NORTHWESTERN UNIVERSITY

A Methodology For Translating Scheduled Software Binaries onto Field Programmable Gate Arrays

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENTS OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical and Computer Engineering

By

David C. Zaretsky

EVANSTON, ILLINOIS

December 2005

Abstract

A METHODOLOGY FOR TRANSLATING SCHEDULED SOFTWARE BINARIES ONTO FIELD PROGRAMMABLE GATE ARRAYS

DAVID C. ZARETSKY

Recent advances in embedded communications and control systems are pushing the computational limits of DSP applications, driving the need for hardware/software co-design systems. This dissertation describes the development and architecture of the FREEDOM compiler that translates DSP software binaries to hardware descriptions for FPGAs as part of a hardware/software co-design. We present our methodology for translating scheduled software binaries to hardware, and described an array of optimizations that were implemented in the compiler. Our balanced scheduling and operation chaining techniques show even greater improvements in performance. Our resource sharing optimization generates templates of reoccurring patterns in a design to reduce resource utilization. Our structural extraction technique identifies structures in a design for partitioning as part of a hardware/software co-design. These concepts were tested in a case study of an MPEG-4 decoder. Results indicate speedups between 14-67x in terms of cycles and 6-22x in terms of time for the FPGA implementation over that of the DSP. Comparison of results with another high-level synthesis tool indicates that binary translation is an efficient method for high-level synthesis.

Acknowledgements

I wish to thank my advisor, Prith Banerjee, for allowing me the opportunity to work under his guidance as a graduate student at Northwestern University. His support and encouragement has led to the accomplishments of my work as described in this dissertation. I would also like to thank my secondary advisor, Robert Dick, who was a tremendous help in much of my research. His guidance and input was immeasurable.

I wish to thank Professors Prith Banerjee, Robert Dick, Seda Ogrenci Memik, and Hai Zhou for participating on the final examination committee for my dissertation.

To my colleague, Gaurav Mittal, who shared a great deal of the burden on this project, I wish to thank you for all your helpful insights in the different aspects of the project that were invaluable to the successful completion of this dissertation. I wish to also thank my fellow colleagues at Northwestern University who have shared research ideas and with whom I have collaborated.

A special thank you to Kees Vissers, Robert Turney, and Paul Schumacher at Xilinx Research Labs for providing me with the MPEG-4 source code to be used in my Ph.D. research and in this dissertation.

Finally, I wish to thank my parents for teaching me… the sky is the limit!

Table of Contents

Abstract ..... **iii**

Acknowledgements ..... **iv**

Table of Contents ..... **v**

List of Tables ..... **xi**

List of Figures ..... **xiii**

Chapter 1: Introduction **1**

1.1 Binary to Hardware Translation 3

1.2 Texas Instruments TMS320C6211 DSP Design Flow 6

1.3 Xilinx Virtex II FPGA Design Flow 8

1.4 Motivational Example 11

1.5 Dissertation Overview 12

Chapter 2: Related Work **14**

2.1 High-Level Synthesis 14

2.2 Binary Translation 17

2.3 Hardware-Software Co-Designs 18

Chapter 3: The FREEDOM Compiler **21**

3.1 The Machine Language Syntax Tree 22

3.1.1 MST Instructions 23

3.1.2 Control Flow Analysis 24
3.1.3 Data Dependency Analysis 24
3.2 The Control and Data Flow Graph 26
3.2.1 Memory Operations 27
3.2.2 Predicate Operations 28
3.2.3 Procedure, Function and Library Calls 30
3.3 The Hardware Description Language 31
3.3.1 Memory Support 31
3.3.2 Procedure Calls 32
3.4 The Graphical User Interface 33
3.5 Verification 34
3.5.1 Verifying Bit-True Accuracy 35
3.5.2 Testbenches 36
3.6 Summary 36
Chapter 4: Building a Control and Data Flow Graph from Scheduled Assembly ... 38
4.1 Related Work 41
4.2 Generating a Control Flow Graph 42
4.3 Linearizing Pipelined Operations 45
4.3.1 Event-Triggered Operations 46
4.3.2 Linearizing Computational Operations 49
4.3.3 Linearizing Branch Operations 51

4.3.4 The Linearization Algorithm 52
4.4 Generating the Control and Data Flow Graph 54
4.5 Experimental Results 55
4.6 Summary 57
Chapter 5: Control and Data Flow Graph Optimizations 58
5.1 CDFG Analysis 58
5.1.1 Structural Analysis 59
5.1.2 Reaching Definitions 60
5.2 CDFG Optimizations 61
5.2.1 Identifying Input and Output Ports 61
5.2.2 Single Static-Variable Assignment 63
5.2.3 Undefined Variable Elimination 64
5.2.4 Common Sub-Expression Elimination 64
5.2.5 Copy Propagation 66
5.2.6 Constant Folding 66
5.2.7 Constant Propagation 67
5.2.8 Strength Reduction 68
5.2.9 Constant Predicate Elimination 68
5.2.10 Boolean Reduction 69
5.2.11 Shift Reduction 70
5.2.12 Redundant Memory Access Elimination 70

5.2.13 Block-Set Merging....72
5.2.14 Dead-Code Elimination....73
5.2.15 Empty Block Extraction....73
5.2.16 Loop Unrolling....74
5.2.17 Register Allocation....75
5.3 Experimental Results....76
5.4 Summary....79
Chapter 6: Scheduling....80
6.1 Related Work....82
6.2 Balanced Scheduling....84
6.3 Balanced Chaining....89
6.3.1 Modeling Delays....91
6.3.2 Balanced Chaining Algorithm....95
6.4 Experimental Results....97
6.5 Summary....103
Chapter 7: Resource Sharing....105
7.1 Related Work....108
7.2 Dynamic Resource Sharing....110
7.2.1 Linear DAG Isomorphism....110
7.2.2 Growing Templates....112
7.2.3 Reconverging Paths....114

7.2.4 Template Matching 115
7.2.5 Selecting Templates 117
7.2.6 Building Template Structures 118
7.2.7 Resource Sharing Algorithm 119
7.3 Experimental Results 121
7.4 Summary 124
Chapter 8: Hardware-Software Partitioning of Software Binaries 125
8.1 Related Work 126
8.2 Structural Extraction 128
8.2.1 Discovering Extractable Structures 129
8.2.2 Selecting Structures for Extraction 132
8.2.3 Extracting Structures 133
8.3 Summary 135
Chapter 9: A Case Study: MPEG-4 136
9.1 Overview of the MPEG-4 Decoder 136
9.1.1 Parser 138
9.1.2 Texture Decoding 139
9.1.3 Motion Compensation 139
9.1.4 Reconstruction 140
9.1.5 Memory Controller 140
9.1.6 Display Controller 141

9.2 Experimental Results 141

9.3 Summary 143

Chapter 10: Conclusions and Future Work 144

10.1 Summary of Contributions 145

10.2 Comparison with High-Level Synthesis Performances 146

10.3 Future Work 147

References 149

Appendix A: MST Grammar 157

Appendix B: HDL Grammar 159

Appendix C: Verilog Simulation Testbench 163

List of Tables

Table 3.1. Supported operations in the MST grammar. 23

Table 4.1. Experimental results on pipelined benchmarks. 56

Table 5.1. Clock cycle results for CDFG optimizations. 78

Table 5.2. Frequency results in MHz for CDFG optimizations. 78

Table 5.3. Area results in LUTs for CDFG optimizations. 79

Table 6.1. Delay models for operations on the Xilinx Virtex II FPGA. 98

Table 6.2. Delay models for operations on the Altera Stratix FPGA 99

Table 6.3. Comparison of scheduling routines for Xilinx Virtex II FPGA. 100

Table 6.4. Comparison of scheduling routines for Altera Stratix FPGA. 101

Table 6.5. Comparison of chaining routines for Xilinx Virtex II FPGA. 102

Table 6.6. Comparison of chaining routines for Altera Stratix FPGA. 103

Table 7.1. Number of templates generated and maximum template sizes for varying look-ahead and backtracking depths. 123

Table 7.2. Number and percentage resources reduced with varying look-ahead and backtracking depth. 123

Table 7.3. Timing results in seconds for resource sharing with varying look-ahead and backtracking depth. 124

Table 9.1. MPEG-4 standard. 137

Table 9.2. Comparison of MPEG-4 decoder modules on DSP and FPGA platforms. .. 142

Table 10.1. Performance comparison between the TI C6211 DSP and the PACT and FREEDOM compiler implementations on the Xilinx Virtex II FPGA. 147

List of Figures

Figure 1.1. Using binary translation to implement a harware/software co-design. 4

Figure 1.2. FREEDOM compiler bridging the gap between DSP and FPGA designs environments 5

Figure 1.3. Texas Instruments C6211 DSP architecture 7

Figure 1.4. Texas Instruments C6211 DSP development flow. 8

Figure 1.5. Xilinx Virtex II FPGA development flow 10

Figure 1.6. Example TI C6000 DSP assembly code. 11

Figure 3.1. Overview of the FREEDOM compiler infrastructure. 22

Figure 3.2. MST instructions containing timesteps and delays for determining data dependencies. 25

Figure 3.3. TI assembly code for a *dot product* function. 26

Figure 3.4. CDFG representation for a *dot product* function. 27

Figure 3.5. Predicated operation in the CDFG. 29

Figure 3.6. CDFG representation of a CALL operation. 30

Figure 3.7. HDL representation for calling the *dot product* function. 32

Figure 3.8. HDL process for asynchronous memory MUX controller. 33

Figure 3.9. Graphical user interface for the FREEDOM compiler 34

Figure 4.1. TI C6000 assembly code for a pipelined *vectorsum* procedure. 39

Figure 4.2. Control flow graph for *vectorsum.* 43

Figure 4.3. Branch target inside a parallel instruction set..... 45

Figure 4.4. MST representation with instruction replication. 45

Figure 4.5. Event-triggering for a pipelined branch operation in a loop body. 48

Figure 4.6. Linearization of a pipelined load instruction in the *vectorsum* procedure. ... 50

Figure 4.7. Linearization of a pipelined branch instruction in *vectorsum*. 52

Figure 4.8. Linearization algorithm for pipelined operations. 54

Figure 4.9. Procedure for generating a CDFG. 55

Figure 5.1. CDFG optimization flow for the FREEDOM compiler. 59

Figure 5.2. Structural analysis on a CFG using graph minimization. 60

Figure 5.3. Adding input and output ports to a CDFG. 63

Figure 5.4. Common sub-expression elimination example. 65

Figure 5.5. Copy propagation examples. 66

Figure 5.6. Constant folding example. 67

Figure 5.7. Constant propagation example. 67

Figure 5.8. Strength reduction example. 68

Figure 5.9. Constant predicate elimination example. 69

Figure 5.10. Boolean reduction example. 69

Figure 5.11. Shift reduction example. 70

Figure 5.12. Redundant memory access elimination examples. 72

Figure 5.13. Block-set merging example. 73

Figure 5.14. Loop unrolling for a self-loop structure in a CDFG..... 74

Figure 6.1. ASAP, ALAP and Balanced scheduling routines. 85

Figure 6.2. Dependency analysis algorithm. 86

Figure 6.3. Balanced scheduling algorithm. 88

Figure 6.4. Comparison of chaining methods..... 91

Figure 6.5. Verilog code for modeling delays. 93

Figure 6.6. Measuring operation delays for FPGA designs..... 93

Figure 6.7. Comparison of delay models for a multiply operation..... 94

Figure 6.8. Balanced chaining algorithm..... 96

Figure 7.1. Reoccurring patterns in a CDFG. 106

Figure 7.2. Equivalent DAG structures. 111

Figure 7.3. Pseudo-code for DAG expressions. 112

Figure 7.4. Pseudo-code for growing nodesets..... 113

Figure 7.5. Generated nodesets and expressions. 113

Figure 7.6. Pseudo-code for reconverging paths. 114

Figure 7.7. Pseudo-code for template matching. 116

Figure 7.8. Adjacency matrix for overlapping sets..... 117

Figure 7.9. Pseudo-code for template selection. 118

Figure 7.10. Generated template for the DAG in Figure 7.5. 119

Figure 7.11. Pseudo-code for resource sharing. 120

Figure 8.1. Procedure for discovering structures for extraction. 130

Figure 8.2. Recursive procedure for identifying extractable structures. 130

Figure 8.3. Procedure for determining if a structure can be extracted. 131

Figure 8.4. GUI interface for selecting structures for extraction. 132

Figure 8.5. Recursive procedure fore extracting structures. 133

Figure 8.6. Extracted MST instructions replaced with a procedure call. 134

Figure 8.7. Verilog code for calling the extracted procedure in a FSM. 134

Figure 9.1. Software model for the MPEG-4 decoder. 138

Figure 9.2. Software implementation for the texture decoding module. 139

Figure 9.3. Software implementation for the motion compensation module. 140

Figure 9.4. Software implementation for the memory controller module. 141

Chapter 1

Introduction

Recent advances in embedded communications and control systems are driving hardware and software implementations for complete systems-on-chip (SOC) solutions, while pushing the computational limits of digital signal processing (DSP) functions. Two way radios, digital cellular phones, wireless Internet, 3G and 4G wireless receivers, MPEG4 video, voice over IP, and video over IP are examples of DSP applications that are typically mapped onto general purpose processors. Often times, the performance of these complex applications is impeded due to limitations in the computational capabilities of the processor.

The conventional way to address the computational bottleneck has been to migrate sections of the application or its entirety to an application specific integrated circuit (ASIC) as part of a hardware-software co-design system. The flexibility of an ASIC allows the designer to optimize for power consumption and functional parallelism. However, the design time and cost of such an implementation is significant. Field

Programmable Gate Arrays (FPGAs), provide an alternative to ASICs with built-in DSP functionality. FPGAs combine the programming advantage of a general purpose DSP processor with the performance advantage of an ASIC. By exploiting its inherent parallelism, it is expected that FPGAs can provide more flexible and optimal solutions for DSP applications in a hardware-software co-design system.

Generally, the path from DSP algorithms to FPGA implementations is a complex and arduous task. DSP engineers initially create high-level models and specifications for the system in high-level languages, such as C/C++ or MATLAB. Hardware design teams use these specifications to create register transfer level (RTL) models in a hardware description language (HDL), such as VHDL and Verilog. The RTL descriptions are synthesized by back-end logic synthesis tools and mapped onto the FPGAs.

In recent years, the size and complexity of designs for FPGAs and other reconfigurable hardware devices have increased dramatically. As a result, the manual approach to hardware designs for these systems has become cumbersome, leading the way to automated approaches for reducing the design time of complex systems. The problem of translating a high-level design to hardware is called *high-level synthesis* [16]. Traditionally, the high-level synthesis problem is one of transforming an abstract model in a high-level application into a set of operations for a system, in which scheduling and binding are performed to optimize the design in terms of area, cycles, frequency, and power. Recently, many researchers have developed synthesis tools that translate descriptions of systems in high-level languages such as C/C++, MATLAB, and

SIMULINK into RTL VHDL and Verilog for hardware implementations. Most high-level synthesis tools allow the designer to make optimization tradeoffs in the design, such as power, area and functional parallelism. By identifying high-level constructs, such as loops and if-then-else blocks, further optimizations can be performed, such as loop unrolling and functional partitioning. However, when high-level language constructs are not readily available, such as in the case where legacy code for an application on an older processor is to be migrated to a new processor architecture, a more interesting problem presents itself, known as *binary translation*.

1.1 Binary to Hardware Translation

Implementing a hardware/software co-design from software binaries is a complicated problem. The compiler must determine the bottlenecks in the computational portion of the software binary, automatically partition these sections and translated them to hardware, generate the proper hardware/software interface, and modify the original software binary to integrate properly with the new co-design. Moreover, the resulting co-design must show improvements over the original implementation for the system to be a worthwhile task. The hardware/software co-design problem is illustrated in Figure 1.1. Binary translation, however, has an advantage over the traditional high-level synthesis approach in that it works with any high-level language and compiler flow since the binary code is essentially the final and universal format for a specific processor. In

addition, software profiling is more accurate at the binary level as compared to the source level, allowing for better hardware/software partitioning.

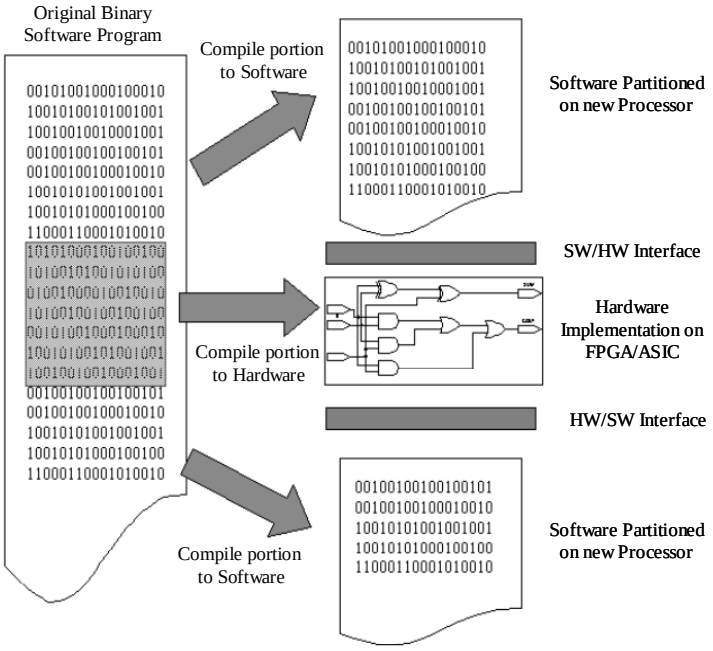


Figure 1.1. Using binary translation to implement a harware/software co-design.

