhd

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

entity cfft1024 is
  port (
    reset_n      : in  std_logic;
    clock        : in  std_logic;
    enable       : in  std_logic;

    start        : in  std_logic;
    busy         : out std_logic;
    done         : out std_logic;

    data_in      : in  std_logic_vector( 7 downto 0);
    data_out     : out std_logic_vector(15 downto 0)
  );
```

```
end entity cfft1024;

architecture rtl of cfft1024 is

  -- ------------------------------------------------------------------------
  -- - Componenet Declarations
  -- ------------------------------------------------------------------------
  component addrgen_bitrev
    generic ( WIDTH : positive );
    port (
      reset_n : in  std_logic;
      clock   : in  std_logic;
      enable  : in  std_logic;
      addr    : out std_logic_vector(WIDTH-1 downto 0)
    );
  end component addrgen_bitrev;

  component addrgen_linear
    generic ( WIDTH : positive );
    port (
      reset_n : in  std_logic;
      clock   : in  std_logic;
      enable  : in  std_logic;
      addr    : out std_logic_vector(WIDTH-1 downto 0)
    );
  end component addrgen_linear;

  component butterfly
    port (
      reset_n : in  std_logic;
      clock   : in  std_logic;
      enable  : in  std_logic;
      w_r     : in  std_logic_vector( 7 downto 0);
      w_i     : in  std_logic_vector( 7 downto 0);
      a_r     : in  std_logic_vector(15 downto 0);
      a_i     : in  std_logic_vector(15 downto 0);
      b_r     : in  std_logic_vector(15 downto 0);
      b_i     : in  std_logic_vector(15 downto 0);
      x_r     : out std_logic_vector(15 downto 0);
      x_i     : out std_logic_vector(15 downto 0);
      y_r     : out std_logic_vector(15 downto 0);
      y_i     : out std_logic_vector(15 downto 0)
    );
  end component butterfly;

  component controller
    port (
      reset_n           : in  std_logic;
      clock             : in  std_logic;
      enable            : in  std_logic;
      start             : in  std_logic;
      busy              : out std_logic;
      done              : out std_logic;
      engine_enable     : out std_logic;
      k                 : out std_logic_vector(8 downto 0);
      bank0r_we         : out std_logic;
      bank0i_we         : out std_logic;
      bank1r_we         : out std_logic;
      bank1i_we         : out std_logic;
      enable_w_addrgen  : out std_logic;
      enable_r_addrgen0 : out std_logic;
      enable_r_addrgen1 : out std_logic;
      select_r_addrgen  : out std_logic;
      write_sel         : out std_logic;
      read_sel          : out std_logic;
      bank_sel          : out std_logic_vector(1 downto 0);
      skew_enable       : out std_logic
    );
  end component controller;

  component mem_bank
    port (
      clock  : in  std_logic;
      we     : in  std_logic;
      w_addr : in  std_logic_vector( 8 downto 0);
```

```
    w_din  : in  std_logic_vector(15 downto 0);
    r_addr : in  std_logic_vector( 8 downto 0);
    r_dout : out std_logic_vector(15 downto 0)
  );
end component mem_bank;

component skew_buffer
  port (
    reset_n : in  std_logic;
    clock   : in  std_logic;
    enable  : in  std_logic;
    din0    : in  std_logic_vector(15 downto 0);
    din1    : in  std_logic_vector(15 downto 0);
    dout0   : out std_logic_vector(15 downto 0);
    dout1   : out std_logic_vector(15 downto 0)
  );
end component skew_buffer;

component twiddle_factors
  port (
    k   : in  std_logic_vector(8 downto 0);
    w_r : out std_logic_vector(7 downto 0);
    w_i : out std_logic_vector(7 downto 0)
  );
end component twiddle_factors;

-- ---------------------------------------------------------------------------
-- - Constants & New Types
-- ---------------------------------------------------------------------------
constant zero8 : std_logic_vector(7 downto 0) := (others=>'0');

-- ---------------------------------------------------------------------------
-- - Registered Signals
-- ---------------------------------------------------------------------------
signal k                 : std_logic_vector( 8 downto 0);

signal engine_enable     : std_logic;
signal w_r               : std_logic_vector( 7 downto 0);
signal w_i               : std_logic_vector( 7 downto 0);
signal a_r               : std_logic_vector(15 downto 0);
signal a_i               : std_logic_vector(15 downto 0);
signal b_r               : std_logic_vector(15 downto 0);
signal b_i               : std_logic_vector(15 downto 0);
signal x_r               : std_logic_vector(15 downto 0);
signal x_i               : std_logic_vector(15 downto 0);
signal y_r               : std_logic_vector(15 downto 0);
signal y_i               : std_logic_vector(15 downto 0);

signal skew_enable       : std_logic;

signal data_real0        : std_logic_vector(15 downto 0);
signal data_real1        : std_logic_vector(15 downto 0);
signal data_imag0        : std_logic_vector(15 downto 0);
signal data_imag1        : std_logic_vector(15 downto 0);

signal bank0r_we         : std_logic;
signal bank0r_w_addr     : std_logic_vector( 8 downto 0);
signal bank0r_w_din      : std_logic_vector(15 downto 0);
signal bank0r_r_addr     : std_logic_vector( 8 downto 0);
signal bank0r_r_dout     : std_logic_vector(15 downto 0);

signal bank0i_we         : std_logic;
signal bank0i_w_addr     : std_logic_vector( 8 downto 0);
signal bank0i_w_din      : std_logic_vector(15 downto 0);
signal bank0i_r_addr     : std_logic_vector( 8 downto 0);
signal bank0i_r_dout     : std_logic_vector(15 downto 0);

signal bank1r_we         : std_logic;
signal bank1r_w_addr     : std_logic_vector( 8 downto 0);
signal bank1r_w_din      : std_logic_vector(15 downto 0);
signal bank1r_r_addr     : std_logic_vector( 8 downto 0);
signal bank1r_r_dout     : std_logic_vector(15 downto 0);

signal bank1i_we         : std_logic;
signal bank1i_w_addr     : std_logic_vector( 8 downto 0);
```