Translating scheduled software binaries from a fixed processor architecture to a hardware system, such as an FPGA or ASIC, is an even more interesting and complex problem than the traditional high-level synthesis problem. A general-purpose processor consists of a fixed number of functional units and physical registers, which often necessitate the use of advanced register-reuse algorithms by compilers to handle large data sets. Consequently, the effects of memory spilling optimizations cause many designs to suffer due to wasted clock cycles from memory loads and stores. The

challenge in translating instructions from a fixed processor architecture to hardware is in undoing these optimizations by reverse-engineering the limitations in the architecture, and then exploiting the fine-grain parallelism in the design using a much larger number of functional units, embedded multipliers, registers and on-chip embedded memories.

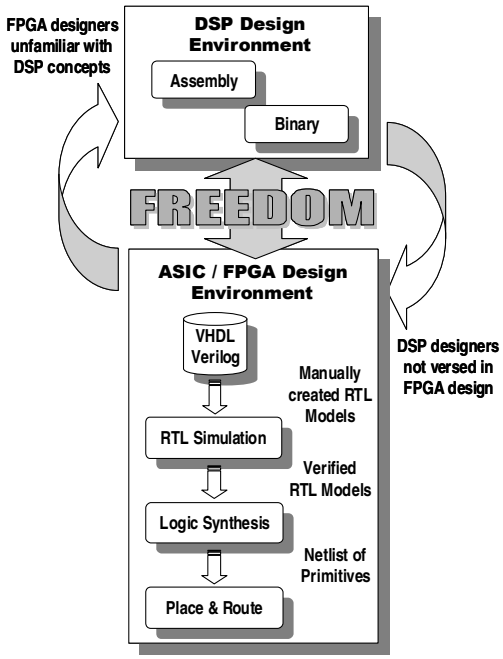


Figure 1.2. FREEDOM compiler bridging the gap between DSP and FPGA designs environments.

A manual approach to this task is quite difficult. Often times, DSP engineers are not familiar with the hardware implementation aspect of FPGAs. Conversely, hardware engineers are often not familiar with the low-level aspects of DSP applications and general purpose processors. An automated approach is often relied upon to partition or

translate software binaries. Towards this effort, we have developed the FREEDOM compiler, which automatically translates DSP software binaries to hardware descriptions for FPGAs [42][75]. “FREEDOM” is an acronym for “**F**abrication of **Re**configurable Hardware **E**nvironments from **D**SP **O**ptimized **M**achine Code.” The concept of the FREEDOM compiler is illustrated in Figure 1.2. The two architectures selected in evaluating the concepts brought forth in this research are the Texas Instruments TMS320C6211 DSP as the general-purpose processor platform and the Xilinx Virtex II FPGA as the hardware platform. A brief description of these architectures and their design flow is provided in the following sections.

1.2 Texas Instruments TMS320C6211 DSP Design Flow

The Texas Instruments C6211 DSP has eight functional units, composed of two load-from-memory data paths (LD1/2), two store-to-memory data paths (ST1/2), two data address paths (DA1/2), and two register file data cross paths (1/2X). It can execute up to eight simultaneous instructions. It supports 8/16/32-bit data, and can additionally support 40/64 bit arithmetic operations. It has two sets of 32 general-purpose registers, each 32-bits wide. It has two multipliers that can perform two 16x16 or four 8x8 multiplies every cycle. It has special support for non-aligned 32/64-bit memory access. The C6211 processor has support for bit level algorithms and for rotate and bit count hardware. Figure 1.3 illustrates the Texas Instruments C6211 DSP architecture.

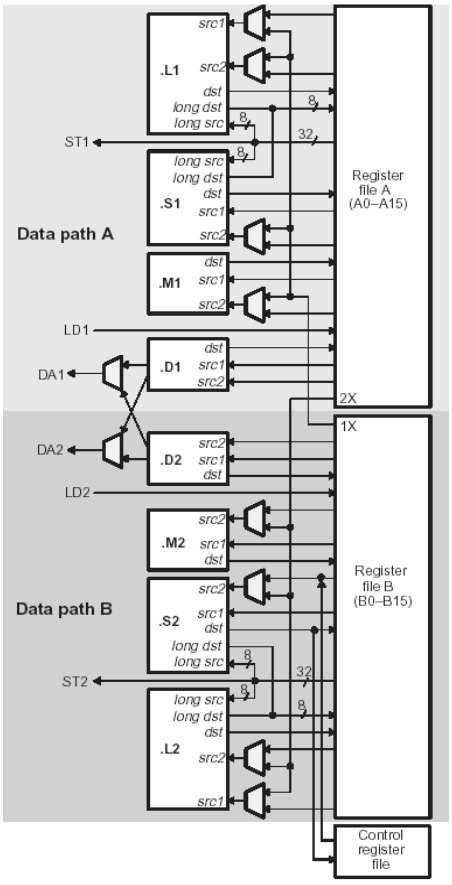


Figure 1.3. Texas Instruments C6211 DSP architecture.

The design flow for an application to be implemented on this processor usually begins with DSP engineers developing specifications and design models in high-level languages such as C/C++, MATLAB, or SIMULINK. These models are simulated and verified at that level using known or randomized data. The high-level design is then compiled to a binary for the processor. The design is once again simulated for verification of correctness using the same data as before. The binary is also profiled during simulation to determine computational bottlenecks. The DSP designers will then

go back and make adjustments to the high-level design or implement other optimizations to produce more efficient results. If the design still does not meet the necessary specifications or timing constraints, DSP engineers will often resolve to manually writing the assembly code. The development flow for the Texas Instruments C6211 DSP is illustrated in Figure 1.4.

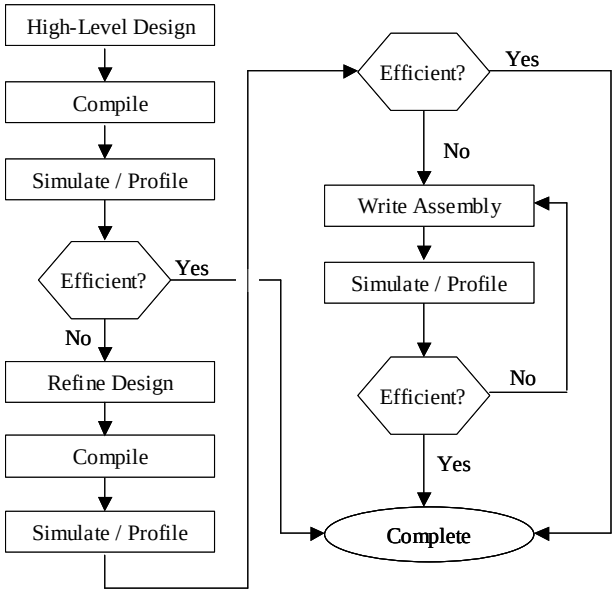


Figure 1.4. Texas Instruments C6211 DSP development flow.

1.3 Xilinx Virtex II FPGA Design Flow

The Xilinx Virtex II FPGA consists of up to 168 18x18 multipliers in a single device, supporting up to 18-bit signed or 17-bit unsigned representation, and cascading to support bigger numbers. The multipliers can be fully combinational or pipelined.

They also consist of up to 3 Mbits of embedded Block RAM, 1.5 Mbits of distributed memory and 100K logic cells. Virtex II FPGAs may contain up to 12 Digital Clock Managers (DCMs) and offers logic performance in excess of 300 MHz.

Similar to the design flow of a DSP, an application to be implemented on an FPGA usually begins with DSP engineers developing specifications and design models in high-level languages such as C/C++, MATLAB, or SIMULINK. These models are simulated and verified at that level using known or randomized data. The high-level design model is then given to hardware engineers, who manually create the hardware descriptions for the FPGA in VHDL or Verilog, or they may use high-level synthesis tools to automatically translate the high-level design to the hardware descriptions. The design is once again simulated for verification of bit-true accuracy using the same data as before, while profiling the design for computational bottlenecks. If the design does not meet the necessary specifications or timing constraints, the hardware designers will go back and make adjustments to the hardware descriptions, or implement other optimizations to produce more efficient results. Once the simulation results meet the specification requirements, the hardware descriptions are synthesized into netlist of gates for the target FPGA using logic synthesis tools. The design is once again simulated for verification at this level. If errors occur in the post-synthesis simulation or design specifications are not met, the hardware engineers must re-evaluate the hardware descriptions and begin the process again. After post-synthesis simulation has passed all requirements and verification, the netlist of gates are placed and routed on the FPGA

using back-end tools. Timing analysis and verification is performed once again at this level. If errors occur or timing constraints are not met, the hardware engineers must re-evaluate the hardware descriptions once again. The development flow for the Xilinx Virtex II FPGA is illustrated in Figure 1.5.

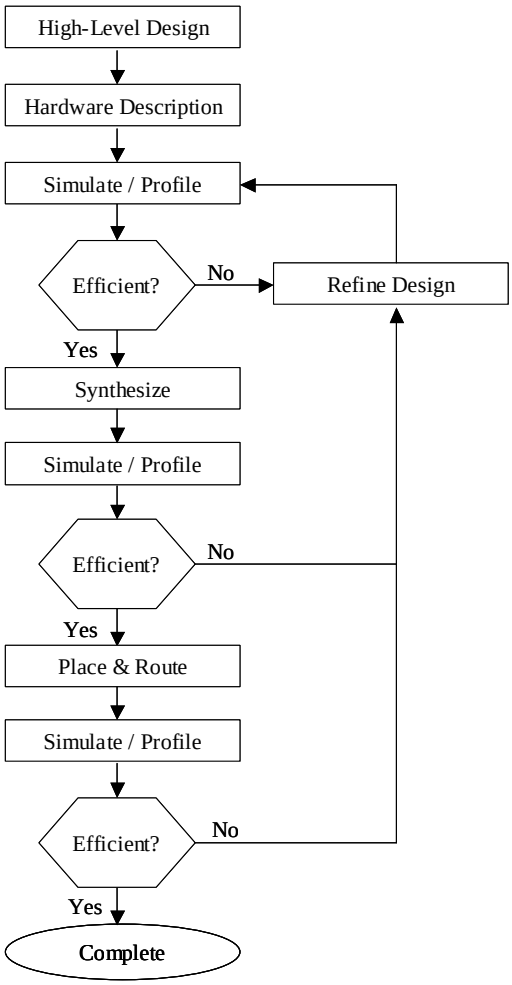


Figure 1.5. Xilinx Virtex II FPGA development flow

1.4 Motivational Example

In order to better understand the complexity of translating software binaries and assembly code to hardware, consider the example Texas Instruments C6211 DSP assembly code in Figure 1.6. The processor has eight functional units (L1, M1, etc.), and may therefore execute at most eight instructions in parallel. In the example code, the MPY instructions require two cycles to execute and all other instructions require one cycle; hence, the instruction following the MPY is a NOP in this example. The `||` symbol in certain instructions signify that the instruction is executed in parallel with the previous instruction. As a result, the section of code requires seven cycles to execute.

```
        MV     .L1    A0,A1
||      MV     .L2    B0,B1
        MPY    .M1    A1,A2,A3
||      MPY    .M2    B1,B2,B3
        NOP           1
        MPY    .M1    A3,A6,A7
||      MPY    .M2    B3,B6,B7
        NOP           1
        ADD    .L1X   A7,B7,A8
        ADD    .L1    A4,A8,A9
```

Figure 1.6. Example TI C6000 DSP assembly code.

A simple translation of this code onto an FPGA, by assigning one operation per state in an RTL finite state machine, would result in no performance benefit. The design would require eight cycles to complete on an FPGA since there are eight instructions, excluding NOPs. Rather, one must explore the parallelism in the design through

scheduling techniques and other optimizations in order to reduce the design complexity and exploit the fine-grain parallelism inherent in the FPGA architecture, thereby reducing the number of execution clock cycles.

1.5 Dissertation Overview

The key contribution of this research is twofold: To provide a description of the process and considerations for automatically translating software binaries from a general-purpose processor into RTL descriptions for implementation on FPGAs. Included in this work is an array of optimizations and scheduling routines, as well as a description of important concepts to consider when translating software binaries to hardware. Additionally, the true test is in experimentally evaluating the quality of the synthesized software binaries in terms of area and performance on FPGAs as compared to general-purpose processor implementations. The research described in this dissertation is a collaborative work with Gaurav Mittal [41].

The work in this dissertation is presented as follows. In Chapter 2, a survey of related work is presented. Chapter 3 presents a detailed overview of the FREEDOM compiler infrastructure, which translates software binaries to hardware descriptions for FPGAs. Chapter 4 discusses the intricacies in performing control and data flow analysis of scheduled and pipelined software binaries. Chapter 5 provides an overview of the optimizations implemented in the FREEDOM compiler. Chapter 6 presents new

scheduling and operation chaining methods for FPGA designs. In Chapter 7, a resource sharing optimization is presented. Chapter 8 discusses a method of hardware/software partitioning using structural extraction. Chapter 9 provides a case study, using the MPEG-4 CODEC as an example application translated from software to hardware using the methods described in the previous chapters. Finally, Chapter 10 presents the conclusions and future work.

Chapter 2

Related Work

The research presented in this dissertation is focused in three main areas of interest: *high-level synthesis*, *binary translation*, and *hardware-software co-design*. The FREEDOM compiler unifies all three of these areas of research, in that it performs high-level synthesis on software binaries for implementation on FPGA architectures in a hardware-software co-design. The following sections discuss some recent advances in these areas of research.

2.1 High-Level Synthesis

The problem of translating a high-level or behavioral language description into a register transfer level representation has been explorer extensively in research and industry. Synopsys developed one of the first successful commercial behavioral synthesis tools in the industry, the *Behavioral Compiler*, which translates behavioral VHDL or Verilog and into RTL VHDL or Verilog. The *Princeton University Behavioral*

Synthesis System [67] is another system that translated behavioral VHDL models and processes into RTL implementations.

Over the years, there has been more work in developing compilers that translate high-level languages into RTL VHDL and Verilog. Many of these tools are available as commercial products by electronic design automation (EDA) companies. Adelante, Celoxica, and Cynapps are a few of the many companies that offer synthesis tools for translating C code to RTL descriptions. Haldar et al. [28] described work on the MATCH compiler for translating MATLAB functions to RTL VHDL for FPGAs, which is now offered as a commercial tool by AccelChip [3]. There have been some system level tools that translate graphical descriptions of systems to RTL descriptions. Examples include *SPW* from Cadence, *System Generator* from Xilinx, and *DSP Builder* from Altera. Other languages have been used in high-level synthesis design flows, such as SystemC, a relatively new language that allows users to write hardware system descriptions using a C++ class library. HardwareC is a hardware description language with syntax similar to C that models system behavior, and was used in the *Olympus Synthesis System* [15] for synthesizing digital circuit designs.

Recent studies in high-level synthesis have focused on incorporating low-level design optimizations and trade-offs in the high-level synthesis process. Dougherty and Thomas [20] presented work on unifying behavioral synthesis and physical design using transformation that act as forces upon each other to guide the decisions in scheduling, allocation, binding, and placement to occur simultaneously. Gu et al. [24] described an

incremental high-level synthesis system that integrates high-level and physical design algorithms to concurrently improve a system's schedule, resource binding, and floorplan. Through incremental exploration of both the behavioral-level and physical-level design space, one is able to reduce the synthesis time, area and power consumption of the system. Bringmann et al. [5] combined partitioning and an extended interconnection cost model in high-level synthesis for multi-FPGA systems with the objective to synthesize a prototype with maximal performance under the given area and interconnection constraints of the target architecture. Peng et al. [51] described methods for automatic logic synthesis of sequential programs and high-level descriptions of asynchronous circuits into fine-grain asynchronous process netlists for high-performance asynchronous FPGA architectures.

In recent years, power and thermal optimizations have played an essential role in high-level synthesis research. Musoll and Cortadella [46] evaluated several power optimizations at high levels during synthesis. Jones et al. [32] described work on the PACT compiler that focuses on power optimizations in translating C code to RTL VHDL and Verilog for FPGAs and ASICs. Chen et al. [7] developed the LOPASS high-level synthesis system for reducing power consumption in FPGA designs using a framework for RTL power estimates that considers wire length. Stammermann et al. [58] described a power optimizations during high-level synthesis in which floorplanning, functional unit binding, and allocation are implemented simultaneously to reduce the interconnect power in circuits. Lakshminarayana and Jha [37] presented an iterative

improvement technique for synthesizing hierarchical DFGs into circuits optimized for power and area. Mukherjee et al. [45] investigated temperature-aware resource allocation and binding techniques during high-level synthesis to minimize the maximum temperature that can be reached by a resource in a design in order to prevent hot spots and increase reliability of integrated circuits.

All of the research studies mention here focused mainly on the conventional high-level synthesis approach in which hardware implementations are generated from high-level applications and the source code was available. In contrast to this traditional approach, the FREEDOM compiler translates software binaries and assembly language codes into RTL VHDL and Verilog for FPGAs. In such cases, the source code and high-level information used in most of these optimizations and techniques may not be readily available, which makes binary translation to hardware an even more interesting problem.

2.2 Binary Translation

There has been a great deal of fundamental research and study of binary translation and decompilation. Cifuentes et al. [10][11][12] described methods for converting assembly or binary code from one processor's instruction set architecture (ISA) to another, as well as decompilation of software binaries to high-level languages. Kruegel et al. [36] have described a technique for decompilation of obfuscated binaries. Dehnert et al. [19] presented work on a technique called *Code Morphing*, in which they

produced a full system-level implementation of the Intel x86 ISA on the Transmeta Crusoe VLIW processor.