```
  signal bank1i_w_din      : std_logic_vector(15 downto 0);
  signal bank1i_r_addr     : std_logic_vector( 8 downto 0);
  signal bank1i_r_dout     : std_logic_vector(15 downto 0);

  signal write_address     : std_logic_vector( 8 downto 0);
  signal enable_w_addrgen  : std_logic;

  signal read_address      : std_logic_vector( 8 downto 0);
  signal read_address0     : std_logic_vector( 8 downto 0);
  signal read_address1     : std_logic_vector( 8 downto 0);
  signal enable_r_addrgen0 : std_logic;
  signal enable_r_addrgen1 : std_logic;
  signal select_r_addrgen  : std_logic;

  signal write_sel         : std_logic;
  signal read_sel          : std_logic;
  signal bank_sel          : std_logic_vector(1 downto 0);

begin
  -- *******************************************************************
  -- * Component Instantiations
  -- *******************************************************************
  write_addrgen: addrgen_bitrev
    generic map( WIDTH=>9 )
    port map(
      reset_n => reset_n,
      clock   => clock,
      enable  => enable_w_addrgen,
      addr    => write_address
    );

  read_addrgen_lin: addrgen_linear
    generic map( WIDTH=>9 )
    port map(
      reset_n => reset_n,
      clock   => clock,
      enable  => enable_r_addrgen0,
      addr    => read_address0
    );

  read_addrgen_br: addrgen_bitrev
    generic map( WIDTH=>9 )
    port map(
      reset_n => reset_n,
      clock   => clock,
      enable  => enable_r_addrgen1,
      addr    => read_address1
    );

  engine: butterfly
    port map (
      reset_n => reset_n,
      clock   => clock,
      enable  => engine_enable,
      w_r     => w_r,
      w_i     => w_i,
      a_r     => a_r,
      a_i     => a_i,
      b_r     => b_r,
      b_i     => b_i,
      x_r     => x_r,
      x_i     => x_i,
      y_r     => y_r,
      y_i     => y_i
    );

  fft_controller: controller
    port map (
      reset_n             => reset_n,
      clock               => clock,
      enable              => enable,
      start               => start,
      busy                => busy,
      done                => done,
```

```
      k                  => k,
      engine_enable      => engine_enable,
      bank0r_we          => bank0r_we,
      bank0i_we          => bank0i_we,
      bank1r_we          => bank1r_we,
      bank1i_we          => bank1i_we,
      enable_w_addrgen   => enable_w_addrgen,
      enable_r_addrgen0  => enable_r_addrgen0,
      enable_r_addrgen1  => enable_r_addrgen1,
      select_r_addrgen   => select_r_addrgen,
      write_sel          => write_sel,
      read_sel           => read_sel,
      bank_sel           => bank_sel,
      skew_enable        => skew_enable
    );

  skew_buffer_r: skew_buffer
    port map (
      reset_n => reset_n,
      clock   => clock,
      enable  => skew_enable,
      din0    => x_r,
      din1    => y_r,
      dout0   => data_real0,
      dout1   => data_real1
    );

  skew_buffer_i: skew_buffer
    port map (
      reset_n => reset_n,
      clock   => clock,
      enable  => skew_enable,
      din0    => x_i,
      din1    => y_i,
      dout0   => data_imag0,
      dout1   => data_imag1
    );

  bank0r: mem_bank
    port map (
      clock  => clock,
      we     => bank0r_we,
      w_addr => bank0r_w_addr,
      w_din  => bank0r_w_din,
      r_addr => bank0r_r_addr,
      r_dout => bank0r_r_dout
    );

  bank0i: mem_bank
    port map (
      clock  => clock,
      we     => bank0i_we,
      w_addr => bank0i_w_addr,
      w_din  => bank0i_w_din,
      r_addr => bank0i_r_addr,
      r_dout => bank0i_r_dout
    );

  bank1r: mem_bank
    port map (
      clock  => clock,
      we     => bank1r_we,
      w_addr => bank1r_w_addr,
      w_din  => bank1r_w_din,
      r_addr => bank1r_r_addr,
      r_dout => bank1r_r_dout
    );

  bank1i: mem_bank
    port map (
      clock  => clock,
      we     => bank1i_we,
      w_addr => bank1i_w_addr,
      w_din  => bank1i_w_din,
      r_addr => bank1i_r_addr,
```

```
    r_dout => bank1i_r_dout
  );

twiddles: twiddle_factors
  port map (
    k   => k,
    w_r => w_r,
    w_i => w_i
  );

-- ************************************************************************
-- * Combinational Assignments
-- ************************************************************************
a_r           <= bank0r_r_dout;
a_i           <= bank0i_r_dout;
b_r           <= bank1r_r_dout;
b_i           <= bank1i_r_dout;

bank0r_w_addr <= write_address;
bank0i_w_addr <= write_address;
bank1r_w_addr <= write_address;
bank1i_w_addr <= write_address;

read_address  <= read_address0 when ( select_r_addrgen='0' ) else
                 read_address1;

bank0r_r_addr <= read_address;
bank0i_r_addr <= read_address;
bank1r_r_addr <= read_address;
bank1i_r_addr <= read_address;

-- ************************************************************************
-- *
-- ************************************************************************
process( reset_n, clock )
begin
  if ( reset_n='0' ) then
    bank0r_w_din <= (others=>'0');
    bank0i_w_din <= (others=>'0');
    bank1r_w_din <= (others=>'0');
    bank1i_w_din <= (others=>'0');

    data_out     <= (others=>'0');
  elsif ( rising_edge(clock) ) then
    if ( enable='1' ) then

      if ( write_sel='0' ) then
        bank0r_w_din <= (zero8 & data_in);
        bank0i_w_din <= (zero8 & data_in);
        bank1r_w_din <= (zero8 & data_in);
        bank1i_w_din <= (zero8 & data_in);
      else
        bank0r_w_din <= data_real0;
        bank0i_w_din <= data_imag0;
        bank1r_w_din <= data_real1;
        bank1i_w_din <= data_imag1;
      end if;

      data_out <= (others=>'0');
      if ( read_sel='1' ) then
        case bank_sel is
          when "00" =>
            data_out <= bank0r_r_dout;
          when "01" =>
            data_out <= bank0i_r_dout;
          when "10" =>
            data_out <= bank1r_r_dout;
          when "11" =>
            data_out <= bank1i_r_dout;
          when others => null;
        end case;
      end if;

    end if;
  end if;
```

```
  end process;