In related work on dynamic binary optimizations, Bala et al. [2] developed the Dynamo system that dynamically optimizes binaries for the HP architecture during run-time. Gschwind et al. [27] developed a similar system for the PowerPC architecture called the Binary-translation Optimized Architecture (BOA). Levine and Schmidt [38] proposed a hybrid architecture called HASTE, in which instructions from an embedded processor are dynamically compiled onto a reconfigurable computational fabric during run-time using a hardware compilation unit to improve performance. Ye et al. [72] developed a similar compiler system for the Chimaera architecture, which contains a small, reconfigurable functional unit integrated into the pipeline of a dynamically scheduled superscalar processor.

2.3 Hardware-Software Co-Designs

The choice of a hardware-software architecture requires balancing of many factors, such as allocation of operations or portions of the design to be implemented on each processing element, inter-processor communication, and device cost, among others. De Micheli et al. [15][16][17] and Ernst [22] have discussed much of the fundamental aspects of hardware/software co-designs.

Generally, hardware/software co-designs are partitioned at the task or process level. Vallerio and Jha [63] presented work on a task graph extraction tool for hardware/software partitioning of C programs. Xie and Wolf [70] described an allocation and scheduling algorithm for a set of data-dependent tasks on a distributed, embedded computing system consisting of multiple processing elements of various types. The co-synthesis process synthesizes a distributed multiprocessor architecture and allocates processes to the target architecture, such that the allocation and scheduling of the task graph meets the deadline of the system, while the cost of the system is minimized. Gupta and De Micheli [26] developed an iterative improvement algorithm to partition real-time embedded systems into a hardware-software co-design implementation based on a cost model for hardware, software, and interface constraints. Wolf [68] described work on a heuristic algorithm that simultaneously synthesizes the hardware and software architectures of a distributed system that meets all specified performance constraints and simultaneously allocates and schedules the software processes onto the processing elements in the distributed system. Xie and Wolf [69] introduced a hardware/software co-synthesis algorithm that optimizes the implementation of distributed embedded systems by selecting one of several possible ASIC implementations for a specific task. They use a heuristic iterative improvement algorithm to generate multiple implementations of a behavioral description of an ASIC and then analyze and compare their performances.

Others have tackled more fine-grain partitioning of hardware/software co-designs. Li et al. [40] developed the Nimble compilation tool that automatically compiles system-level applications specified in C to a hardware/software embedded reconfigurable architecture consisting of a general-purpose processor, an FPGA, and a memory hierarchy. The hardware/software partitioning is performed at the loop and basic-block levels, in which they use heuristics to select frequently executed or time-intensive loop bodies to move to hardware.

Another relative area of research is hardware-software partitioning of software binaries. Stitt and Vahid [60] reported work on hardware-software partitioning of software binaries, in which they manually translated software binary kernels from frequently executed loops on a MIPS processor and investigated their hardware implementations on a Xilinx Virtex FPGA. More recently, Stitt and Vahid [59] have reported work on fast dynamic hardware/software partitioning of software binaries for a MIPS processor, in which they automatically map kernels consisting of simple loops onto reconfigurable hardware. The hardware used was significantly simpler than commercial FPGA architectures. Consequently, their approach was limited to combinational logic structures, sequential memory addresses, pre-determined loop sizes, and single-cycle loop bodies.

In contrast to these methods, the FREEDOM compiler can translate entire software binaries or portions at a block or structural level to hardware implementations on FPGAs as both stand-alone designs and hardware/software co-designs.

Chapter 3

The FREEDOM Compiler

This chapter provides an overview of the FREEDOM compiler infrastructure [42][75], as illustrated in Figure 3.1. The FREEDOM compiler was designed to have a common entry point for all assembly languages. Towards this effort, the front-end uses a description of the source processor's ISA in order to configure the assembly language parser. The ISA specifications are written in SLED from the New Jersey Machine-Code toolkit [54][55]. The parser translates the input source code into an intermediate assembly representation called the *Machine Language Syntax Tree (MST)*. Simple optimizations, linearization and procedure extraction [43] are implemented at the MST level. A *control and data flow graph (CDFG)* is generated from the MST instructions, where more complex optimizations, scheduling, and resource binding are preformed. The CDFG is then translated into an intermediate high-level *Hardware Description Language (HDL)* that models processes, concurrency, and finite state machines. Additional optimizations and customizations for the target architecture are performed on

the HDL. This information is acquired via the *Architecture Description Language (ADL)* files. The HDL is translated directly to RTL VHDL and Verilog to be mapped onto FPGAs, and a testbench is generated to verify bit-true accuracy in the design. A *graphical user interface (GUI)* was design to manage projects and compiler optimizations for the designs.

The following sections provide an overview for different aspects of the compiler, including the MST, the CDFG, and the HDL. A brief description of the GUI is also presented, as well as the techniques for verification of the generated designs.

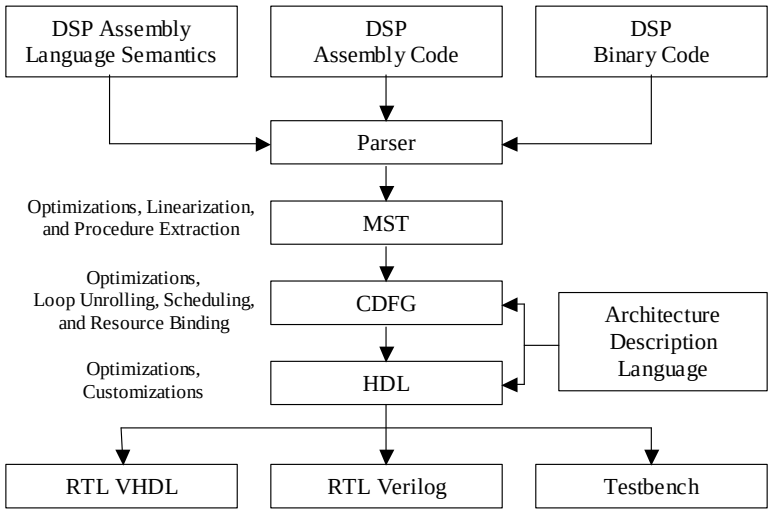


Figure 3.1. Overview of the FREEDOM compiler infrastructure.

3.1 The Machine Language Syntax Tree

The Machine Language Syntax Tree (MST) is an intermediate language whose syntax is similar to the MIPS ISA. The MST is generic enough to encapsulate most

ISAs, including those that support predicated and parallel instruction sets. An MST design is made up of procedures, each containing a list of instructions. The MST grammar is described in detail in Appendix A.

3.1.1 MST Instructions

Many advanced general-purpose processors handle predicated operations as part of the ISA. To accommodate for these complex ISA, all MST instructions are three-operand, predicated instructions in the format: *[pred] op src1 src2 dst*, syntactically defined as: *if (pred=true) then op (src1, src2) → dst.* MST operands are composed of four types: Register, Immediate, Memory, and Label types. MST operators are grouped into six categories: Logical, Arithmetic, Compare, Branch, Assignment, and General types. Table 3.1 lists the supported operations in the MST language.

Table 3.1. Supported operations in the MST grammar.

Logical	Arithmetic	Compare	Branch	Assignment	General
AND	ADD	CMPEQ	BEQ	LD	NOP
NAND	DIV	CMPGE	BGEQ	MOVE	
NOR	MULT	CMPGT	BGT	ST	
NOT	NEG	CMPLE	BLEQ	UNION	
OR	SUB	CMPLT	BLT		
SLL		CMPNE	BNEQ		
SRA			CALL		
SRL			GOTO		
XNOR			JMP		
XOR					

3.1.2 Control Flow Analysis

An MST procedure is a self-contained group of instructions, whose control flow is independent of any other procedure in the design. Intra-procedural control flow is altered using branch type instructions, such as BEQ, GOTO, JMP, etc. The destination operand for these branch operations may be a Label, Register or Immediate value. Inter-procedural control is transferred from one procedure to another using the CALL operation. The destination operand of the CALL instruction must be a label containing the name of procedure, function or library.

3.1.3 Data Dependency Analysis

The fixed number of physical registers on a processor necessitates advanced register reuse algorithms in compilers. These optimizations often introduce false dependencies based on register names, resulting in difficulties when determining correct data dependencies. This is especially true when dealing with scheduled or pipelined binaries and parallel instruction sets. To resolve these discrepancies, each MST instruction is assigned a timestep, specifying a linear instruction order, and an operation delay, equivalent to the number of execution cycles. Each cycle begins with an integer-based timestep, *T*. Each instruction *n* in a parallel instruction set is assigned the timestep $T_n = T + (0.01 * n)$. Assembly instructions may be translated into more than one MST instruction. Each instruction *m* in an expanded instruction set is assigned the timestep

$T_m = T_n + (0.0001 * m)$. The write-back time for the instruction, or the cycle in which the result is valid, is defined as *wb = timestep + delay*. If an operation delay is zero, the resulting data is valid instantaneously. However, an operation with delay greater than zero has its write-back time rounded down to the nearest whole number, or *floor(timestep)*, resulting in valid data at the beginning of the write-back cycle.

Figure 3.2 illustrates how the instruction timestep and delay are used to determine data dependencies in the MST. In the first instruction, the MULT operation has one delay slot, and the resulting value in register A4 is not valid until the beginning of cycle 3. In cycle 2, the result of the LD instruction is not valid until the beginning of cycle 7, and the result of the ADD instruction is not valid until the beginning of cycle 3. Consequently, the ADD instruction in cycle 3 is dependant upon the result of the MULT operation in cycle 1 and the result of the ADD operation in cycle 2. Likewise, the first three instructions are dependant upon the same source register, A4.

```
TIMESTEP    PC      OP    DELAY  SRC1    SRC2    DST
 1.0000   0X0020   MULT  (2)    $A4,     2,     $A4
 2.0000   0X0024   LD    (5)   *($A4),          $A2
 2.0100   0X0028   ADD   (1)    $A4,     4,     $A2
 3.0000   0X002c   ADD   (1)    $A4,     $A2,   $A3
```

Figure 3.2. MST instructions containing timesteps and delays for determining data dependencies.

3.2 The Control and Data Flow Graph

Each MST procedure is converted into an independent control and data flow graph (CDFG). A control flow graph is a block-level representation of the flow of control in a procedure, designated by the branch operations. A data flow graph is made up of nodes connected via edges, which represents the data dependencies in the procedure. Using the write-back times (*wb* = *timestep* + *delay*) for each operation, one may calculate the data dependencies for each MST instruction, as described in Section 3.1.3.

```
DOTPROD: MVK   .S1  500,A1
         ZERO  .L1  A7
         MVK   .S1  2000,A3
LOOP:    LDW   .D1  *A4++,A2
         LDW   .D1  *A3++,A5
         NOP        4
         MPY   .M1  A2,A5,A6
         SUB   .S1  A1,1,A1
         ADD   .L1  A6,A7,A7
    [A1] B     .S2  LOOP
         NOP        5
         STW   .D1  A7,*A3
```

Figure 3.3. TI assembly code for a *dot product* function.

The nodes in the CDFG are distinguished in five different types: *Constants*, *Variables*, *Values*, *Control*, and *Memory*. Constant and Variable nodes are inputs to operations. Value nodes are operation nodes, such as addition and multiplication. Control nodes represent branch operations in the control flow. Memory nodes refer to

memory read and write operations. Figure 3.3 shows the TI assembly code for a *dot product* procedure, while Figure 3.4 illustrates the CDFG representation generated by the FREEDOM compiler.

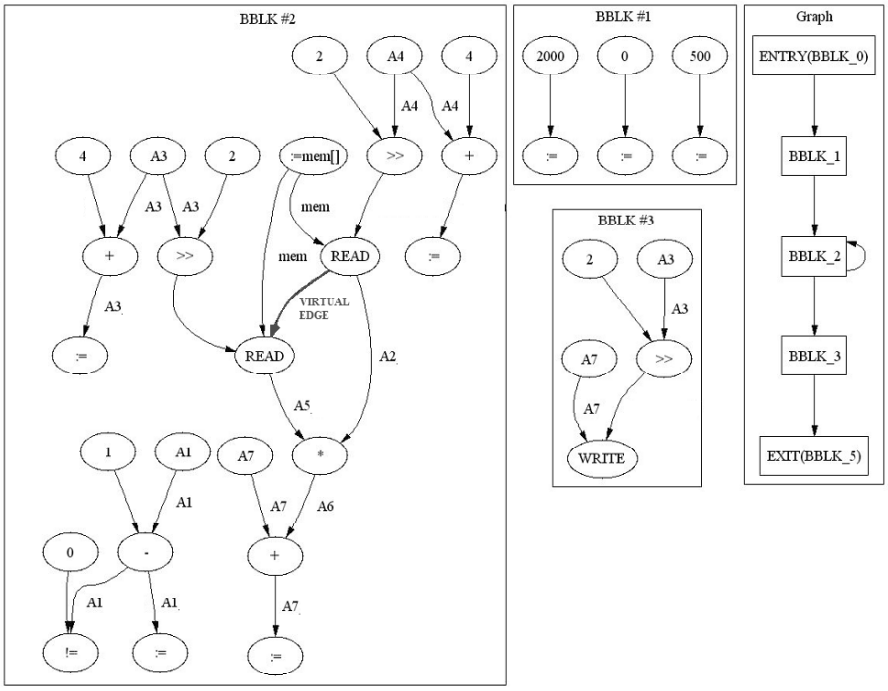


Figure 3.4. CDFG representation for a *dot product* function.

3.2.1 Memory Operations

Memory read and write operations are represented in the CDFG as single node operations with two input edges from source nodes. The source nodes for a read operation consist of an address and a memory variable, while the source nodes for a write operation consist of an address and the value to be written. The memory element to which these operations read and write are distinguished by the name of the memory

variable. In other words, each memory variable is mapped to independent memory elements. Essentially, memory partitioning may be accomplished by changing the memory variable name in the read and write operations.

When rescheduling operations for an FPGA design, it is possible that the ordering sequence of memory operations may shift, resulting in memory hazards. Additionally, some FPGAs do not allow more than one memory operation in a single cycle. To ensure proper scheduling and that memory operations occur in the correct linear sequence, virtual edges are added from each read and write operation to subsequent memory operations. This is demonstrated in block 2 in Figure 3.4 above, where a virtual edge is added between the two memory read operations.

3.2.2 Predicate Operations

Predicated instructions are operations that are conditionally executed under a predicate condition. Predicated operations are handled in the CDFG by adding the predicate operand as a source node to the operation node. Additionally, an edge from the previous definition of the destination node is also added as a source node in order to ensure proper data dependencies. The result is an *if-then-else* statement, which assigns the previous definition to the destination operand if the predicate condition is not met. This is illustrated in Figure 3.5.

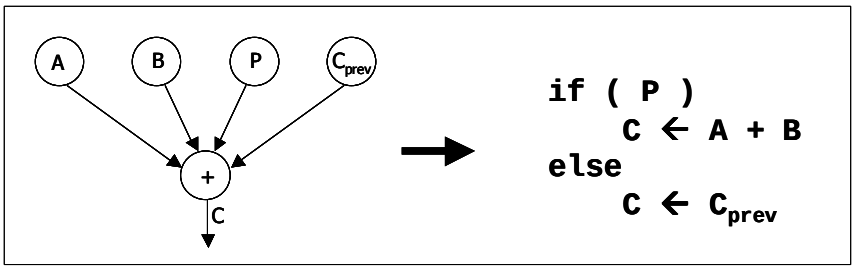


Figure 3.5. Predicated operation in the CDFG.

Predicated operations pose an interesting problem when optimizing a CDFG. Because the result of the predicate condition is generally non-deterministic at compile time, predicated operations effectively prevent many optimizations from ensuing. Additionally, the extra multiplexer logic increases the area in the design. Although this method introduces limitations in the optimizations, it does however increase the opportunity for greater parallelism in the design.

Considering this problem, two methods have been explored to eliminate predicates on operations. The first method replaces the predicate with a branch operation that jumps over the instruction if the predicate condition is not met. This resulted in a significant number of disruptions in the data flow, effecting the overall efficiency and parallelism in the design. The second method converts predicates into a set of instructions that model a multiplexer, in which the predicate condition selects either the new value or the old value to pass through. This method resulted in a significant increase in instructions, area and clock cycles. It was therefore determined that the original method is the most efficient way to handle predicates in the CDFG.

3.2.3 Procedure, Function and Library Calls

Procedure, function and library calls are handled in the CDFG by a CALL node. The input and output edges to a CALL node represent the I/O ports for the procedure. Any variable or memory read from in the procedure becomes an input port; any variable or memory written to in the procedure become output ports. Memory operands are always set up as both input and output nodes in order to ensure the proper sequence of memory operations. In Figure 3.6, an example CDFG is shown in which the *dot product* procedure is called twice. A memory dependency exists between the two procedure calls, preventing the two functions from running in parallel, as well as any read/write hazards.

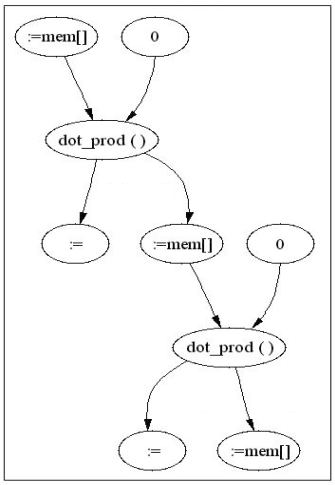


Figure 3.6. CDFG representation of a CALL operation.

3.3 The Hardware Description Language

The Hardware Description Language (HDL) is an intermediate low-level language, which models processes, concurrency, and finite state machines. Syntactically, the HDL is similar to the VHDL and Verilog grammar. The HDL grammar is described in detail in Appendix B.

Each CDFG in a design is mapped to an individual HDL Entity. In its simplest form, each CDFG operation node is assigned to an individual state in an HDL finite state machine. However, by performing scheduling techniques, one may exploit the parallelism in the design by increasing the number of instructions assigned to each state, thereby reducing the total number of execution cycles in the design. Additional optimizations and customizations are performed on the HDL to enhance the efficiency of the output and to correctly support the target device's architecture.

3.3.1 Memory Support

Memory models are generated in the HDL as required by backend synthesis tools to automatically infer both synchronous and asynchronous RAMs. These memory elements are used as FIFO buffers as communication between a co-processor and the hardware. Pipelining is used for the memory operations to improve the throughput performance in the design.

3.3.2 Procedure Calls

Each CDFG that is translated to HDL is expressed as a completely synchronous finite state machine (FSM) process within an Entity model. A procedure that is called from another is represented as an instantiation of that HDL entity. Figure 3.7(a) shows the HDL for an instantiation of the *dot product* procedure. The process is controlled via I/O signals; it is initialized by setting the *reset* signal high in the first state, and cycles in the next state until the function concludes when the *done* signal is set. To prevent memory contention between processes, an asynchronous process is set up as a multiplexer to distribute memory control between all processes. The memory control is enabled by assigning a select value to the *mem_mux_ctrl* signal just before a process begins, as show in Figure 3.7(b). The HDL model for the memory MUX control process is shown in Figure 3.8.

```
dot_prod : dot_prod_1
(
    clock <= clock
    reset <= dot_prod_1_reset
    done <= dot_prod_1_done
    DP <= 32'hs00000000
    A0 <= A0
    mem_addr <= dot_prod_1_mem_addr
    mem_d <= dot_prod_1_mem_d
    mem_q <= dot_prod_1_mem_q
    mem_we <= dot_prod_1_mem_we
);
```

(a) Process for *dot product* function.

```
state: 0
   mem_mux_ctrl <= 1
   dot_prod_1_reset <= 1
   State <= main_process_1

state: 1
   dot_prod_1_reset <= 0
   if (dot_prod_1_done == 1)
       mem_mux_ctrl <= 0
       state <= 2
   else
       state <= 1
```

(b) FSM for running *dot product*.

Figure 3.7. HDL representation for calling the *dot product* function.

```
process mem_mux_process( mem_mux_ctrl, mem_q, dot_prod_0_mem_addr,
        dot_prod_0_mem_d, dot_prod_0_mem_we, dot_prod_1_mem_addr,
        dot_prod_1_mem_d, dot_prod_1_mem_we )
begin
   dot_prod_0_mem_q <= 32'bXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   dot_prod_1_mem_q <= 32'bXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

   switch( mem_mux_ctrl )
      case 0:
         mem_addr <= dot_prod_0_mem_addr
         mem_d <= dot_prod_0_mem_d
         dot_prod_0_mem_q <= mem_q
         mem_we <= dot_prod_0_mem_we
      case 1:
         mem_addr <= dot_prod_1_mem_addr
         mem_d <= dot_prod_1_mem_d
         dot_prod_1_mem_q <= mem_q
         mem_we <= dot_prod_1_mem_we
      default:
         mem_addr <= 10'bXXXXXXXXXX
         mem_d <= 32'bXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         dot_prod_0_mem_q <= 32'bXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         mem_we <= 1'bX
         dot_prod_1_mem_q <= 32'bXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   end switch
end process
```

Figure 3.8. HDL process for asynchronous memory MUX controller.

3.4 The Graphical User Interface

The graphical user interface (GUI), shown in Figure 3.9, was designed as an interactive means for interfacing with the FREEDOM compiler. It allows for easy management of projects and access to all related files via the *project workspace* window (left). The *file view* shows the source files included in the project and those generated by the compiler, while the *model view* shows the procedures in the design listed topologically in a call graph tree. All compilation information and messages, such as

warnings and errors, are displayed in the *log window* (bottom). Files opened in the GUI are displayed using syntax highlighting of keywords specific to the file type (right). The tool and menu bars allow easy access to the tools and settings for projects and the GUI.

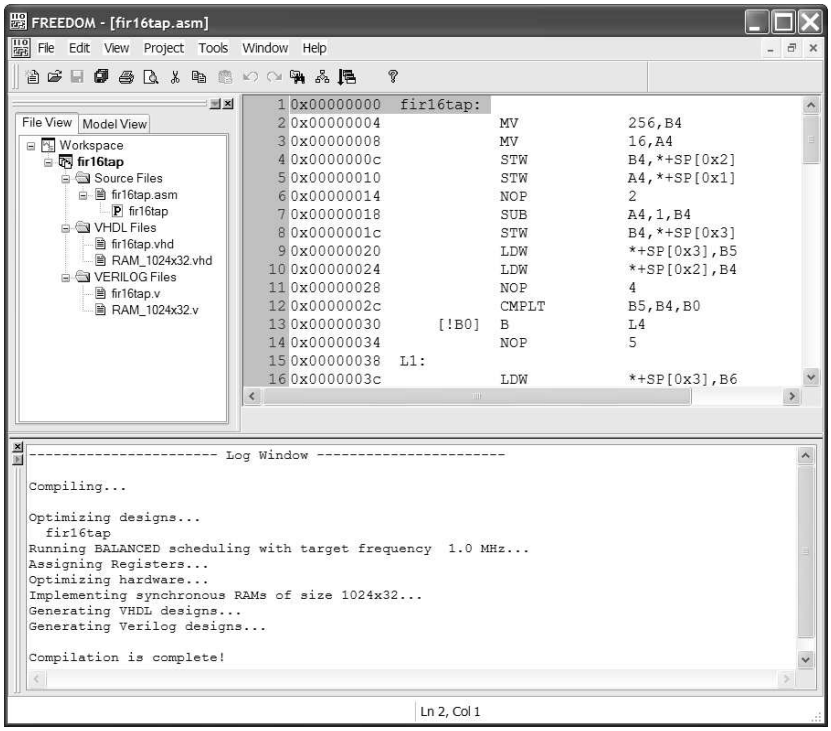


Figure 3.9. Graphical user interface for the FREEDOM compiler.

3.5 Verification

Verification is an integral part of high-level synthesis, generally performed in simulation. When implementing a design in hardware, verification must be performed at three levels: at the design stage, after synthesis, and once again after place and route for timing verification. As part of this process, input and output data must be captured in

order to verify correctness and bit-true accuracy. The following subsections discuss methods of verification for the FREEDOM compiler in more detail.

3.5.1 Verifying Bit-True Accuracy

It is essential that the RTL designs generated by the compiler be verified as bit-true accurate. This is accomplished by capturing input and output data at the source level and comparing results from the RTL simulation against the original output data.

Generating input/output testbench data for DSP designs may be approached using one of two methods. Generally, most DSP application designers implement their designs in a high-level language such as MATLAB or C/C++; the designs are then compiled into binaries for the specific DSP processor. One may easily capture or generate input/output data at this high-level stage using print statements. Alternatively, most general-purpose processors have simulation and verification tools. For instance, Texas Instrument's *Code Composer Studio* software allows one to probe points within a simulation to verify the correctness of data in memory. The file I/O allows the user to write memory data to a file at any point during the simulation. One may use this method to capture input data before a function is called, and capture output data once the function is complete. This data may then be used in during simulation of the hardware models to verify the bit-true accuracy.

3.5.2 Testbenches

RTL VHDL and Verilog designs are generally verified using simulation tools such as *ModelSim* by Mentor Graphics. Input and output data are captured and used for verification of design correctness. The designs require a top-level testbench model, which runs the design and handles the clocking, reset, and I/O data during simulation. Currently, a general template is used for all testbenches. The testbench includes a memory model, which is loaded with the input data from a file before the simulation begins. Upon completion, the output data is read from memory and written to an output file. One may then compare the contents in this file against the original captured output data. An example testbench written in Verilog may be found in Appendix C.

3.6 Summary

This chapter provided an overview of the FREEDOM compiler infrastructure. The front end is a dynamic parser that may be configured to allow multiple assembly languages as input. The input assembly language is converted to the MST, a virtual machine language in which data dependency analysis is performed. The CDFG is constructed from the MST, and is the central point in the compiler where optimizations and scheduling are performed. The CDFG is converted into the HDL, where hardware customizations are performed using the resource information from ADL files. A GUI

was developed to allow user manageability of projects and selecting optimizations. A methodology for bit-true verification was also described.

Chapter 4

Building a Control and Data Flow Graph from Scheduled Assembly

In the high-level synthesis process, an abstract design is usually converted into a control and data flow graph (CDFG), composed of nodes representing inputs, outputs, and operations. The CDFG is a fundamental component of most high-level synthesis tools, where most optimizations and design decisions are performed to improve frequency, power, timing, and area. Building a CDFG consists of a two-step process: building the control flow graph (CFG), which represents the path of control in the design, and building the data flow graph (DFG), which represents the data dependencies in the design. Much research has been performed on CDFG generation from software binaries and assembly code. However, there has been very little work on generating complete CDFGs from scheduled or pipelined software binaries. Data dependency analysis of such binaries is more challenging than that of sequential binaries.

```
0x0000 VECTORSUM:   ZERO  A7
0x0004              LDW   *A4++, A6   ; 4 delay slots
0x0008      ||      B     LOOP        ; 5 delay slots
0x000C              LDW   *A4++, A6
0x0010      ||      B     LOOP
0x0014              LDW   *A4++, A6
0x0018      ||      B     LOOP
0x001C              LDW   *A4++, A6
0x0020      ||      B     LOOP
0x0024              LDW   *A4++, A6
0x0028      ||      B     LOOP
0x002C      ||      SUB   A1, 4, A1
0x0030 LOOP:        ADD   A6, A7, A7
0x0034      || [A1] LDW   *A4++, A6
0x0038      || [A1] SUB   A1, 1, A1
0x003C      || [A1] B     LOOP        ; branches executes here
0x0040              STW   A7, *A5
0x0044              NOP   4
```

Figure 4.1. TI C6000 assembly code for a pipelined *vectorsum* procedure.

When translating assembly codes from digital signal processors (DSPs), it is common to encounter highly pipelined software binaries that have been optimized manually or by a compiler. Consider the Texas Instrument C6000 DSP assembly code for the *vectorsum* function in Figure 4.1. In this architecture, branch operations contain 5 delay slots, and loads contain 4 delay slots. The || symbol indicates the instruction is executed in parallel with the previous instruction and the [ ] symbol indicates the operation is predicated on an operand. Clearly, the *vectorsum* code is highly pipelined; each branch instruction is executed in consecutive iterations of the loop. Moreover, the dependencies of the ADD instruction in the loop body change with each iteration of the loop: in the first iteration of the loop A6 is dependent on the load at instruction 0x0004, in the second iteration of the loop A6 is dependent on the load at instruction 0x000C,

etc. Generating a CDFG to represent this pipelined structure is very challenging. In doing so, one must consider the varying data dependencies and also ensure that each branch is executed at its proper time and place. Branch instructions that fall within the delay slots of other branch instructions complicate the structure of the control flow graph. For instance, when the predicate condition, A1, on the branch instruction in the loop body is *false*, the previous branch instructions that were encountered during the execution sequence will continue to propagate and execute. This may occur within the loop, or possibly after exiting the loop. More complex software pipelines may contain branch instructions with various targets, producing multiple exit points in a CDFG block.

In this chapter, a methodology for generating CDFGs from scheduled and pipelined assembly code is presented. This process consists of three stages: generating a control flow graph, linearizing the assembly code, and generating the data flow graph. We use the methods described by Cooper et al. [14] for generating a CFG from scheduled assembly code. We extend their work to support more complex architectures that employ parallel instruction sets and dynamic branching. We also present a linearization process, in which pipelined structures are serialized into linear assembly. This allows for proper data dependency analysis when constructing data flow graphs. This methodology was incorporated in the FREEDOM compiler, which translates DSP assembly code into hardware descriptions for FPGAs. Preliminary work on this research

has been published [74]; here we present a more in-depth and detailed description of the research.

4.1 Related Work

There has been some fundamental research and study on control and data flow analysis, an essential aspect in binary translation. Cifuentes et al. [12] described methods of control and data flow analysis in translating assembly to a high-level language. Kastner and Wilhelm [33] reported work on generating CFGs from assembly code. Decker and Kastner [18] described a method of reconstructing a CFG from predicated assembly code. Amme et al. [1] presented work on a memory aliasing technique, in which data dependency analysis is computed on memory operations using a value-based analysis and modified version of the GCD test [4].

There has been very little work on generating CDFGs from highly pipelined software binaries in which branch instructions appear in the delay slots of other branch instructions. The most comprehensive work on building CFGs from pipelined assembly code was reported by Cooper et al. [14]. However, their method does not consider the complexities of modern processor architectures that utilize instruction-level parallelism and dynamic branching techniques. In this paper, we address these issues and present methods to handle CDFG generation from software binaries that feature these sophisticated scheduling techniques.

4.2 Generating a Control Flow Graph

Cooper et al. [14] presented a three-step process for building a CFG from scheduled assembly code, which was used as the first stage in the proposed work. The first step of their algorithm partitions the code at labels (entry points) into a set of basic blocks. During this process, they assume all entry points are complete, and no branch targets an instruction without a label. The second step adds edges between basic blocks in the CFG to represent the normal flow of control. Here, they only consider non-pipelined branch instructions, or those that do not appear within the delay slots of other branch instructions. Pipelined branches are handled in the third step using an iterative algorithm that simulates the flow of control for the program by propagating branch and counter information from block to block. Their method is shown to terminate in linear time for CFGs containing only branches with explicit targets. Figure 4.2 illustrates the CFG generated for the *vectorsum* procedure in Figure 4.1.

In practice, the assumptions made in their work pose some difficulties in generating CFGs for some modern processor architectures. For instance, they assume all labels and branch targets are well defined. However, some disassemblers limit the labels to a procedure level only and refrain from including them locally within procedure bounds. In some architectures, registers may be used in branch targets, as in the case of a long jump where a static PC value is loaded into the register prior to the branch

instruction. To handle these situations, we introduce a pre-processing step that determines all static branch targets and adds the respective labels to the instructions.

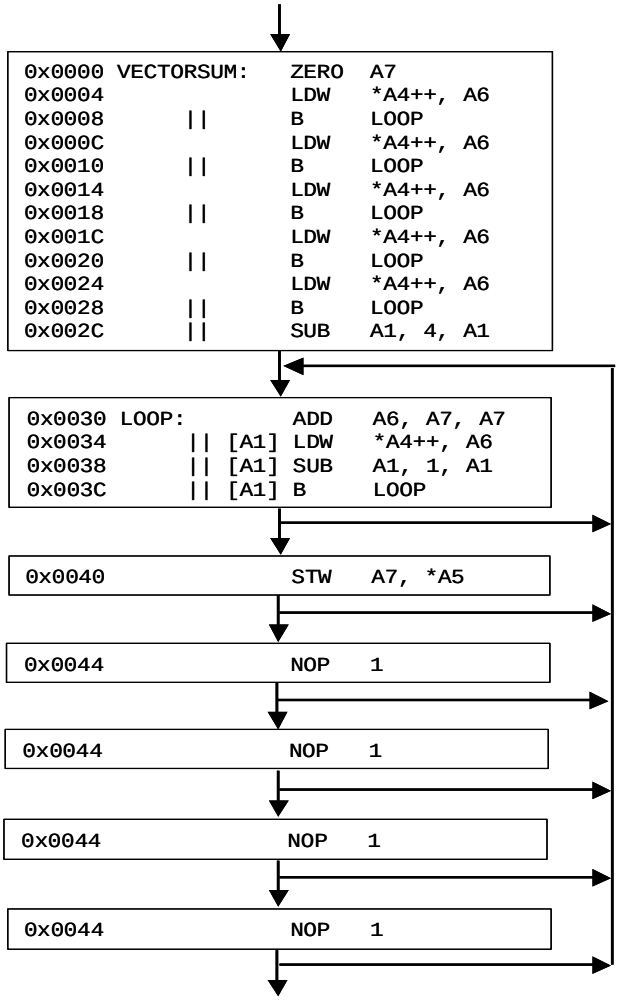


Figure 4.2. Control flow graph for *vectorsum*.

Some architectures may also support dynamic branch targets, in which the destination value may be passed to a register as a function parameter, such as with procedure prologues and epilogues. In these situations, we take an optimistic approach

by assuming the dynamic branch operation is a procedure call. The branch is temporarily treated as a NOP instruction when building the initial CFG to allow the control flow to propagate through. We rely on post-processing steps, such as alias analysis and procedure extraction to determine the possible destinations [43]. The CFG is then regenerated with the newly identified destination values.

Many of today's processor architectures utilize instruction-level parallelism to achieve higher performances, which complicates generation of CFGs. For instance, a branch destination may have a target within a parallel set of instructions. This would break up the control flow at intermediate points within a basic block, creating erroneous data dependencies.