end architecture rtl;
```

D.6. controller.vhd

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

entity controller is
  port (
    reset_n           : in  std_logic;
    clock             : in  std_logic;
    enable            : in  std_logic;

    start             : in  std_logic;
    busy              : out std_logic;
    done              : out std_logic;

    engine_enable     : out std_logic;
    k                 : out std_logic_vector(8 downto 0);

    bank0r_we         : out std_logic;
    bank0i_we         : out std_logic;
    bank1r_we         : out std_logic;
    bank1i_we         : out std_logic;

    enable_w_addrgen  : out std_logic;
    enable_r_addrgen0 : out std_logic;
    enable_r_addrgen1 : out std_logic;
    select_r_addrgen  : out std_logic;

    write_sel         : out std_logic;
    read_sel          : out std_logic;
    bank_sel          : out std_logic_vector(1 downto 0);

    skew_enable       : out std_logic
  );
end entity controller;

architecture rtl of controller is

  -- -------------------------------------------------------------------------
  -- - Componenet Declarations
  -- -------------------------------------------------------------------------
  component reg_pipe_single
    generic (
      DEPTH : positive
    );
    port (
      reset_n : in  std_logic;
      clock   : in  std_logic;
      enable  : in  std_logic;
      i       : in  std_logic;
      o       : out std_logic
    );
  end component;

  -- -------------------------------------------------------------------------
  -- - Constants & New Types
  -- -------------------------------------------------------------------------
  constant N_POINTS        : integer := 1024;
  constant N_DATA          : integer := N_POINTS*2;
  constant N_DATA_DIV2     : integer := N_DATA/2;
  constant NODES_PER_LEVEL : integer := N_POINTS/2;
--  constant LEVELS          : integer := log2(N_POINTS);
  constant LEVELS          : integer := 10;
  constant NODES           : integer := LEVELS*NODES_PER_LEVEL;

  type ControllerStateType is ( IDLE,
```

```
                            WR_D0R, WR_D0I, WR_D1R, WR_D1I,
                            START_PROCESS, PROCESS_NODE, FLUSH,
                            RD_D0R, RD_D0I, RD_D1R, RD_D1I,
                            DONE_PROCESS );
  attribute syn_encoding of ControllerStateType : type is "onehot";
  signal ctrl_ps, ctrl_ns : ControllerStateType;
  attribute syn_state_machine of ctrl_ps : signal is true;

  -- ------------------------------------------------------------------------
  -- - Registered Signals
  -- ------------------------------------------------------------------------
  signal enable_w_addrgen_int0     : std_logic;
  signal enable_w_addrgen_int1     : std_logic;
  signal enable_w_addrgen_int1_del : std_logic;

  signal bank0r_we0                : std_logic;
  signal bank0i_we0                : std_logic;
  signal bank1r_we0                : std_logic;
  signal bank1i_we0                : std_logic;

  signal bank0r_we1                : std_logic;
  signal bank0i_we1                : std_logic;
  signal bank1r_we1                : std_logic;
  signal bank1i_we1                : std_logic;

  signal bank0r_we1_del            : std_logic;
  signal bank0i_we1_del            : std_logic;
  signal bank1r_we1_del            : std_logic;
  signal bank1i_we1_del            : std_logic;

  signal skew_enable_int           : std_logic;

  signal data_count                : std_logic_vector(10 downto 0);
  signal node_count                : std_logic_vector(12 downto 0);
  signal flush_count               : std_logic_vector( 3 downto 0);
  signal done_int                  : std_logic;

begin

  -- ************************************************************************
  -- * Component Instantiations
  -- ************************************************************************
  i_pipe_enable_w_addrgen: reg_pipe_single
    generic map ( DEPTH=>12 )
    port map (
      reset_n => reset_n,
      clock   => clock,
      enable  => enable,
      i       => enable_w_addrgen_int1,
      o       => enable_w_addrgen_int1_del
    );

  i_pipe_bank0r_we: reg_pipe_single
    generic map ( DEPTH=>12 )
    port map (
      reset_n => reset_n,
      clock   => clock,
      enable  => enable,
      i       => bank0r_we1,
      o       => bank0r_we1_del
    );

  i_pipe_bank0i_we: reg_pipe_single
    generic map ( DEPTH=>12 )
    port map (
      reset_n => reset_n,
      clock   => clock,
      enable  => enable,
      i       => bank0i_we1,
      o       => bank0i_we1_del
    );

  i_pipe_bank1r_we: reg_pipe_single
    generic map ( DEPTH=>12 )
    port map (
```

```
    reset_n => reset_n,
    clock   => clock,
    enable  => enable,
    i       => bank1r_we1,
    o       => bank1r_we1_del
  );

i_pipe_bank1i_we: reg_pipe_single
  generic map ( DEPTH=>12 )
  port map (
    reset_n => reset_n,
    clock   => clock,
    enable  => enable,
    i       => bank1i_we1,
    o       => bank1i_we1_del
  );

i_pipe_skew_enable: reg_pipe_single
  generic map ( DEPTH=>10 )
  port map (
    reset_n => reset_n,
    clock   => clock,
    enable  => enable,
    i       => skew_enable_int,
    o       => skew_enable
  );

-- *******************************************************************
-- * Combinational Assignments
-- *******************************************************************
busy             <= '0' when( ctrl_ps=IDLE ) else '1';
k                <= shl( node_count(8 downto 0), node_count(12 downto 9) );

enable_w_addrgen <= enable_w_addrgen_int0 or enable_w_addrgen_int1_del;
bank0r_we        <= bank0r_we0 or bank0r_we1_del;
bank0i_we        <= bank0i_we0 or bank0i_we1_del;
bank1r_we        <= bank1r_we0 or bank1r_we1_del;
bank1i_we        <= bank1i_we0 or bank1i_we1_del;

-- *******************************************************************
-- *
-- *******************************************************************
sync: process( reset_n, clock )
begin
  if ( reset_n='0' ) then
    done_int              <= '0';
    done                  <= '0';
    data_count            <= (others=>'0');
    node_count            <= (others=>'0');
    flush_count           <= (others=>'0');

    write_sel             <= '0';

    bank0r_we0            <= '0';
    bank0i_we0            <= '0';
    bank1r_we0            <= '0';
    bank1i_we0            <= '0';

    bank0r_we1            <= '0';
    bank0i_we1            <= '0';
    bank1r_we1            <= '0';
    bank1i_we1            <= '0';

    read_sel              <= '0';
    bank_sel              <= "00";

    engine_enable         <= '0';

    skew_enable_int       <= '0';

    enable_w_addrgen_int0 <= '0';
    enable_w_addrgen_int1 <= '0';
    enable_r_addrgen0     <= '0';
    enable_r_addrgen1     <= '0';
    select_r_addrgen      <= '0';
```

```
  ctrl_ps              <= IDLE;
elsif ( rising_edge(clock) ) then

  done <= done_int;

  if ( enable='1' ) then
    ctrl_ps                 <= ctrl_ns;

    done_int                <= '0';

    bank0r_we0              <= '0';
    bank0i_we0              <= '0';
    bank1r_we0              <= '0';
    bank1i_we0              <= '0';

    bank0r_we1              <= '0';
    bank0i_we1              <= '0';
    bank1r_we1              <= '0';
    bank1i_we1              <= '0';

    read_sel                <= '0';
    bank_sel                <= "00";

    enable_w_addrgen_int0 <= '0';
    enable_w_addrgen_int1 <= '0';
    enable_r_addrgen1     <= '0';

    case ctrl_ps is
      when IDLE   =>
        enable_r_addrgen0 <= '0';
        select_r_addrgen  <= '0';
        data_count        <= (others=>'0');
        node_count        <= (others=>'0');
        flush_count       <= (others=>'0');

      when WR_D0R =>
        write_sel             <= '0';
        bank0r_we0            <= '1';
        data_count            <= data_count + '1';

      when WR_D0I =>
        write_sel             <= '0';
        bank0i_we0            <= '1';
        data_count            <= data_count + '1';
        enable_w_addrgen_int0 <= '1';

      when WR_D1R =>
        write_sel             <= '0';
        bank1r_we0            <= '1';
        data_count            <= data_count + '1';

      when WR_D1I =>
        write_sel             <= '0';
        bank1i_we0            <= '1';
        data_count            <= data_count + '1';
        enable_w_addrgen_int0 <= '1';

      when START_PROCESS =>
        write_sel         <= '1';
        enable_r_addrgen0 <= '1';
        enable_r_addrgen1 <= '0';
        select_r_addrgen  <= '0';
        node_count        <= (others=>'0');
        if ( ctrl_ns=PROCESS_NODE ) then
          engine_enable     <= '1';
          skew_enable_int   <= '1';
        end if;

      when PROCESS_NODE =>
        enable_w_addrgen_int1 <= '1';
        bank0r_we1            <= '1';
        bank0i_we1            <= '1';
        bank1r_we1            <= '1';
        bank1i_we1            <= '1';
```

```
            node_count            <= node_count + '1';

          when FLUSH =>
            flush_count <= flush_count + '1';
            if ( ctrl_ns=RD_D0R ) then
              done_int          <= '1';
              engine_enable     <= '0';
              skew_enable_int   <= '0';
              enable_r_addrgen0 <= '0';
              select_r_addrgen  <= '1';
            end if;

          when RD_D0R =>
            read_sel          <= '1';
            bank_sel          <= "00";
            data_count        <= data_count + '1';
            enable_r_addrgen1 <= '1';

          when RD_D0I =>
            read_sel          <= '1';
            bank_sel          <= "01";
            data_count        <= data_count + '1';

          when RD_D1R =>
            read_sel          <= '1';
            bank_sel          <= "10";
            data_count        <= data_count + '1';
            enable_r_addrgen1 <= '1';

          when RD_D1I =>
            read_sel          <= '1';
            bank_sel          <= "11";
            data_count        <= data_count + '1';

          when DONE_PROCESS =>

          when others =>
--              data_count <= (others=>'0');
        end case;

      end if;
    end if;
  end process sync;

  -- ***********************************************************************
  -- *
  -- ***********************************************************************
  combin: process( ctrl_ps, start, data_count, node_count, flush_count )
  begin
    case ctrl_ps is
      when IDLE   =>
        if ( start='0' ) then
          ctrl_ns <= IDLE;
        else
          ctrl_ns <= WR_D0R;
        end if;

      when WR_D0R => ctrl_ns <= WR_D0I;
      when WR_D0I => ctrl_ns <= WR_D1R;
        if ( data_count=N_DATA_DIV2-1 ) then
          ctrl_ns <= WR_D1R;
        else
          ctrl_ns <= WR_D0R;
        end if;
      when WR_D1R => ctrl_ns <= WR_D1I;
      when WR_D1I =>
        if ( data_count=N_DATA-1 ) then
          ctrl_ns <= START_PROCESS;
        else
          ctrl_ns <= WR_D1R;
        end if;

      when START_PROCESS =>
        ctrl_ns <= PROCESS_NODE;
```

```
      when PROCESS_NODE =>
        if ( node_count=NODES-1 ) then
          ctrl_ns <= FLUSH;
        else
          ctrl_ns <= PROCESS_NODE;
        end if;

      when FLUSH  =>
        if ( flush_count/="1100" ) then
          ctrl_ns <= FLUSH;
        else
          ctrl_ns <= RD_D0R;
        end if;

      when RD_D0R => ctrl_ns <= RD_D0I;
      when RD_D0I =>
        if ( data_count=N_DATA_DIV2-1 ) then
          ctrl_ns <= RD_D1R;
        else
          ctrl_ns <= RD_D0R;
        end if;

      when RD_D1R => ctrl_ns <= RD_D1I;
      when RD_D1I =>
        if ( data_count=N_DATA-1 ) then
          ctrl_ns <= DONE_PROCESS;
        else
          ctrl_ns <= RD_D1R;
        end if;

      when DONE_PROCESS =>
        ctrl_ns <= IDLE;

      when others => ctrl_ns <= IDLE;
    end case;
  end process combin;

end architecture rtl;
```