In Figure 4.3, the ADD, SUB, and SRL instructions are scheduled in parallel. However, if the predicated branch is taken, the ADD instruction is not executed. Consequently, the entry label on the SUB instruction partitions the control flow in the middle of the parallel set, placing the latter two instructions in a separate basic block. This forces the A7 operand in the SRL instruction to use the resulting value from the ADD instruction in the previous block. To account for such discrepancies, we introduce a procedure that checks for entry points (labels) within a parallel set of instructions. If such an entry point exists, the instructions falling below the entry point are replicated and added to the top portion of the parallel set. Figure 4.4 shows the MST code after instruction replication. The SUB and SRL instructions have been replicated and a branch operation has been added to jump over the replicated code segment. We rely on

subsequent optimizations in the CDFG, such as code-hoisting [44], to eliminate superfluous operations.

```
0x0800         [A1] B     L1
0x0804              NOP   5
0x0808              ADD   A4, A7, A7
0x080C  L1: ||      SUB   A4, 1, A4
0x0810      ||      SRL   A4, A7, A8
0x0814  L2:         ...
```

Figure 4.3. Branch target inside a parallel instruction set.

```
10.0000  0x0800   [A1] GOTO (6) L1
11.0000  0x0804        NOP  (5) 5
16.0000  0x0808        ADD  (1) $A4, $A7, $A7
16.0100  0x080C        SUB  (1) $A4, 1, $A4    ; replicated SUB
16.0200  0x0810        SRL  (1) $A4, $A7, $A8  ; replicated SRL
16.0300  0x0810        GOTO (0) L2             ; added 'branch-over'
17.0000  0x080C  L1:   SUB  (1) $A4, 1, $A4
17.0100  0x0810        SRL  (1) $A4, $A7, $A8
18.0000  0x0814  L2:   ...
```

Figure 4.4. **MST representation with instruction replication.**

4.3 Linearizing Pipelined Operations

In the previous section, a methodology for generating a CFG from pipelined assembly code was presented. The CFG represents the flow of control in the program via edges connecting basic blocks in the graph. However, the CFG does not inherently contain any information regarding propagation delay or data dependencies in a pipelined control flow. This information must be well defined to generate a proper DFG.

This section describes the linearization process for pipelined operations. The concept of this process is to serialize the pipelined assembly instructions into linear assembly, such that the each pipelined instruction has a well-defined data flow path. The process for linearizing computational operations (arithmetic, logical, memory, etc.) and branch operations are described independently, as they function differently in pipeline architectures. The linearization process assumes that the CFG is complete, i.e., no edges will be inserted between blocks in the future. Consequently, if new edges are added in the future, data propagation and data dependencies are not guaranteed to be correct. To ensure its completeness, we force the algorithm to cover all possible control paths when generating the CFG. This is accomplished in a preprocessing pass that ensures all branch instructions in the program are predicated. A constant predicate of '1', whose condition always resolves to *true*, is added to all non-predicated branch instructions. This forces the branch to be treated as a conditional, and allows the control flow to propagate to the fall-through block. Subsequent optimizations, such as dead-code elimination [44], will remove any resulting extraneous operations.

4.3.1 Event-Triggered Operations

When translating pipelined or scheduled assembly code from one architecture to another, it is essential that the compiler capture the propagation delay and data dependencies correctly. Failure to do so may result in false data dependencies, incorrect data value propagation, and possibly an ill-terminated or non-terminating program.

Referring back to the *vectorsum* procedure in Figure 4.1, we find that the main loop body will execute an unknown number of times until the predicate condition on the branch instruction is *false*, namely, when *A1 = 0*. At that point, the loop will continue to iterate for 5 more cycles until the branches within the pipeline have completed. During this time, data is still computed and propagated through the loop. Should the compiler not consider the propagation delay on the branch instructions, it may terminate the loop early, producing erroneous data. Similarly, failure to consider the propagation delay in the pipelined load instructions will also result in erroneous data.

As a solution, we introduce the concept of an *event-triggered* operation, composed of a *trigger* and an *execute* stage. An event *trigger* is analogous to the read stage in a pipelined architecture, where the instruction is fetched and register values are read; an event *execute* is analogous to the write-back stage in the pipeline, during which the values are written to the destination register or memory. The event triggering and execution stages are offset by the delay of the operation.

An operation event is encapsulated in the MST language using a virtual shift register with a precision *d*, corresponding to the number of delay cycles for the operation. Virtual registers are temporary operands created by the compiler that do not exist within the framework of the source architecture's physical registers. In practice, this results in the addition of a very small shift register since most ISAs generally have no more than 4-6 delay slots in any given multi-cycle instruction. When a pipelined instruction is encountered during the normal flow of the program, an event is triggered

by assigning a ‘1’ to the highest bit (*d-1*) in the shift register. In each successive cycle, a shift-right logical operation is performed on the register. The event is executed after *d* cycles, when a ‘1’ appears in the zero bit of the shift register.

Figure 4.5 illustrates the event triggering for the branch operation in the loop body of the *vectorsum* procedure, which has an operation delay of 6 cycles. In the first iteration of the loop, an event is triggered when the branch instruction is encountered by setting the high bit of shift register. In each subsequent cycle, the register is shifted right while a new event is triggered. After six iterations, event 1 is executed and the branch to LOOP is taken. This is followed by subsequent event executions through the tenth iteration of the loop, until the pipeline in the shift register has been cleared.

The technique described here is utilized in the linearization process for pipelined operations as discussed in the following sections.

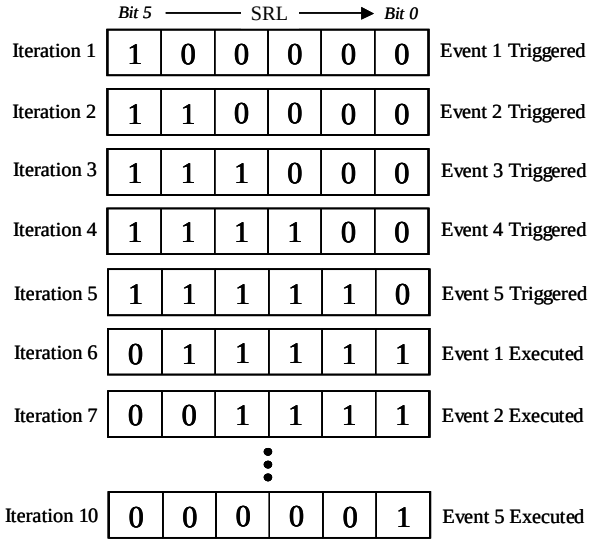


Figure 4.5. Event-triggering for a pipelined branch operation in a loop body.

4.3.2 Linearizing Computational Operations

In the linearization process for computational operations, multi-cycle instructions are serialized into a well-defined data flow path along the pipeline. In order to accomplish this task, virtual registers are introduced to break multi-cycle instructions into a sequence of multiple *single-cycle* instructions. Each instruction in the sequence is guarded by a predicate on an event-triggering register, as described above. Should the program encounter the instruction through a path outside the normal pipeline data flow path, the predicate will prevent the operation from executing.

The linearization process works as follows: For an instruction with *n* delay slots, the original instruction is modified to write to a temporary virtual register R_n, and the delay of the instruction is changed to a single cycle. In each of the subsequent *n-1* cycles, the value is propagated through virtual registers along the pipelined data flow path by assigning $R_{n-1} \leftarrow R_n$, $R_{n-2} \leftarrow R_{n-1}$, …, $R_0 \leftarrow R_1$ in sequence, where R_0 is the original register name. Each of these instructions is predicated on its respective cycle bit of the shift register: *P[n-1]* through *P[0]*. If the end of a basic block is reached, the linearization is propagated to the successor blocks. This approach assumes that no two instructions are scheduled such that both have the same destination register and write-back stages in the same cycle. This is a fair assumption, since compilers generally do not produce code resulting in race conditions. If two or more identical instructions have intersecting pipeline paths, redundant instructions may be avoided by tracking the

timesteps to which they have been written. We rely on optimizations, such as copy and constant propagation [44], to remove any extraneous operations.

```
          :                 :
12.000  0x000C            MOVE(0) 1, $P1[4]       ; LD event cycle 1
12.001  0x000C            SRL(1) $P1, 1, $P1
12.002  0x000C  [$P1[4]]  LD(1) *mem($A4), $A6_4
          :                 :
13.000  0x000C            SRL(1) $P1, 1, $P1      ; LD event cycle 2
13.001  0x000C  [$P1[3]]  MOVE(1) $A6_4, $A6_3
          :                 :
14.000  0x000C            SRL(1) $P1, 1, $P1      ; LD event cycle 3
14.001  0x000C  [$P1[2]]  MOVE(1) $A6_3, $A6_2
          :                 :
15.000  0x000C            SRL(1) $P1, 1, $P1      ; LD event cycle 4
15.001  0x000C  [$P1[1]]  MOVE(1) $A6_2, $A6_1
          :                 :
16.000  0x000C  LOOP:     SRL(1) $P1, 1, $P1      ; LD event cycle 5
16.001  0x0014            OR(0) $P1[0], $P2[0], $MP1
16.002  0x001C            OR(0) $MP1, $P3[0], $MP6
16.003  0x0024            OR(0) $MP6, $P4[0], $MP13
16.004  0x0034            OR(0) $MP13, $P5[0], $MP16
16.005  0x000C  [$MP16]   MOVE(1) $A6_1, $A6    ; intersecting paths
          :                 :                   ; for LDs 1,2,3,4,5
```

Figure 4.6. **Linearization of a pipelined load instruction in the *vectorsum* procedure.**

Figure 4.6 illustrates the linearization process in the MST for the first pipelined load (LD) instruction in the *vectorsum* example of Figure 4.1. In timestep 12, an event is triggered for the LD instruction by posting a ‘1’ to the high bit in the virtual shift register *P1*. Additionally, the LD instruction is modified to write to virtual register *A6_4*, and the operation delay is changed from 5 cycles to 1 cycle. In the subsequent cycles, *A6_4* is written to *A6_3*, *A6_3* is written to *A6_2*, and *A6_2* is written to *A6_1*, at which point the

linearization is propagated to the *LOOP* block. *A6_1* is finally written to the physical register *A6* in timestep 16. Each of these move instructions is guarded by a predicate on a *P1* bit, which is right-shifted in each cycle along the same control path. The same methodology is applied to each LD instruction in program. Although the propagation instructions may read and write to the same register in parallel, the one-cycle delay on each instruction enforces the correct data dependencies.

It is interesting to note that the pipelined LD instructions have intersecting paths. As an example, all five LD instructions will have their 5^{th} cycles intersect in the same timestep (16), where *A6* ← *A6_1*. To avoid extraneous instructions, the propagation instructions are merged by OR-ing their predicates, as shown in the figure.

4.3.3 Linearizing Branch Operations

Unlike computational instructions, branch instructions do not propagate data. Rather, they trigger a change in control flow after a certain number of delay cycles. In linearizing branch operations, only the *event* is propagated through the CFG, as described above. At each branch execution point in the CFG, which can only be the end of a basic block, a copy of the branch instruction is inserted. The branch instruction is predicated on the event shift-register. Similar to the process above, if two or more of the same branch instruction have intersecting paths, redundant instructions may be eliminated by tracking the timesteps to which the instructions have been written. Two or more of the same branch instruction that execute at the same point can be merged by

OR-ing their predicates. The original branch instructions are replaced with NOP instructions in order to maintain the correct instruction flow. Figure 4.7 illustrates the linearization process for pipelined branch operations.

```
           :            :
11.000  0x0008         MOVE(0) 1, $P1[5]    ; branch event cycle 1
11.001  0x0008         SRL(1)  $P1, 1, $P1
11.002  0x0008         NOP(1)  1            ; branch replaced with
NOP
           :            :
12.000  0x0008         SRL(1) $P1, 1, $P1   ; branch event cycle 2
           :            :
13.000  0x0008         SRL(1) $P1, 1, $P1   ; branch event cycle 3
           :            :
14.000  0x0008         SRL(1) $P1, 1, $P1   ; branch event cycle 4
           :            :
15.000  0x0008         SRL(1) $P1, 1, $P1   ; branch event cycle 5
           :            :
16.000  0x0008  LOOP:  SRL(1) $P1, 1, $P1   ; branch event cycle 6
16.008  0x0008         OR(0) $P1[0], $P2[0], $MP0
16.009  0x0010         OR(0) $MP0, $P3[0], $MP1
16.010  0x0018         OR(0) $MP1, $P4[0], $MP2
16.011  0x0020         OR(0) $MP2, $P5[0], $MP3
16.012  0x0028         OR(0) $MP3, $P6[0], $MP4
16.013  0x003C  [$MP4] GOTO(0)  LOOP        ; intersection paths
           :            :                   ; for branches 1-6
```

Figure 4.7. Linearization of a pipelined branch instruction in *vectorsum*.

4.3.4 The Linearization Algorithm

Figure 4.8 presents our algorithm for linearizing pipelined operations. The procedure has the same general organization as the algorithm presented by Cooper et al. [14] for generating CFGs. The algorithm initially creates a *worklist* of *instruction counters* for each basic block in the CFG in lines 1-3, and then iterates through the

worklist in lines 4-26. An instruction counter is particular to a block, and holds a list of pending instructions and a counter representing the remaining clock cycles before each instruction is executed. To prevent redundant iterations over blocks, in lines 8-9, the algorithm checks that the block has not seen any of the pending instruction counters before continuing. The algorithm then iterates over the block by *whole timesteps* in lines 10-20. The instructions in each timestep are iterated through in lines 11-17, as the algorithm searches in line 12 for previously unvisited pipelined instructions to add to the instruction counter. Lines 13-15 add a counter for the branch instructions with cycle delays greater than zero; the original branch instruction is replaced with a NOP instruction to maintain the correct program flow. Lines 16-17 add counters for all multi-cycle instructions whose write-back time falls outside the block. Unique *event* instructions are inserted for each pending instruction in lines 18-20, as described above; those that have completed are removed from the instruction counter list. After iterating over the instructions within each timestep, the pending instruction counters are decremented in line 21. At the conclusion of the iteration over timesteps in the block, lines 22-26 propagate all pending counters to new instruction counters for each successor block edge; the new instruction counters are added to the worklist. The algorithm terminates once no new instruction counters are encountered by any block and the worklist is empty. The algorithm runs in $O(n)$ time, where n is the number of instructions in the program, assuming a small, constant number of outgoing edges between blocks.

```
  Linearize_Pipelined_Operations( CFG )
 1  worklist = empty list of InstrCounters
 2  for each basic block in CFG do
 3    add InstrCounter(block) to worklist
 4  while worklist->size() > 0 do
 5    instr_counter = worklist->front()
 6    remove instr_counter from worklist
 7    block = instr_counter->block
 8    if block has seen all live counters in instr_counter then
 9      continue
10    for each whole timestep ts in block do
11      for each instruction i in timestep ts do
12        if i has not been seen by instr_counter then
13          if i is a branch instruction and i->delay > 0 then
14            add {i:i->delay} to instr_counter
15            replace branch instruction i with NOP instruction
16          else if (i->timestep + i->delay) > block->max_time
17            add {i:i->delay} to instr_counter
18        for each counter c in instr_counter do
19          insert a unique event instruction for c in timestep ts
20          if c = 0 then remove c from instr_counter
21      instr_counter->DecrementCounters()
22    if instr_counter has live counters
23      for each successor s of block do
24        target_instr_counter = InstrCounter(s)
25        add unique live counters to target_instr_counter
26        add target_instr_counter to worklist
```

Figure 4.8. Linearization algorithm for pipelined operations.

4.4 Generating the Control and Data Flow Graph

In the previous sections we described how to build a CFG and break data dependencies in pipelined and scheduled assembly code. In this section we combine the two techniques to generate the complete CDFG. The procedure is described in Figure 4.9, which takes a list of assembly instructions as input and returns a CDFG. The procedure begins with a preprocessing step to ensure that all branch instructions in the

program are predicated as described in the previous section. The algorithm constructs the CFG using Cooper's algorithm, and then linearizes the pipelined operations as described above. The data flow graph is then generated from the newly serialized instructions. Source edges are connected based on the timing dependencies in the MST as described in Section 3.1.3.

```
  Generate_CDFG( instr_list )
1  Predicate_Pipelined_Instrs( instr_list )
2  CFG = Generate_Ctrl_Flow_Graph( instr_list )
3  Linearize_Pipelined_Operations( CFG )
4  CDFG = Generate_Data_Flow_Graph( CFG )
5  return CDFG
```

Figure 4.9. Procedure for generating a CDFG.

4.5 Experimental Results

The correctness of the methodology presented in this chapter was verified on 8 highly pipelined benchmarks in the Texas Instruments C6000 DSP assembly language. The FREEDOM compiler generated CDFGs and RTL code targeting the Xilinx Virtex II FPGA. Each benchmark was simulated using Mentor Graphic's ModelSim to verify bit-true accuracy and obtain cycle counts.

There has been little work reported on translating highly pipelined software binaries to RTL code for FPGAs. This makes comparison with other approaches difficult. However, it is interesting to consider the impact and effectiveness of this

algorithm in a high-level synthesis tool. Table 4.1 shows comparisons in cycle counts for the TI C6000 DSP and the Virtex II FPGA, generated by the FREEDOM compiler. Also shown is the number of pipelined operations in each benchmark and the number of instructions inserted during the linearization process to demonstrate the impact on code size when using this approach.

Results indicate the FREEDOM compiler successfully generated the correct CDFGs from the pipelined assembly code, allowing complex optimizations and scheduling to significantly reduce clock cycles in the FPGA design. On average, approximately 9 instructions were added for each pipelined operation and there was a 27% increase in code size during the linearization process. *Please note that these values reflect the size of the design before CDFG optimizations, which will further reduce implementation complexity.*

Table 4.1. Experimental results on pipelined benchmarks.