D.7. mem_bank.vhd

```
-------------------------------------------------------------------------------
--
-------------------------------------------------------------------------------
library ieee;
library synplify;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use synplify.attributes.all;

entity mem_bank is
  port (
    clock  : in  std_logic;

    we     : in  std_logic;
    w_addr : in  std_logic_vector( 8 downto 0);
    w_din  : in  std_logic_vector(15 downto 0);

    r_addr : in  std_logic_vector( 8 downto 0);
    r_dout : out std_logic_vector(15 downto 0)
  );
end entity mem_bank;

architecture rtl of mem_bank is

  -- ---------------------------------------------------------------------------
  -- - Constants & New Types
  -- ---------------------------------------------------------------------------
  type MemType is array( 0 to 511 ) of std_logic_vector(15 downto 0);

  -- ---------------------------------------------------------------------------
  -- - Registered Signals
  -- ---------------------------------------------------------------------------
  signal mem : MemType;
```

```
  attribute syn_ramstyle of mem : signal is "block_ram";

  signal r_addr_reg : std_logic_vector(8 downto 0);

begin
  -- ************************************************************************
  -- * Combinational Assignments
  -- ************************************************************************
  r_dout <= mem( conv_integer(r_addr_reg) );

  -- ************************************************************************
  -- *
  -- ************************************************************************
  Write: process( clock )
  begin
    if ( rising_edge(clock) ) then
      if ( we='1' ) then
        mem( conv_integer(w_addr) ) <= w_din;
      end if;

      r_addr_reg <= r_addr;
    end if;
  end process Write;

end architecture rtl;
```