Benchmark	DSP Cycles	FPGA Cycles	# Pipelined Instructions	# Added Instructions
memmove	125747	2516	33	352 (24.7%)
memcpy	69615	2004	14	136 (52.3%)
divi	282301	16127	17	141 (27.3%)
mpyd	1329176	39669	26	269 (14.0%)
remi	260148	16888	13	130 (34.6%)
dsp_fir_gen	30851	685	49	683 (43.1%)
lms_filter	33537580	773288	147	967 (13.7%)
noise_canceller_fir	8239397	163778	21	105 (5.3%)

4.6 Summary

This chapter presented a methodology for correctly representing the data dependencies and data propagation when generating CDFGs from highly pipelined and scheduled assembly code. This process consists of three stages: generating a control flow graph, linearizing the assembly code, and generating the data flow graph. We use a known method for generating the control flow graph from scheduled assembly code and describe further techniques for handling more complex architectures that employ parallel instruction sets and dynamic branching. We present a linearization process, in which pipelined structures are serialized into linear assembly. This allows for proper data dependency analysis when generating the data flow graph.

The work was verified in the FREEDOM compiler on 8 highly pipelined software binaries for the TI C6000 DSP, targeting the Xilinx Virtex II FPGA. Results indicate that data dependencies were correctly identified, enabling the compiler to perform complex optimizations and scheduling to reduce clock cycles in the designs.

Chapter 5

Control and Data Flow Graph Optimizations

This chapter presents the optimizations that were implemented for the FREEDOM compiler. The optimizations shown in Figure 5.1 have been implemented on the CDFG, and are cycled through until there are no more changes. Most of these optimizations are well known algorithms [44]. Many of them necessitate *structural analysis* and *reaching definitions* [44], which are discussed in the following sections.

5.1 CDFG Analysis

The following sections discuss methods of control and data flow analysis that are performed on a CDFG. *Structural analysis* is performed on the control flow graph (CFG), while *reaching definitions* is performed on the data flow graph (DFG).

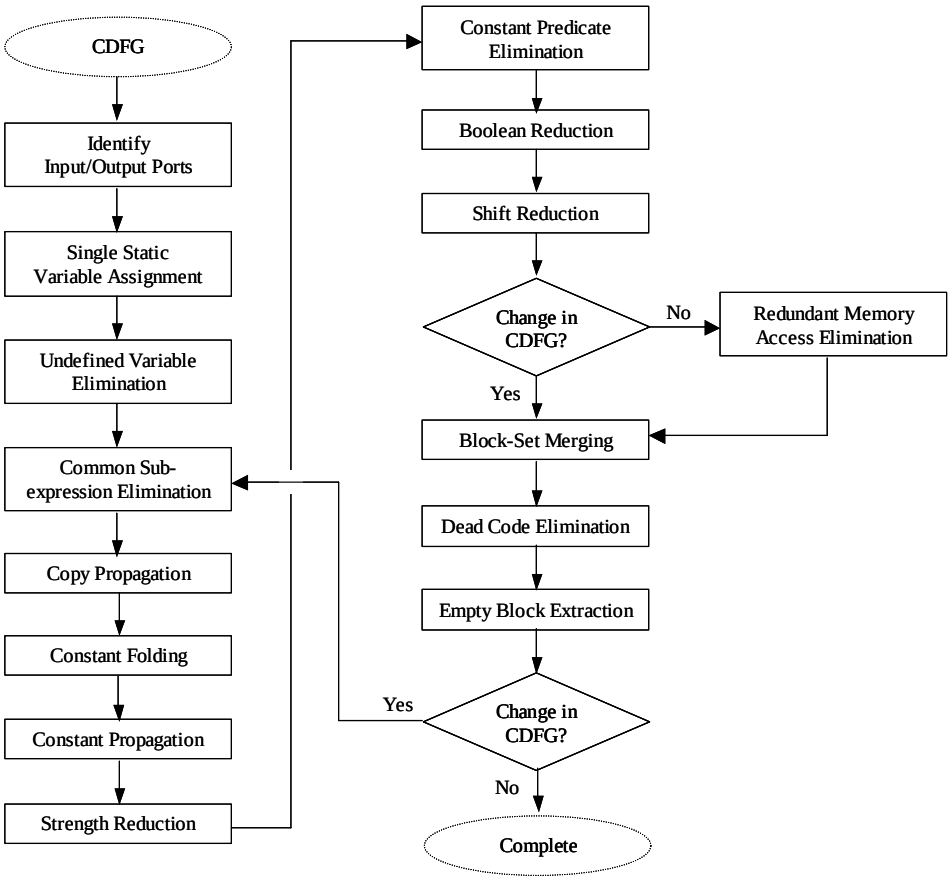


Figure 5.1. CDFG optimization flow for the FREEDOM compiler.

5.1.1 Structural Analysis

Structural analysis is a method of determining high-level constructs within a CFG, such as loop bodies. Structural analysis was implemented in the FREEDOM compiler using the graph minimization technique [44]. The following structures are supported in the FREEDOM compiler: *block-set, if-then, if-then-else, self-loop, while-loop, natural-loop,* and *improper cycles.* Figure 5.2 illustrates the graph minimization

technique performed on a CFG. The first iteration reduces the *if-then-else* construct. The second iteration reduces the *self-loop*. Finally, the third iteration reduces the *block-set*.

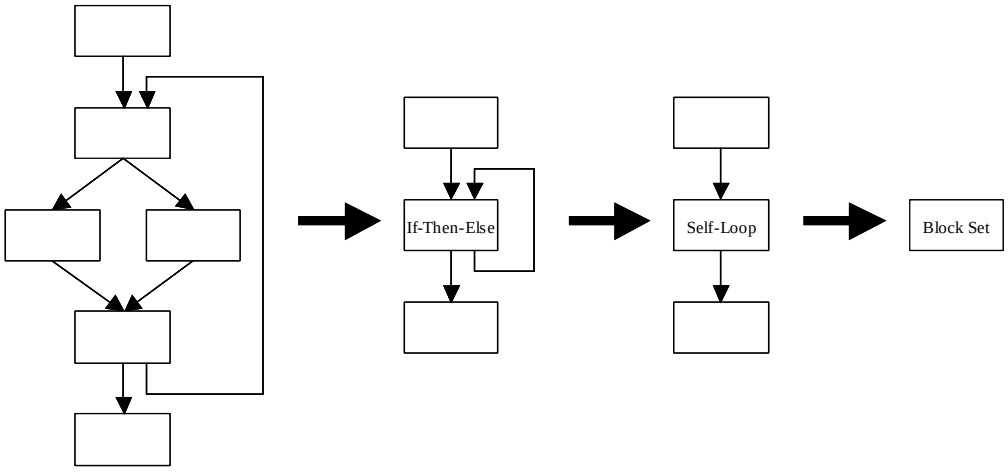


Figure 5.2. Structural analysis on a CFG using graph minimization.

5.1.2 Reaching Definitions

Reaching definitions is a method of data dependency analysis [44]. Four bit vectors are used to represent each node in the CDFG during the analysis. The *PRSV(i)* vector represents the definitions that are preserved by block *i*. The *GEN(i)* vector represents the definitions which are killed in block *i*. Using these two vectors, we may then define the bit vectors *RCHin(i)* and *RCHout(i)*, which represent definitions that reach the beginning and end of block *i*, respectively. A definition is said to reach the beginning of a block *i* if that definition reaches the end of all blocks preceding *i*. A definition is said to reach the end of block *i* if it either occurs in block *i* and the variable

it defines is not redefined in *i*, or it reaches the beginning of *i* and is preserved by *i*. The *RCHout(i)* vector is thus defined as:

$$RCHout(i) = GEN(i) \vee (RCHin(i) \wedge PRSV(i)) \quad for\ all\ i.$$

In order to solve the system of bit vector equations, the equations are iterated until no further changes result. The final system of equations is used to generate *definition-use (DU) chains* and *use-definition (UD) chains*. The former maps a definition to all it uses in the CDFG, while the latter maps a use of a variable to all of its definitions in the CDFG. These mappings are used to determine data dependencies in the CDFG.

5.2 CDFG Optimizations

The following sections discuss the CDFG optimizations that were implemented in the FREEDOM compiler. Most of these optimizations are well known, while others are modified versions or are extensions of the traditional implementation.

5.2.1 Identifying Input and Output Ports

Input and output (I/O) ports are identified immediately after generating a CDFG. When identifying I/O ports for a hardware-software co-design, it is especially critical to limit the cut-size in the graph. In other words, a compiler often generates many intermediate variables, or virtual registers, beyond the scope of the original source

architecture. These variables should never be considered as I/O port candidates. In binary translation, the maximum number of ports, or the maximum cut-size of the graph, would be limited to the number of physical registers in the source processor architecture.

Identifying I/O ports requires the use of DU-chains and UD-chains. There are three types of ports supported in the FREEDOM compiler: *input ports*, *output ports* and *inout ports*. An *input port* is defined as a node that is used, but has no prior definition. Any physical register that is written to is added as an *output port* since the procedure must capture any global changes occurring to the register file. An *inout port* is a variable that has been defined as both an *input* and *output port*. However, we rely on single static-variable assignment (SSA), describe in 5.2.2, to prevent *inout port* types because backend synthesis tools generally do not support bi-directional ports.

Input ports are added to the entry block in the CDFG and output ports are added to the exit block in the CDFG, as illustrated in Figure 5.3. When identifying ports in the CDFG, all *output ports* must also have an equivalent *input port* added, since transformations in the CDFG during optimizations can cause changes in data dependencies to values outside the procedural boundaries. By adding an equivalent input port for each output port we cover all possibilities. During optimizations, unused ports or those whose value does not change between the input and output are eliminated.

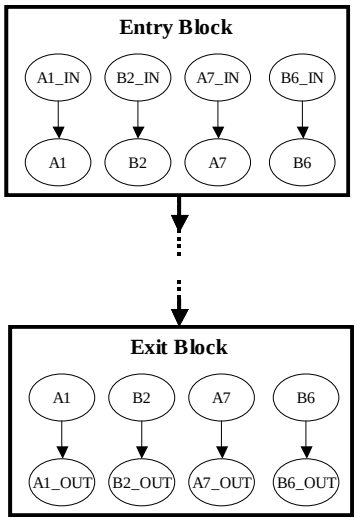


Figure 5.3. Adding input and output ports to a CDFG.

5.2.2 Single Static-Variable Assignment

Single static-variable assignment (SSA) is a method of breaking data dependencies by ensuring that every assignment in the CDFG has a unique variable name [44]. SSA form is invaluable, as it simplifies the dependencies in the DU-chains and UD-chains, allowing for more effective optimizations.

Traditionally, a *Φ-function* is used in SSA to join multiple assignments to a variable, stemming from different paths in the CDFG. The number of arguments to the *Φ-function* is equal to the number of definitions of the variable from each point in the CDFG. This method often causes a significant bottleneck when handling numerous data paths. Interestingly, once the pipelined operations in the CDFG have been linearized as described in 4.3, the *Φ-function* becomes superfluous, as only the latest definition of a variable will reach the end of the block and propagate through the control flow. Those

instructions with multi-cycle delays that originally crossed basic block boundaries have since been serialized into multiple single-cycle instructions. As a result, the latest definition of each SSA variable may be assigned back to its original variable name at the end of the block, thus eliminating the need for the *Φ-function*. Optimizations, such as copy propagation and dead-code elimination [44], will remove extraneous assignment operations created by this process.

5.2.3 Undefined Variable Elimination

Undefined variable elimination initializes variables to zero that are not input ports and are used without a prior definition. Variables are initialized in the entry block of the CDFG. By initializing undefined values to zero, undefined variable elimination allows other optimizations, such as constant folding and constant propagation [44], to more efficiently reduce the CDFG. This is a valid optimization because in hardware an undefined variable would either be automatically set to zero initially or left at high-impedance, producing erroneous results. Here we are ensuring proper value initialization during compile time to reflect the resulting hardware implementation.

5.2.4 Common Sub-Expression Elimination

Common sub-expression elimination removes redundant operations by replacing them with previously saved values [44]. An operation is said to be a common sub-

expression if it is preceded by another instruction containing the same expression, and the operands remain unchanged between the two evaluations. This optimization is implemented locally in each CDFG block by mapping each operation node to a hash key representing the operation as function of its inputs. If a subsequent operation produces the same hash key, it is replaced with an assignment from the previously mapped node.

To ensure proper equivalence checking, the hashing function must capture low-level information such as operation precision, sign, bit-range select, and predicated values. Additionally, the source operands in the hash string are sorted alpha-numerically for operations that hold the commutative and associative properties to ensure expressions are always hashed consistently. As an example, a 16-bit signed addition operation would be hash-defined as *SADD16(A1[15:0], B2[15:0])*, where register *A1* appears before *B2* in the operand ordering.

This optimization produces many assignment operations, and therefore we rely on optimizations such as copy propagation [44] to reduce the CDFG even further. Figure 5.4 shows an example of common sub-expression elimination, where the expression assign to *e* is evaluated and replaced by an assignment from the value *c*.

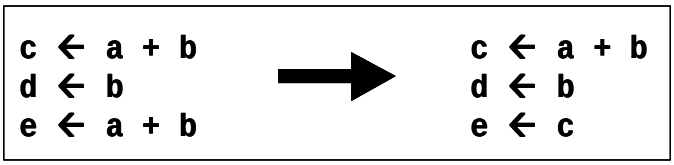


Figure 5.4. Common sub-expression elimination example.

5.2.5 Copy Propagation

Copy propagation is a transformation that given an assignment of variables $x \leftarrow y$, replaces subsequent uses of x with y, as long as the intervening instructions have not changed the value of either x or y [44]. However, if x is a predicated assignment operation, one may not propagate the value y to any later uses of x because the result is said to be indeterminate, unless the succeeding operations are predicated on the same conditional expression. Figure 5.5 illustrates examples of copy propagation, where subsequent uses of *a* and *d* are replaced by *b* and *c*, respectively.

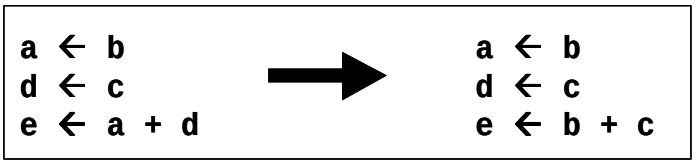


Figure 5.5. Copy propagation examples.

5.2.6 Constant Folding

Constant folding is an optimization that solves for operations on constant values at compile time [44]. The result is a reduction in extraneous computations and resources, and allows constant propagation to further reduce the CDFG. Figure 5.6 illustrates examples of constant folding, where each operation is solved at compile time.

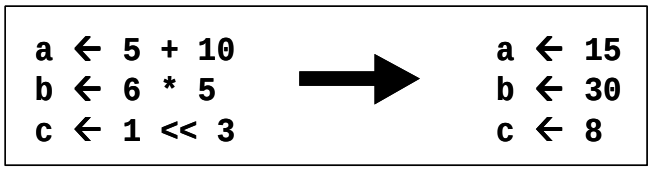


Figure 5.6. Constant folding example.

5.2.7 Constant Propagation

Constant propagation is similar to copy propagation, in that given an assignment $x \leftarrow c$ for a variable x and a constant c, the optimization replaces later uses of x with c, as long as the intervening instructions have not changed the value of x [44]. Similarly, if x is a predicated assignment operation, one may not propagate the value c to any later uses of x because the result is said to be indeterminate, unless the succeeding operations are predicated on the same conditional expression. Constant propagation is performed across blocks, and therefore requires reaching definitions to determine data dependencies. This optimization allows the opportunity for others, such as constant folding and strength reduction, to further reduce the CDFG as illustrated in Figure 5.7.

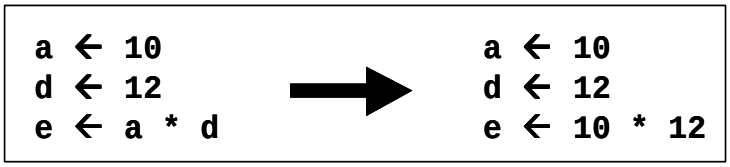


Figure 5.7. Constant propagation example.

5.2.8 Strength Reduction

Strength reduction replaces operations that are costly in terms of area, power, or cycle time with less expensive ones [44]. For instance, multiplication by a factor of two may be replaced by a shift operation, and an addition or subtraction operation with a zero operand can be eliminated. Constant folding is inherently an aspect of strength reduction since solving constant expressions at compile time also reduces the cost of resources. Figure 5.8 illustrates examples of strength reduction.

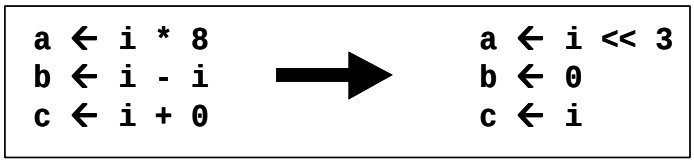


Figure 5.8. Strength reduction example.

5.2.9 Constant Predicate Elimination

Predicated instructions are problematic when optimizing a CDFG because the result of a predicated operation is deemed indeterminate, even for constant expressions. Specifically, predicates prevent copy and constant propagation of values, unless the succeeding operations share the same predicate. Removing predicates is essential in optimizing a CDFG.