D.8. mult.vhd

```
-------------------------------------------------------------------------------
--
-------------------------------------------------------------------------------
library ieee;
library synplify;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
--use ieee.std_logic_signed.all;
use synplify.attributes.all;

entity mult is
  generic (
    A_WIDTH : positive := 8;
    B_WIDTH : positive := 8
  );
  port (
    reset_n : in  std_logic;
    clock   : in  std_logic;
    enable  : in  std_logic;

    a       : in  std_logic_vector(A_WIDTH-1           downto 0);
    b       : in  std_logic_vector(B_WIDTH-1           downto 0);
    p       : out std_logic_vector((A_WIDTH+B_WIDTH-1) downto 0)
  );
end entity mult;

architecture rtl of mult is

  -- ---------------------------------------------------------------------------
  -- - Componenet Declarations
  -- ---------------------------------------------------------------------------
  component reg_pipe_single
    generic (
      DEPTH : positive
    );
    port (
      reset_n : in  std_logic;
      clock   : in  std_logic;
      enable  : in  std_logic;
      i       : in  std_logic;
      o       : out std_logic
    );
  end component;

  -- ---------------------------------------------------------------------------
```

```
  -- - Constants & New Types
  -- ---------------------------------------------------------------------------
  type PTYPE is array(B_WIDTH-1 downto 1) of std_logic_vector(A_WIDTH   downto 0);
  type ATYPE is array(B_WIDTH-1 downto 1) of std_logic_vector(A_WIDTH-1 downto 0);

  -- ---------------------------------------------------------------------------
  -- - Registered Signals
  -- ---------------------------------------------------------------------------
  signal pp0   : std_logic_vector(A_WIDTH-1 downto 0);
  signal pp    : PTYPE;

  signal a_reg : ATYPE;
  signal b_reg : std_logic_vector(B_WIDTH-1 downto 1);

  signal pp1   : std_logic_vector(A_WIDTH downto 0);
  signal pp2   : std_logic_vector(A_WIDTH downto 0);
  signal pp3   : std_logic_vector(A_WIDTH downto 0);
  signal pp4   : std_logic_vector(A_WIDTH downto 0);
  signal pp5   : std_logic_vector(A_WIDTH downto 0);
  signal pp6   : std_logic_vector(A_WIDTH downto 0);
  signal pp7   : std_logic_vector(A_WIDTH downto 0);

begin

  -- ***************************************************************************
  -- * Component Instantiations
  -- ***************************************************************************
  i_bn: for i in B_WIDTH-1 downto 1 generate
    i_b: reg_pipe_single
      generic map ( DEPTH=>i )
      port map ( reset_n=>reset_n, clock=>clock, enable=>enable, i=>b(i), o=>b_reg(i) );
  end generate;

  -- Calculate the final result
  i_p0: reg_pipe_single
    generic map ( DEPTH=>B_WIDTH-1 )
    port map ( reset_n=>reset_n, clock=>clock, enable=>enable, i=>pp0(0), o=>p(0) );

  i_pn: for i in B_WIDTH-2 downto 1 generate
    i_pn: reg_pipe_single
      generic map ( DEPTH=>B_WIDTH-1-i )
      port map ( reset_n=>reset_n, clock=>clock, enable=>enable, i=>pp(i)(0), o=>p(i) );
  end generate;

  -- ***************************************************************************
  -- * Combinational Assignments
  -- ***************************************************************************
  p((A_WIDTH+B_WIDTH-1) downto B_WIDTH-1) <= pp(B_WIDTH-1)(A_WIDTH downto 0);

  -- ***************************************************************************
  -- *
  -- ***************************************************************************
  process( reset_n, clock )
  begin
    if ( reset_n='0' ) then
      a_reg <= (others=>(others=>'0'));
      pp0   <= (others=>'0');
      pp    <= (others=>(others=>'0'));
    elsif ( rising_edge(clock) ) then
      if ( enable='1' ) then
        for i in B_WIDTH-1 downto 2 loop
          a_reg(i) <= a_reg(i-1);
        end loop;
        a_reg(1) <= a;

        -- Calculate the first multiplication
        for i in A_WIDTH-1 downto 0 loop
          pp0(i) <= a(i) and b(0);
        end loop;

        -- Calculate the intermediate results
        for i in 1 to B_WIDTH-1 loop
          if i=1 then
            if ( b_reg(i)='1' ) then
              pp(i) <= (pp0(A_WIDTH-1) & pp0(A_WIDTH-1) & pp0(A_WIDTH-1 downto 1)) +
```

```
                   (a_reg(i)(A_WIDTH-1) & a_reg(i));
          else
            pp(i) <= pp0(A_WIDTH-1) & pp0(A_WIDTH-1) & pp0(A_WIDTH-1 downto 1);
          end if;
        elsif ( i=(B_WIDTH-1) ) then
          if ( b_reg(i)='1' ) then
            pp(i) <= (pp(i-1)(A_WIDTH) & pp(i-1)(A_WIDTH downto 1)) -
                     (a_reg(i)(A_WIDTH-1) & a_reg(i));
          else
            pp(i) <= (pp(i-1)(A_WIDTH) & pp(i-1)(A_WIDTH downto 1));
          end if;
        else
          if ( b_reg(i)='1' ) then
            pp(i) <= (pp(i-1)(A_WIDTH) & pp(i-1)(A_WIDTH downto 1)) +
                     (a_reg(i)(A_WIDTH-1) & a_reg(i));
          else
            pp(i) <= (pp(i-1)(A_WIDTH) & pp(i-1)(A_WIDTH downto 1));
          end if;
        end if;
      end loop;
    end if;

  end if;
end process;

end architecture rtl;
```

D.9. reg_pipe.vhd

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

entity reg_pipe is
  generic (
    DEPTH : positive;
    WIDTH : positive
  );
  port (
    reset_n : in  std_logic;
    clock   : in  std_logic;
    enable  : in  std_logic;
    i       : in  std_logic_vector(WIDTH-1 downto 0);
    o       : out std_logic_vector(WIDTH-1 downto 0)
  );
--begin
--  assert DEPTH>1 report "Test" severity ERROR;
--  assert WIDTH>1 report "Test" severity ERROR;
end entity reg_pipe;

architecture rtl of reg_pipe is

  -- ---------------------------------------------------------------------------
  -- - Constants & New Types
  -- ---------------------------------------------------------------------------
  type RegType is array(DEPTH-1 downto 0) of std_logic_vector(WIDTH-1 downto 0);

  -- ---------------------------------------------------------------------------
  -- - Registered Signals
  -- ---------------------------------------------------------------------------
  signal reg : RegType;

begin

  -- ***************************************************************************
  -- * Combinational Assignments
  -- ***************************************************************************
  o <= reg(DEPTH-1);

  -- ***************************************************************************
  -- *
```

```
-- ***********************************************************************
  process( reset_n, clock )
  begin
    if ( reset_n='0' ) then
      reg <= (others=>(others=>'0'));
    elsif ( rising_edge(clock) ) then
      if ( enable='1' ) then
        if ( DEPTH>1 ) then
          for i in DEPTH-1 downto 1 loop
            reg(i) <= reg(i-1);
          end loop;
        end if;
        reg(0) <= i;
      end if;
    end if;
  end process;

end architecture rtl;
```

D.10. reg_pipe_single.vhd

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

entity reg_pipe_single is
  generic (
    DEPTH : positive
  );
  port (
    reset_n : in  std_logic;
    clock   : in  std_logic;
    enable  : in  std_logic;
    i       : in  std_logic;
    o       : out std_logic
  );
--begin
--  assert ( DEPTH>1 ) report "Test" severity ERROR;
end entity reg_pipe_single;

architecture rtl of reg_pipe_single is

  -- ---------------------------------------------------------------------------
  -- - Registered Signals
  -- ---------------------------------------------------------------------------
  signal reg : std_logic_vector(DEPTH-1 downto 0);

begin

  -- ***********************************************************************
  -- * Combinational Assignments
  -- ***********************************************************************
  o <= reg(DEPTH-1);

  -- ***********************************************************************
  -- *
  -- ***********************************************************************
  process( reset_n, clock )
  begin
    if ( reset_n='0' ) then
      reg <= (others=>'0');
    elsif ( rising_edge(clock) ) then
      if ( enable='1' ) then
        if ( DEPTH>1 ) then
          reg <= reg(DEPTH-2 downto 0) & i;
        else
          reg(0) <= i;
        end if;
      end if;
    end if;
  end process;
```

```
end architecture rtl;
```