Constant predicate elimination solves for predicated operations whose predicate operand is evaluated to a constant expression at compile time. If the predicate condition is met, the predicate is removed from the operation node. Otherwise, the operation is

replaced with an assignment from its previous definition. The result of this optimization leads to a reduction in the number of multiplexers implemented in a hardware design, saving the cost of area and critical path. Additionally, it allows other optimizations to function more efficiently in reducing the CDFG. In the example in Figure 5.9, the first predicated operation is evaluated to be true, so the predicate is removed. However, the second predicated operation is evaluated to be false, and therefore we propagate the previous definition of *c* to itself.

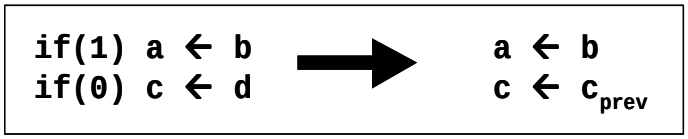


Figure 5.9. Constant predicate elimination example.

5.2.10 Boolean Reduction

Boolean reduction is an optimization that reduces sequences of conditional operations or predicates. The result of this optimization leads to a reduction in the number of multiplexers implemented in a hardware design, saving the cost of area and critical path. In Figure 5.10, *d* is reduced to the inverse Boolean expression of *a*.

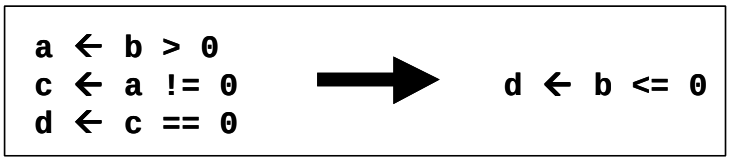


Figure 5.10. Boolean reduction example.

5.2.11 Shift Reduction

Shift Reduction reduces sequences of shift operations by summing their shift values into a single operand, thus reducing the number of barrel shifters in the design. The CDFG can be further reduced if these shift operations include constant shift values, which may be evaluated at compile time. This is illustrated in Figure 5.11.

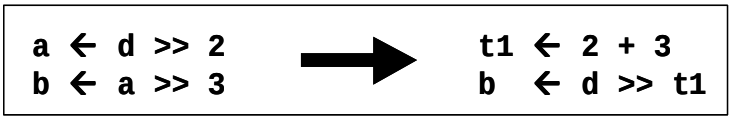


Figure 5.11. Shift reduction example.

5.2.12 Redundant Memory Access Elimination

Redundant memory access elimination is an expansion of common sub-expression elimination, in which the string hashing method is applied to memory addresses to determine if two memory operations intersect in order to remove redundant memory operations. This optimization is essential in removing the effects of memory spilling and redundant memory operations that often occur after unrolling a loop.

In this optimization it is imperative that address expressions are accurately compared. Any ambiguities between the hash expressions can cause errors in the evaluation. Therefore, we apply several rules in hashing address expressions for memory operations. Firstly, address expressions are evaluated to the top of the CDFG, not just a single level as in common sub-expressions. This ensures the memory addresses are

represented as accurately as possible. Secondly, this optimization is only implemented once all other optimizations have minimized the CDFG as much as possible and has since become stable, as illustrated in Figure 5.1. This ensures there are no unnecessary copy assignments or other ambiguities in the CDFG prior to hashing the memory address expression. Finally, this optimization does not cross the boundary of any function call with memory dependencies or memory operations whose address is calculated by means of a memory read operation. The reason for these two restrictions is that it is not possible to definitively assert whether or not memory accesses intersect across these boundaries, and therefore the memory values are said to be indeterminate.

A memory operation is said to be redundant if it is preceded by another memory operation containing the same address expression, and the value in memory remains unchanged between the two evaluations. In such a case, the following rules may be applied: If two consecutive memory read operations access the same address, the second memory read operation is eliminated and the result of the first operation is forwarded. This is called a *read-after-read reduction*. If two consecutive memory write operations access the same address, the first memory write operation is eliminated since the second over-writes the first. This is called a *write-after-write reduction*. If a memory read operation follows a memory write operation in which both access the same address, the memory read operation is eliminated and the value written to memory is forwarded. This is called a *read-after-write reduction*. Predicated memory operations cannot be reduced,

since the values read or written to memory are said to be indeterminate, unless the subsequent memory operations are predicated on the same conditional expression.

Figure 5.12 illustrates examples of these memory reduction techniques. In the first set, *b* is forwarded to *c* in *read-after-read reduction*. In the second set, the first memory write operation is eliminated due to a *write-after-write reduction*. In the final set, the read operation is eliminated and *h* is forwarded to *i* in a *read-after-write reduction*.

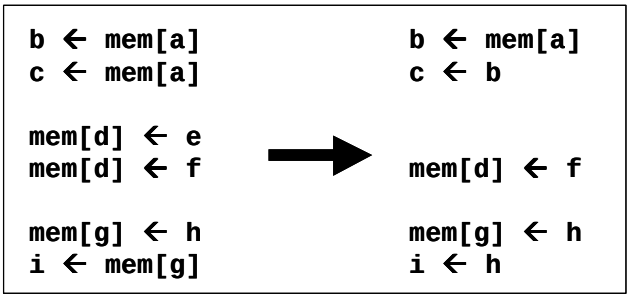


Figure 5.12. Redundant memory access elimination examples.

5.2.13 Block-Set Merging

It is often the case that a CDFG may contain sets of consecutive blocks, in which the first block is the only predecessor to the second block, and the second block is the only successor to the first block. *Block-set merging* is an optimization that merges these consecutive blocks and the nodes within, allowing for more effective optimizations in reducing the design complexity and more parallelism when applying scheduling techniques. After the nodes in the blocks have been merged, it is necessary to run SSA

on the merged block to break data dependencies once again. The block-set merging technique is illustrated in Figure 5.13.

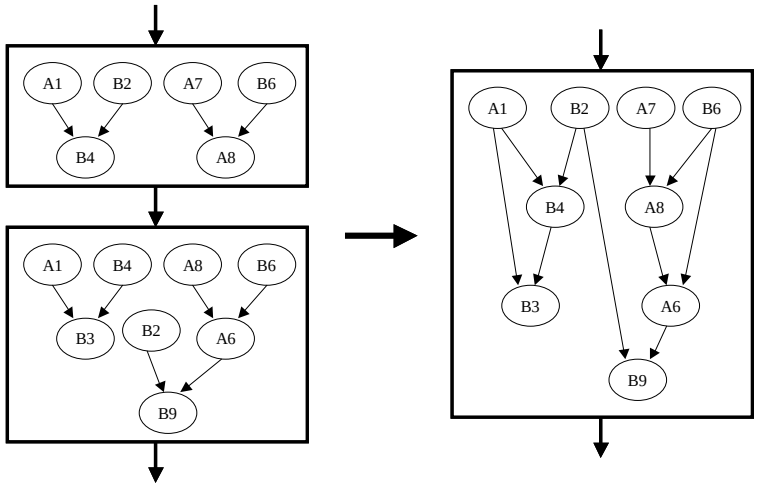


Figure 5.13. Block-set merging example.

5.2.14 Dead-Code Elimination

Dead-code elimination [44] removes extraneous operations that have no effect on the rest of the design. This is accomplished by utilizing DU-chains to find operations that have no subsequent use and are not output ports. The result is a minimization or resources and area in the design.

5.2.15 Empty Block Extraction

Dead-code elimination may end up removing all nodes within a block, leaving an empty block within the control flow graph. Although, inherently there is no ill effect, the

empty block do not serve a purpose, and may inadvertently prevent others from merging or possibly add extraneous control states. *Empty Block Extraction* removes these blocks from the CFG and modifies the edges to reflect the new control flow.

5.2.16 Loop Unrolling

Loop unrolling is a method of decreasing clock cycles in a design by unrolling the body of a loop in order to allow more instructions to be scheduled in parallel. This optimization requires structural analysis to be performed initially in order to identify loop constructs in the design. Only certain loop structures may be unrolled, such as *self-loops* and *while-loops*. Loop unrolling is implemented in the FREEDOM compiler by inserting successive copies of the blocks (and their instructions) within a loop body. We rely on block-set merging to join the consecutive copies of the blocks in the loop body. Figure 5.14 shows a self-loop structure in the CDFG (left), which is unrolled four times (center). The consecutive block sets and their nodes are then merged together (right).

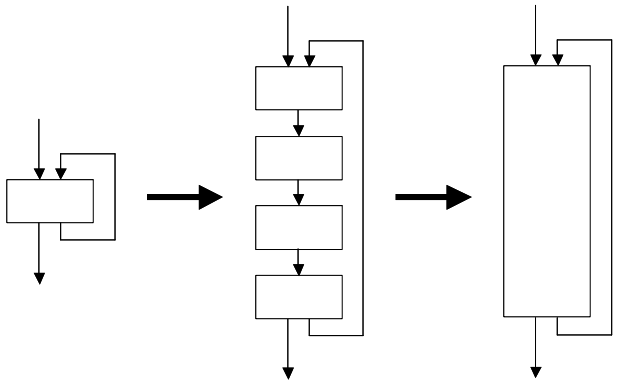


Figure 5.14. Loop unrolling for a self-loop structure in a CDFG.

5.2.17 Register Allocation

Register allocation is an optimization that is performed after scheduling to reduce the number of registers. Reducing registers in circuit designs generally leads to smaller design size. Unlike general-purpose processor architectures, FPGAs are not limited to a small, fixed number of registers. However, the scheduling of operations affect the number of possible register reuses.

Register allocation was implemented in the FREEDOM compiler using the Linear-Scan (left-edge) algorithm [53]. We assume the nodes in the CDFG are in SSA form, in which data dependencies are broken. We also assume an unbound number of register resources in the target FPGA, and our task is to assign the variable lifetimes to the smallest subset of registers. Structural analysis is performed beforehand to identify loops and other constructs.

Prior to running the Linear-Scan algorithm, one must determine the *liveness* of each variable, or the time from the variable's first definition until its last use. This information is obtained from the nodes in the CDFG after scheduling. Our approach for calculating the live intervals of registers in a CDFG is as follows: The nodes in each basic block are sorted in depth-first order. The nodes in each block are then mapped to a register table containing the name of the node and its minimum/maximum time interval; each node that is encountered is added to the table under its corresponding name. The lifetime interval is updated by comparing the current start time with the node's timestamp, and comparing the current end time with the timestamp of all successive uses

of the node. If a *Variable* node exists inside a loop, the time interval for that register name is extended to the loop body's time boundaries. The register table is used in the Linear-Scan algorithm by renaming the list of nodes in each mapping with a newly allocated register name, based on the time intervals. We use a simple approach that does not make use of lifetime holes or necessitate memory spilling. The algorithm runs in *O(n)* time, where *n* is the total number of nodes in the CDFG.

5.3 Experimental Results

This section reports the results of the FREEDOM compiler optimizations on a set of ten benchmarks. The benchmarks were written in C and compile to the TI C6211 DSP architecture using TI's Code Composer Studio. The FREEDOM compiler was then used to translated the software binaries to hardware descriptions, targeting the Xilinx Virtex II XC2V250 FPGA. Table 5.1, Table 5.2, and Table 5.3 present the results in clock cycles, frequency, and area for the Xilinx Virtex II FPGA, respectively. The benchmarks were run with different combinations of optimizations to show their individual effects.

The first column shows the results of the compiler's base case (B) with a simple scheduling routine and no optimizations. The use of traditional optimizations facilitates a reduction in the complexity and size of the CDFG. The results are apparent when comparing the base case (B) and the optimizations alone (B+O), as the clock cycles and

area dramatically decrease. In many cases, the optimizations also allowed for better frequency results due to less complex structures in the resulting hardware.

When combining traditional optimizations with loop unrolling (B+O+U), results show improvement in clock cycles due to greater levels of parallelism in the design. Area increased as expected, while frequency results dropped due to larger complex structures in the resulting design. The *ellip* benchmark did not have loops to unroll and therefore showed no changes in results.

The addition of memory optimizations (B+O+U+M) enables us to eliminate the redundant memory operations that occur after unrolling loop bodies, allowing for greater parallelism in the design. This is apparent in the dramatic decrease in the clock cycles for most of the benchmarks. The memory optimizations in turned allowed other optimizations to reduce the design further, as apparent in the reduced area and higher frequency results in most benchmarks. The *sobel* benchmark did not produce any changes in results because its memory operations are scattered among different *if-then-else* structures in the design. Consequently, memory forwarding was not affective in this benchmark.

The final column presents results with register allocation (B+O+U+M+R). It is interesting to note that this optimization in fact caused a negative effect in almost all benchmarks; results showed design areas increased while frequencies decreased. Seemingly, the effect of register reuse requires the insertion of additional multiplexers by the backend synthesis tools in order to support multiple uses of the register across

different combinational logic blocks. This in fact causes the design area, interconnect and the critical path to increase. Accordingly, results show that the optimal method for implementing FPGA designs would be to not use register reuse optimizations, and leave the design in SSA form.

Table 5.1. Clock cycle results for CDFG optimizations.

	B		B+O		B+O+U		B+O+U+M		B+O+U+M+R	
Benchmark	Cycles	%	Cycles	%	Cycles	%	Cycles	%	Cycles	%
dot_prod	11010	100.0	7004	63.6	5204	47.3	1654	15.0	1654	15.0
Iir	14487	100.0	8988	62.0	7656	52.8	6306	43.5	6306	43.5
Matmul_32	1852963	100.0	1139471	61.5	877327	47.3	179944	9.7	179944	9.7
Gcd	469	100.0	200	42.6	173	36.9	167	35.6	167	35.6
Diffeq	997	100.0	725	72.7	668	67.0	158	15.8	158	15.8
Ellip	199	100.0	189	95.0	189	95.0	66	33.2	66	33.2
laplace	71739	100.0	47599	66.4	40219	56.1	9431	13.1	9431	13.1
fir16tap	207056	100.0	119071	57.5	94971	45.9	25073	12.1	25073	12.1
fir_cmplx	17898	100.0	10516	58.8	5364	30.0	4084	22.8	4084	22.8
Sobel	97606	100.0	49493	50.7	44543	45.6	44543	45.6	44543	45.6

Table 5.2. Frequency results in MHz for CDFG optimizations.

	B		B+O		B+O+U		B+O+U+M		B+O+U+M+R	
Benchmark	Freq	%	Freq	%	Freq	%	Freq	%	Freq	%
dot_prod	107.4	100.0	132.8	123.6	120.4	112.1	132.8	123.6	99.5	92.6
Iir	122.0	100.0	131.4	107.7	107.4	88.0	112.2	92.0	89.5	73.4
matmul_32	106.0	100.0	131.7	124.2	92.0	86.8	127.2	120.0	85.0	80.2
Gcd	103.9	100.0	127.8	123.0	117.7	113.3	111.3	107.1	113.8	109.5
Diffeq	106.3	100.0	112.2	105.6	106.5	100.2	122.1	114.9	96.1	90.4
Ellip	114.4	100.0	108.0	94.4	108.0	94.4	129.6	113.3	123.0	107.5
Laplace	101.4	100.0	91.0	89.7	86.2	85.0	108.0	106.5	89.0	87.8
fir16tap	75.8	100.0	118.9	156.9	109.8	144.9	105.7	139.4	92.5	122.0
fir_cmplx	91.2	100.0	130.1	142.7	104.1	114.1	126.7	138.9	82.2	90.1
Sobel	77.5	100.0	131.6	169.8	92.0	118.7	92.0	118.7	69.3	89.4

Table 5.3. Area results in LUTs for CDFG optimizations.

	B		B+O		B+O+U		B+O+U+M		B+O+U+M+R	
Benchmark	Area	%	Area	%	Area	%	Area	%	Area	%
dot_prod	566	100.0	223	39.4	1761	311.1	1463	258.5	4112	726.5
Iir	1402	100.0	478	34.1	3183	227.0	3064	218.5	10608	756.6
matmul_32	2447	100.0	1290	52.7	3269	133.6	2231	91.2	6311	257.9
Gcd	1067	100.0	566	53.0	799	74.9	812	76.1	1062	99.5
Diffeq	1693	100.0	1096	64.7	2451	144.8	1639	96.8	4972	293.7
Ellip	2995	100.0	2277	76.0	2277	76.0	1815	60.6	2430	81.1
Laplace	2461	100.0	1544	62.7	5617	228.2	3784	153.8	7401	300.7
fir16tap	1661	100.0	957	57.6	2702	162.7	1789	107.7	5594	336.8
fir_cmplx	1674	100.0	998	59.6	3589	214.4	3563	212.8	12738	760.9
Sobel	2734	100.0	1443	52.8	9188	336.1	9188	336.1	20525	750.7

5.4 Summary

This chapter presented the CDFG optimizations that were implemented for the FREEDOM compiler. Most of the optimizations are well known algorithms, while others were custom optimizations design for this compiler. The objective of these optimizations is to reduce the size and complexity of the CDFG. Results on ten benchmarks show significant improvements in area and timing performance using the minimization techniques and transformations outlined in this chapter. However, *register allocation* seems to have a negative effect on the area for FPGA designs, which is contrary to its objective of reducing registers in the design. Consequently, it is preferable to leave FPGA designs in SSA form to obtain better area and frequency performance.

Chapter 6

Scheduling

High-level synthesis tools often allow the designer to perform tradeoffs between optimizations, usually using estimations. At high levels of abstraction, these estimates are prone to errors, leading to inaccurate predictions of the final implementation after several levels of transformations. Consequently, poor or inaccurate decisions at high levels in the synthesis process will have direct impact on all future optimizations and the resulting hardware implementation.