D.11. skew_buffer.vhd

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

entity skew_buffer is
  port (
    reset_n : in  std_logic;
    clock   : in  std_logic;

    enable  : in  std_logic;

    din0    : in  std_logic_vector(15 downto 0);
    din1    : in  std_logic_vector(15 downto 0);

    dout0   : out std_logic_vector(15 downto 0);
    dout1   : out std_logic_vector(15 downto 0)
  );
end entity skew_buffer;

architecture rtl of skew_buffer is

  -- ---------------------------------------------------------------------------
  -- - Constants & New Types
  -- ---------------------------------------------------------------------------
  type RegType is array(0 to 1) of std_logic_vector(15 downto 0);

  -- ---------------------------------------------------------------------------
  -- - Registered Signals
  -- ---------------------------------------------------------------------------
  signal reg0          : RegType;
  signal reg1          : RegType;
  signal reg2          : RegType;
  signal reg3          : RegType;

  signal in_count      : std_logic_vector(1 downto 0);
  signal in_count_del  : std_logic_vector(1 downto 0);
  signal in_count_del2 : std_logic_vector(1 downto 0);

begin

  -- ***************************************************************************
  -- *
  -- ***************************************************************************
  process( reset_n, clock )
  begin
    if ( reset_n='0' ) then
      reg0          <= (others=>(others=>'0'));
      reg1          <= (others=>(others=>'0'));
      reg2          <= (others=>(others=>'0'));
      reg3          <= (others=>(others=>'0'));
      in_count      <= (others=>'0');
      in_count_del  <= (others=>'0');
      in_count_del2 <= (others=>'0');
      dout0         <= (others=>'0');
      dout1         <= (others=>'0');
    elsif ( rising_edge(clock) ) then
      if ( enable='1' ) then
        in_count      <= in_count + '1';
        in_count_del  <= in_count;
        in_count_del2 <= in_count_del;

        case in_count is
          when "00"   => reg0(1) <= din1; reg0(0) <= din0;
          when "01"   => reg1(1) <= din1; reg1(0) <= din0;
          when "10"   => reg2(1) <= din1; reg2(0) <= din0;
          when others => reg3(1) <= din1; reg3(0) <= din0;
```

```
        end case;

        case in_count_del2 is
          when "00"   => dout1 <= reg1(0); dout0 <= reg0(0);
          when "01"   => dout1 <= reg1(1); dout0 <= reg0(1);
          when "10"   => dout1 <= reg3(0); dout0 <= reg2(0);
          when others => dout1 <= reg3(1); dout0 <= reg2(1);
        end case;
      end if;
    end if;
  end process;


end architecture rtl;
```

D.12. twiddle_factor.vhd

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

entity twiddle_factors is
  port (
    k   : in  std_logic_vector(8 downto 0);
    w_r : out std_logic_vector(7 downto 0);
    w_i : out std_logic_vector(7 downto 0)
  );
end entity twiddle_factors;

architecture rtl of twiddle_factors is

  -- ---------------------------------------------------------------------------
  -- - Constants & New Types
  -- ---------------------------------------------------------------------------
  type LookupTable is array(0 to 511) of std_logic_vector(7 downto 0);

  constant WR : LookupTable := (
    X"7F", X"7E", X"7E", X"7E", X"7E", X"7E", X"7E", X"7E",
    X"7E", X"7E", X"7E", X"7E", X"7E", X"7E", X"7E", X"7E",
    X"7E", X"7E", X"7E", X"7E", X"7E", X"7D", X"7D", X"7D",
    X"7D", X"7D", X"7D", X"7D", X"7D", X"7C", X"7C", X"7C",
    X"7C", X"7C", X"7C", X"7C", X"7B", X"7B", X"7B", X"7B",
    X"7B", X"7B", X"7A", X"7A", X"7A", X"7A", X"79", X"79",
    X"79", X"79", X"79", X"78", X"78", X"78", X"78", X"77",
    X"77", X"77", X"77", X"76", X"76", X"76", X"75", X"75",
    X"75", X"75", X"74", X"74", X"74", X"73", X"73", X"73",
    X"72", X"72", X"72", X"71", X"71", X"71", X"70", X"70",
    X"70", X"6F", X"6F", X"6E", X"6E", X"6E", X"6D", X"6D",
    X"6C", X"6C", X"6C", X"6B", X"6B", X"6A", X"6A", X"6A",
    X"69", X"69", X"68", X"68", X"67", X"67", X"66", X"66",
    X"66", X"65", X"65", X"64", X"64", X"63", X"63", X"62",
    X"62", X"61", X"61", X"60", X"60", X"5F", X"5F", X"5E",
    X"5E", X"5D", X"5D", X"5C", X"5B", X"5B", X"5A", X"5A",
    X"59", X"59", X"58", X"58", X"57", X"57", X"56", X"55",
    X"55", X"54", X"54", X"53", X"52", X"52", X"51", X"51",
    X"50", X"4F", X"4F", X"4E", X"4E", X"4D", X"4C", X"4C",
    X"4B", X"4B", X"4A", X"49", X"49", X"48", X"47", X"47",
    X"46", X"45", X"45", X"44", X"43", X"43", X"42", X"41",
    X"41", X"40", X"3F", X"3F", X"3E", X"3D", X"3D", X"3C",
    X"3B", X"3B", X"3A", X"39", X"39", X"38", X"37", X"37",
    X"36", X"35", X"34", X"34", X"33", X"32", X"32", X"31",
    X"30", X"2F", X"2F", X"2E", X"2D", X"2C", X"2C", X"2B",
    X"2A", X"2A", X"29", X"28", X"27", X"27", X"26", X"25",
    X"24", X"24", X"23", X"22", X"21", X"21", X"20", X"1F",
    X"1E", X"1E", X"1D", X"1C", X"1B", X"1B", X"1A", X"19",
    X"18", X"18", X"17", X"16", X"15", X"14", X"14", X"13",
    X"12", X"11", X"11", X"10", X"0F", X"0E", X"0D", X"0D",
    X"0C", X"0B", X"0A", X"0A", X"09", X"08", X"07", X"07",
    X"06", X"05", X"04", X"03", X"03", X"02", X"01", X"00",
    X"00", X"00", X"FF", X"FE", X"FD", X"FD", X"FC", X"FB",
    X"FA", X"F9", X"F9", X"F8", X"F7", X"F6", X"F6", X"F5",
```

```
  X"F4", X"F3", X"F3", X"F2", X"F1", X"F0", X"EF", X"EF",
  X"EE", X"ED", X"EC", X"EC", X"EB", X"EA", X"E9", X"E8",
  X"E8", X"E7", X"E6", X"E5", X"E5", X"E4", X"E3", X"E2",
  X"E2", X"E1", X"E0", X"DF", X"DF", X"DE", X"DD", X"DC",
  X"DC", X"DB", X"DA", X"D9", X"D9", X"D8", X"D7", X"D6",
  X"D6", X"D5", X"D4", X"D4", X"D3", X"D2", X"D1", X"D1",
  X"D0", X"CF", X"CE", X"CE", X"CD", X"CC", X"CC", X"CB",
  X"CA", X"C9", X"C9", X"C8", X"C7", X"C7", X"C6", X"C5",
  X"C5", X"C4", X"C3", X"C3", X"C2", X"C1", X"C1", X"C0",
  X"BF", X"BF", X"BE", X"BD", X"BD", X"BC", X"BB", X"BB",
  X"BA", X"B9", X"B9", X"B8", X"B7", X"B7", X"B6", X"B5",
  X"B5", X"B4", X"B4", X"B3", X"B2", X"B2", X"B1", X"B1",
  X"B0", X"AF", X"AF", X"AE", X"AE", X"AD", X"AC", X"AC",
  X"AB", X"AB", X"AA", X"A9", X"A9", X"A8", X"A8", X"A7",
  X"A7", X"A6", X"A6", X"A5", X"A5", X"A4", X"A3", X"A3",
  X"A2", X"A2", X"A1", X"A1", X"A0", X"A0", X"9F", X"9F",
  X"9E", X"9E", X"9D", X"9D", X"9C", X"9C", X"9B", X"9B",
  X"9A", X"9A", X"9A", X"99", X"99", X"98", X"98", X"97",
  X"97", X"96", X"96", X"96", X"95", X"95", X"94", X"94",
  X"94", X"93", X"93", X"92", X"92", X"92", X"91", X"91",
  X"90", X"90", X"90", X"8F", X"8F", X"8F", X"8E", X"8E",
  X"8E", X"8D", X"8D", X"8D", X"8C", X"8C", X"8C", X"8B",
  X"8B", X"8B", X"8B", X"8A", X"8A", X"8A", X"89", X"89",
  X"89", X"89", X"88", X"88", X"88", X"88", X"87", X"87",
  X"87", X"87", X"87", X"86", X"86", X"86", X"86", X"85",
  X"85", X"85", X"85", X"85", X"85", X"84", X"84", X"84",
  X"84", X"84", X"84", X"84", X"83", X"83", X"83", X"83",
  X"83", X"83", X"83", X"83", X"82", X"82", X"82", X"82",
  X"82", X"82", X"82", X"82", X"82", X"82", X"82", X"82",
  X"82", X"82", X"82", X"82", X"82", X"82", X"82", X"82"
);

constant WI : LookupTable := (
  X"00", X"00", X"FF", X"FE", X"FD", X"FD", X"FC", X"FB",
  X"FA", X"F9", X"F9", X"F8", X"F7", X"F6", X"F6", X"F5",
  X"F4", X"F3", X"F3", X"F2", X"F1", X"F0", X"EF", X"EF",
  X"EE", X"ED", X"EC", X"EC", X"EB", X"EA", X"E9", X"E8",
  X"E8", X"E7", X"E6", X"E5", X"E5", X"E4", X"E3", X"E2",
  X"E2", X"E1", X"E0", X"DF", X"DF", X"DE", X"DD", X"DC",
  X"DC", X"DB", X"DA", X"D9", X"D9", X"D8", X"D7", X"D6",
  X"D6", X"D5", X"D4", X"D4", X"D3", X"D2", X"D1", X"D1",
  X"D0", X"CF", X"CE", X"CE", X"CD", X"CC", X"CC", X"CB",
  X"CA", X"C9", X"C9", X"C8", X"C7", X"C7", X"C6", X"C5",
  X"C5", X"C4", X"C3", X"C3", X"C2", X"C1", X"C1", X"C0",
  X"BF", X"BF", X"BE", X"BD", X"BD", X"BC", X"BB", X"BB",
  X"BA", X"B9", X"B9", X"B8", X"B7", X"B7", X"B6", X"B5",
  X"B5", X"B4", X"B4", X"B3", X"B2", X"B2", X"B1", X"B1",
  X"B0", X"AF", X"AF", X"AE", X"AE", X"AD", X"AC", X"AC",
  X"AB", X"AB", X"AA", X"A9", X"A9", X"A8", X"A8", X"A7",
  X"A7", X"A6", X"A6", X"A5", X"A5", X"A4", X"A3", X"A3",
  X"A2", X"A2", X"A1", X"A1", X"A0", X"A0", X"9F", X"9F",
  X"9E", X"9E", X"9D", X"9D", X"9C", X"9C", X"9B", X"9B",
  X"9A", X"9A", X"9A", X"99", X"99", X"98", X"98", X"97",
  X"97", X"96", X"96", X"96", X"95", X"95", X"94", X"94",
  X"94", X"93", X"93", X"92", X"92", X"92", X"91", X"91",
  X"90", X"90", X"90", X"8F", X"8F", X"8F", X"8E", X"8E",
  X"8E", X"8D", X"8D", X"8D", X"8C", X"8C", X"8C", X"8B",
  X"8B", X"8B", X"8B", X"8A", X"8A", X"8A", X"89", X"89",
  X"89", X"89", X"88", X"88", X"88", X"88", X"87", X"87",
  X"87", X"87", X"87", X"86", X"86", X"86", X"86", X"85",
  X"85", X"85", X"85", X"85", X"85", X"84", X"84", X"84",
  X"84", X"84", X"84", X"84", X"83", X"83", X"83", X"83",
  X"83", X"83", X"83", X"83", X"82", X"82", X"82", X"82",
  X"82", X"82", X"82", X"82", X"82", X"82", X"82", X"82",
  X"82", X"82", X"82", X"82", X"82", X"82", X"82", X"82",
  X"81", X"82", X"82", X"82", X"82", X"82", X"82", X"82",
  X"82", X"82", X"82", X"82", X"82", X"82", X"82", X"82",
  X"82", X"82", X"82", X"82", X"82", X"83", X"83", X"83",
  X"83", X"83", X"83", X"83", X"83", X"84", X"84", X"84",
  X"84", X"84", X"84", X"84", X"85", X"85", X"85", X"85",
  X"85", X"85", X"86", X"86", X"86", X"86", X"87", X"87",
  X"87", X"87", X"87", X"88", X"88", X"88", X"88", X"89",
  X"89", X"89", X"89", X"8A", X"8A", X"8A", X"8B", X"8B",
  X"8B", X"8B", X"8C", X"8C", X"8C", X"8D", X"8D", X"8D",
  X"8E", X"8E", X"8E", X"8F", X"8F", X"8F", X"90", X"90",
```

```
    X"90", X"91", X"91", X"92", X"92", X"92", X"93", X"93",
    X"94", X"94", X"94", X"95", X"95", X"96", X"96", X"96",
    X"97", X"97", X"98", X"98", X"99", X"99", X"9A", X"9A",
    X"9A", X"9B", X"9B", X"9C", X"9C", X"9D", X"9D", X"9E",
    X"9E", X"9F", X"9F", X"A0", X"A0", X"A1", X"A1", X"A2",
    X"A2", X"A3", X"A3", X"A4", X"A5", X"A5", X"A6", X"A6",
    X"A7", X"A7", X"A8", X"A8", X"A9", X"A9", X"AA", X"AB",
    X"AB", X"AC", X"AC", X"AD", X"AE", X"AE", X"AF", X"AF",
    X"B0", X"B1", X"B1", X"B2", X"B2", X"B3", X"B4", X"B4",
    X"B5", X"B5", X"B6", X"B7", X"B7", X"B8", X"B9", X"B9",
    X"BA", X"BB", X"BB", X"BC", X"BD", X"BD", X"BE", X"BF",
    X"BF", X"C0", X"C1", X"C1", X"C2", X"C3", X"C3", X"C4",
    X"C5", X"C5", X"C6", X"C7", X"C7", X"C8", X"C9", X"C9",
    X"CA", X"CB", X"CC", X"CC", X"CD", X"CE", X"CE", X"CF",
    X"D0", X"D1", X"D1", X"D2", X"D3", X"D4", X"D4", X"D5",
    X"D6", X"D6", X"D7", X"D8", X"D9", X"D9", X"DA", X"DB",
    X"DC", X"DC", X"DD", X"DE", X"DF", X"DF", X"E0", X"E1",
    X"E2", X"E2", X"E3", X"E4", X"E5", X"E5", X"E6", X"E7",
    X"E8", X"E8", X"E9", X"EA", X"EB", X"EC", X"EC", X"ED",
    X"EE", X"EF", X"EF", X"F0", X"F1", X"F2", X"F3", X"F3",
    X"F4", X"F5", X"F6", X"F6", X"F7", X"F8", X"F9", X"F9",
    X"FA", X"FB", X"FC", X"FD", X"FD", X"FE", X"FF", X"00"
  );

begin

  -- ***************************************************************
  -- * Combinational Assignments
  -- ***************************************************************
  w_r <= WR( conv_integer(k) );
  w_i <= WR( conv_integer(k) );

end architecture rtl;
```

D.13. Testbench "cfft1024_tb.vhd"

```
--------------------------------------------------------------------------------
--
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;
use ieee.std_logic_textio.all;

entity cfft1024_tb is
end entity cfft1024_tb;

architecture behavioral of cfft1024_tb is

  -- ----------------------------------------------------------------------------
  -- - Componenet Declarations
  -- ----------------------------------------------------------------------------
  component cfft1024
    port (
      reset_n     : in  std_logic;
      clock       : in  std_logic;
      enable      : in  std_logic;
      start       : in  std_logic;
      busy        : out std_logic;
      done        : out std_logic;
      data_in     : in  std_logic_vector( 7 downto 0);
      data_out    : out std_logic_vector(15 downto 0)
    );
  end component cfft1024;

  component twiddle_factors
    port (
      k   : in  std_logic_vector(8 downto 0);
      w_r : out std_logic_vector(7 downto 0);
      w_i : out std_logic_vector(7 downto 0)
    );
  end component twiddle_factors;
```

```
  -- ----------------------------------------------------------------------
  -- - Constants & New Types
  -- ----------------------------------------------------------------------
  constant N_POINTS        : integer := 1024;
  constant N_DATA          : integer := N_POINTS*2-1;
  constant NODES_PER_LEVEL : integer := N_POINTS/2;
--  constant LEVELS          : integer := log2(N_POINTS);
  constant LEVELS          : integer := 10;
  constant ITERATIONS      : integer := NODES_PER_LEVEL*LEVELS;

  type MemType is array( 0 to 511 ) of std_logic_vector(15 downto 0);
  signal mem_bank0r : MemType;
  signal mem_bank0i : MemType;
  signal mem_bank1r : MemType;
  signal mem_bank1i : MemType;

  -- ----------------------------------------------------------------------
  -- - Procedures and Functions
  -- ----------------------------------------------------------------------

  -- ----------------------------------------------------------------------
  -- - Signal Declarations
  -- ----------------------------------------------------------------------
  signal reset_n  : std_logic;
  signal clock    : std_logic;
  signal enable   : std_logic;
  signal start    : std_logic;
  signal busy     : std_logic;
  signal done     : std_logic;
  signal data_in  : std_logic_vector( 7 downto 0);
  signal data_out : std_logic_vector(15 downto 0);

begin

  -- **********************************************************************
  -- * Component Instantiations
  -- **********************************************************************
  uut: cfft1024
    port map (
      reset_n  => reset_n,
      clock    => clock,
      enable   => enable,
      start    => start,
      busy     => busy,
      done     => done,
      data_in  => data_in,
      data_out => data_out
    );

--  twiddles: twiddle_factors
--    port map (
--      k   => k,
--      w_r => w_r,
--      w_i => w_i
--    );

  -- **********************************************************************
  -- * Combinational Assignments
  -- **********************************************************************
  ClkGen: process
  begin
    reset_n <= '0', '1' after 5 ns;
    loop
      clock <= '0', '1' after 10 ns;
      wait for 20 ns;
    end loop;
  end process ClkGen;

  -- **********************************************************************
  -- *
  -- **********************************************************************
  ApplyStimulus: process
--    file input_vector  : text open read_mode  is "source.txt";
    file input_vector    : text;
    variable l           : line;
```

```
    variable d_real       : std_logic_vector( 7 downto 0);
    variable d_imag       : std_logic_vector( 7 downto 0);
    variable data_count   : std_logic_vector(10 downto 0);

  begin

    data_count  := (others=>'0');

    enable  <= '0';
    start   <= '0';
    data_in <= (others=>'0');
    wait until rising_edge(clock);

    enable <= '1';
    start  <= '1';

    wait until rising_edge(clock);
    start  <= '0';

    file_open( input_vector, "source.txt", read_mode );
    while not endfile( input_vector ) loop
      readline( input_vector, l );

      read( l, d_real );
      data_in <= d_real;
      wait until rising_edge(clock);

      readline( input_vector, l );
      read( l, d_imag );
      data_in <= d_imag;
      wait until rising_edge(clock);
    end loop;

    file_close( input_vector );
    wait;
  end process ApplyStimulus;

  -- ***********************************************************************
  -- *
  -- ***********************************************************************
  CaptureOutput: process

    file output_vector : text;
    variable l         : line;
    variable data_count  : std_logic_vector(10 downto 0);

  begin
    file_open( output_vector, "result.txt", write_mode );
    loop
      wait until rising_edge(clock);
      exit when done='1';
    end loop;

    data_count := (others=>'0');
    while ( data_count/=N_DATA ) loop
      wait until rising_edge(clock);
      data_count := data_count + '1';
      write( l, data_out );
      write( l, string'("    [") );
      hwrite( l, data_out );
      write( l, string'("]") );
      writeline( output_vector, l );
    end loop;
    write( l, data_out );
    write( l, string'("    [") );
    hwrite( l, data_out );
    write( l, string'("]") );
    writeline( output_vector, l );

    file_close( output_vector );
    wait;
  end process CaptureOutput;

end architecture behavioral;
```