Accurate estimations play a particularly important role in scheduling of operations for hardware implementation on FPGA. The operation nodes in the CDFG are scheduled by assigning each operation to an RTL state in a finite state machine using schemes such as As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) scheduling [16]. When resources usage is critical, other schemes such as list scheduling and force-directed scheduling are implemented. When targeting FPGAs, infinite resources are often assumed; the objective of scheduling is to minimize the number of

clock cycles (states), while maximizing the parallelism in the design as much as possible, even at the cost of area. While ASAP and ALAP scheduling are quite efficient for this task, they generally hamper instruction-level parallelism by producing a non-uniform distribution of operations among the state cycles. Consequently, this results in an uneven distribution of resource usage, latency and power.

Operation chaining is a technique that is effective in reducing cycles in a design by allowing the result of an operation to be used immediately rather than in the next cycle. It is expected that scheduling without operation chaining will produce the best overall frequency at the cost of cycles, since the critical path is limited to the operation in the design with the largest delay. Conversely, a naïve approach to operation chaining may often result in large critical paths, low frequencies and suboptimal performance. Optimally, we would like to find a chaining technique that optimizes both frequency and clock cycles simultaneously.

In previous work, we evaluated simple scheduling and chaining routines in the context of translating software binaries to FPGA implementations [76]. In this chapter we presents a balanced scheduling routine, in which operations are uniformly distributed across states. A balanced chaining routine is also presented, which reduces the clock cycles and critical path of the design given a target frequency. In order to accurately determine the best-fit operation chaining, we require a delay model that considers varying bit-width for each operation implemented in the target FPGA architecture. The methodology for obtaining such models is also presented.

6.1 Related Work

Numerous algorithms for scheduling have been developed over the years by various researchers. For a given data flow graph, scheduling determines the concurrency of the resulting implementation by assigning operations in a CDFG to specific cycles, assuming either unconstrained or constrained resources. In this paper we study the use of scheduling in the context of unconstrained resources. As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) scheduling are often used for this purpose.

Force-directed scheduling was introduced by Paulin and Knight [50] as a way of minimizing resources under timing constraints. The algorithm uses the ASAP and ALAP times to determine the time frame for each operation, whereby a force is computed as a distribution function to determine the best schedule for the operation. The worst-case time complexity for the algorithm in cubic with the number of operations. Efficiency improvements were shown by Verhaegh et al. [65] through incremental force calculations that reduce the complexity to quadratic with the number of operations. Paulin and Knight [50][51] have also shown how force-directed scheduling can be integrated with list-scheduling, where the force calculations are used as the priority function. In contrast to force-directed scheduling, the balanced scheduling approach presented here only considers the instruction-level parallelism in the design, where operations are not distinguished by their type. In other words, it attempts to uniformly distribute the number of operations per cycle within the ASAP/ALAP time

frame, whereas force-directed scheduling attempts to balance similar operation types among states.

Kerns and Eggers [35] introduced a balanced scheduling algorithm that schedules instructions based on an estimate of the amount of load-level parallelism in the program. The scheduler computes load instruction weights based on a measure of the number of instructions that may execute in parallel with each load instruction. The instructions are then spread out to cover the load latency. Our balanced scheduling algorithm is different in that it considers the instruction-level parallelism for all single and multi-cycle operations, not just load instructions. The weight of each instruction is based solely on the number of hierarchical dependencies.

Much research has been conducted on estimating delays in high-level synthesis. However, very little research has been performed on using estimated delays at high levels of abstraction in the context of operation chaining during scheduling of CDFGs for FPGA designs. Nemani and Najm [48] proposed a technique for measuring delays of combinational logic circuits, but their work is limited to Boolean functions. Nourani and Papachristou [49] presented a method of estimating delays for RTL logic in the context of false-path detection, in which they construct a Propagation Delay Graph to compute the critical delay in the design. Srinivasan et al. [57] described a system for estimating area and delay from behavioral RTL logic descriptions using best-fit polynomial models. Their method, however, requires logic synthesis of the design into a network of simple gates in order to estimate the delay. Xu and Kurdahi [71] presented an approach for

estimating area and timing delays for FPGA designs based on CLB and wire modeling, given an input logic netlist. Nayak et al. [47] developed an area and delay estimator for a high-level synthesis compiler that translates MATLAB code to RTL VHDL and Verilog for FPGAs. Their method of prediction is formulated as an equation based on constant parameters to be determined experimentally for each operation. Jiang et al. [30] presented a similar approach in which accurate high-level macro-model equations are used for estimating area, delay, and power of various RTL operations for a target FPGA architecture. Experimental values were obtained for each operation during high-level synthesis with varying precisions and the macro-model equation for the operations was extrapolated from a best-fit curve. Our method of estimating critical delays for our balanced chaining algorithm is based on their approach.

6.2 Balanced Scheduling

In conventional FPGA designs, performance is often optimized without regard for resource requirements. The goal is to reduce the number of clock cycles and increase the parallelism and frequency in the design, even at the cost of area. Typically, a scheduling method such as ASAP or ALAP is used, resulting in an imbalance in the number of instructions scheduled per clock cycle. In ASAP scheduling, a large number of operations are executed within the first few cycles, followed by fewer operations in the successive cycles. In ALAP scheduling, fewer operations are executed in the

beginning, followed by a large number of operations executed at the end. Both scheduling routines produce the same number of clock cycles. The benefit of balanced scheduling over ASAP and ALAP is a uniform distribution of operations among all clock cycles. Balanced instruction-level parallelism may also result in improved resource usage, latency, power, and heat dissipation. Figure 6.1 illustrates the ASAP, ALAP, and balanced scheduling routines. In the diagram, each operation node has a one-cycle latency. In balanced scheduling, there is an even distribution of operations among the four cycles, while the other methods show imbalances in the number of operations.

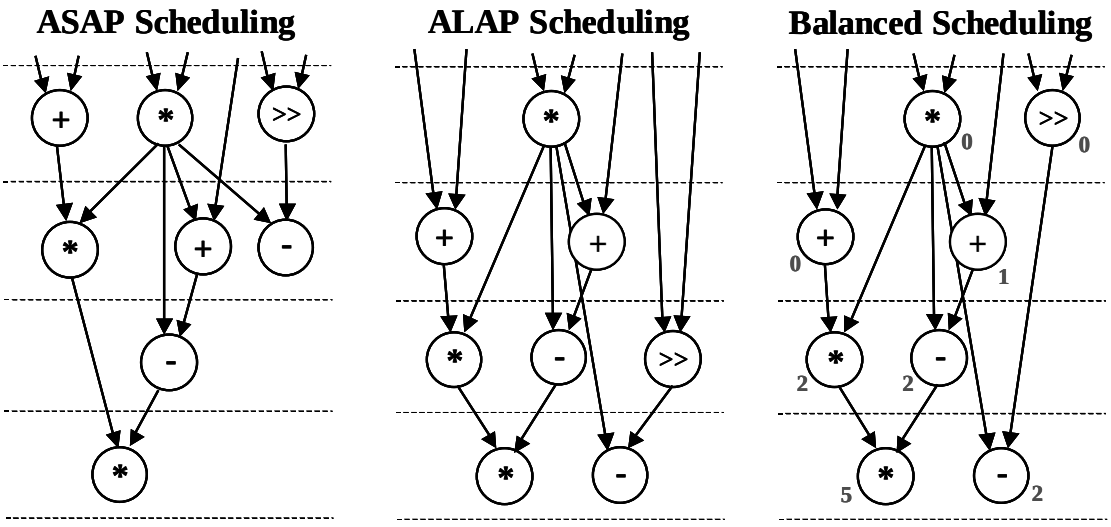


Figure 6.1. ASAP, ALAP and Balanced scheduling routines.

Balanced scheduling is implemented in two stages. In the first stage, dependency analysis is performed on each node, in which the total number of parent dependencies in the directed acyclic graph (DAG) hierarchy is determined. In Figure 6.1, the number of

node dependencies for each operation is shown for balanced scheduling. The second stage uses the number of dependencies to selectively forward nodes to later cycles in order to balance the instruction-level parallelism.

Figure 6.2 presents the algorithm for dependency analysis, which accepts as arguments a basic block, *B*, and a mapping, *D*, of each operation node to its number of dependencies. The algorithm iterates through the nodes in a block in topological order, while adding all unique predecessors to the node's dependency list. Finally, the size of each node's dependency list is assigned to the map, *D*. Clearly, the algorithm runs in linear time with the number of nodes in the CDFG.

```
 Dependency_Analysis( Block: B, Map: D )
1  topologically sort the nodes in B
2  DM is a mapping of nodes to their dependencies
3  for each node n in block B do
4    for each predecessor node p of n do
5      if p is an operation then
6        add each unique node in DM[p] to DM[n]
7        add p to DM[n]
8  for each node n in block B do
9    D[n] = DM[n].size()
```

Figure 6.2. Dependency analysis algorithm.

The balanced scheduling routine is presented in Figure 6.3, which uniformly distributes operation nodes among the cycles in the design. At first, the algorithm initializes the timesteps for each node in the CDFG using ASAP scheduling in line 2. In lines 3-35, it reiterates over the CDFG to optimize the load balance in each basic

block. In line 4, the dependency analysis is performed on the nodes in the block. Lines 5-10 count the number of operation nodes within the block, while adding each node to a map, *T*, which groups nodes by timestep. The average load balance per clock cycle is then computed. Lines 11-35 traverse *T* in reverse order, beginning with the latest cycle *(t)*. It searches for the best node to forward to that cycle by traversing each preceding cycle *(t-1, t-2, t-3, etc.)* until the earliest cycle in the block is reached. The forwarding node is selected based on two criteria: the node has the largest number of dependencies and forwarding the node does not violate any latency constraints, as described in lines 20-28. When a node is forwarded, *T* is updated by remapping the forwarded node to its new cycle. Once a cycle is balanced, or if it is not possible to balance it after traversing all preceding cycles, the algorithm continues on to balance the load in the next cycle *(t-1)* in *T*.

An analysis of the algorithm shows a worst case when no forwarding is possible for all nodes. This results in $O(nt)$ time complexity, where n is the number of operations in the block and t is the number of cycles after ASAP scheduling. If the DAG is very narrow, i.e., there is one node scheduled per state cycle such that $t = n$, the complexity resolves to $O(n^2)$. However, since most CDFGs have tree-like structures, on average we can expect $t = \log n$, yielding an average time complexity of approximately $O(n \log n)$.

```
  Balanced_Scheduling( Graph: G )
 1  D is a mapping of nodes to dependency counts
 2  ASAP_Scheduling( G )
 3  for each block b in G do
 4    Dependency_Analysis( b, D )
 5    num_opers = 0
 6    for each operation node n in b do
 7      n_opers = n_opers + 1
 8      add n to T[n->GetTimeStep()]
 9    cycles = b->GetEndTime() - b->GetStartTime()
10    avg_load = n_opers / cycles
11    for each element t in T in reverse order do
12      time = t.time
13      ptime = t.time - 1
14      while T[time].size() < avg_load and
15            ptime >= b->GetStartTime() do
16        max_depend = 0
17        best_node = NULL
18        for each node n of T[ptime] do
19          bool fwd_node = true
20          wb_time = time + n->getCycles()
21          if wb_time > b->GetEndTime() then
22            fwd_node = false
23          for each successor s of n do
24            if wb_time >= s->GetTimeStep() then
25              fwd_node = false
26          if fwd_node and D[n] > max_depend
27            max_depend = D[n]
28            best_node = n
29        if best_node != NULL then
30          best_node->SetTimeStep( time )
31          T[ptime].remove( best_node )
32          add best_node to T[time]
33        else if T[time].size() < avg_load then
34          ptime = ptime - 1
35        else break
```

Figure 6.3. Balanced scheduling algorithm.

To illustrate the balanced scheduling further, refer to Figure 6.1. The CDFG is initialized with ASAP scheduling, assuming each node has a latency of one cycle. Dependency analysis is performed on each node, as shown in the figure, and the load

balance is determined to be 2 (8 nodes / 4 states). In a bottom-up approach, beginning at the fourth state, one node is required to balance the load. The algorithm traverses the preceding states to find a node to forward to the fourth state. Beginning in the third state, a single *subtract* operation is found, but the 1-cycle latency prevents it from being forwarded. In the second state, the *multiply* and *subtract* operators have the same number of dependencies, but only the *subtract* operator has no latency restriction, and so it is selected to be forwarded to the fourth cycle. Since the fourth state is now balanced, we continue on to balance the third state, which also requires a single node. Beginning with state two, the *multiply* operation has the most dependencies and no latency restriction, so it is forwarded to state three. Now the second state requires a single node to balance. Looking to the first state, all three operations have the same number of dependencies, but only the *add* and *shift* operations have no latency restriction. The *add* operation is encountered first and is forwarded to the second state. Finally, the first state is already balanced and the procedure is complete.

6.3 Balanced Chaining

While balanced scheduling can significantly improve the distribution of operations among state cycles over ASAP and ALAP, it is somewhat naïve, for it does not implicitly consider variations in the operation delays. To obtain optimal

performance, it is essential to consider the operation delays when scheduling and chaining operations.

In this section a balanced chaining algorithm is presented, which uses a delay modeling technique to predict the critical path of a design. We also present our methodology for obtaining accurate delay models of operations during high-level synthesis. These models are used to obtain better timing performances during scheduling by optimally chaining RTL operations within each state of a finite state machine.

Consider the DAGs shown in Figure 6.4, scheduled using three different chaining routines. In the examples we assume that multiplication operations have delays of 5 ns, while addition and subtraction operations have delays of 2 ns. In the first case, ASAP scheduling without chaining would require 4 cycles. The critical delay, or the worst-case path delay, is 5 ns, resulting in a maximum allowed frequency is 200 MHz. It would take a total of 20 ns to complete the computations.

A naïve approach to operation chaining would schedule three multipliers within a single clock cycle, as illustrated in the second case in the figure. The resulting implementation takes 15 ns to complete, which is faster than the unchained approach. However, after closer inspection, it is apparent that the critical path can be balanced by chaining only the add-subtract operation sequence. This results in 3 cycles and a critical delay of 5 ns, as illustrated in the third case in the figure.

It is interesting to note that although the balanced chaining schedule runs in the same time as the unconstrained chaining schedule, more importantly, it produces a

maximum allowed frequency of 200 MHz compared to 67 MHz in the latter case. Consequently, the lower frequency yielded in the naïve approach limits the performance of the entire design.

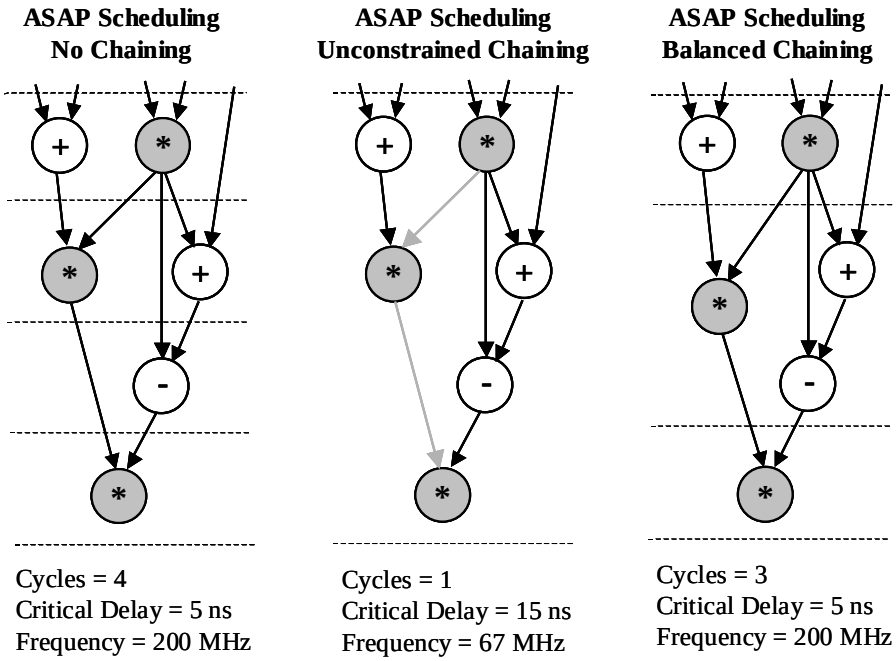


Figure 6.4. Comparison of chaining methods.

6.3.1 Modeling Delays

In order to obtain an optimal scheduling of operations, it is essential to accurately model operation delays. Generally, operation delays depend on the FPGA architecture and the precision of the calculation. It is therefore necessary to obtain delay estimates for varying precisions of each operation for each target FPGA architecture.

Our method of delay modeling is based on previous work by Jiang et al. [30], which demonstrates that it is possible to accurately model the delay, area, and power of operations for ASICs with constant, linear, and quadratic equations. This process consists of two steps: acquiring the operation delays for varying precisions and creating high-level equations to model the delay of each operation as a function of precision.

In acquiring operation delays, it is possible to use values at different stages during the high-level synthesis process. Primarily, values should be obtained after synthesis or place and route. We used delay values after synthesis using Synplicity's *Synplify Pro v8.0* tool. Our goal is to obtain frequency results within a small margin of error as compared to those reported by the synthesis tool. Since the synthesis tool generally performs many transformations on the RTL code, it is essential to obtain accurate values for each precision. This is accomplished by isolating operations from all other logic in the design. Figure 6.5 provides the Verilog code for a 16-bit multiplication operation, where the operation is isolated within the *OP* module. Figure 6.6 illustrates the technology mapping for the design on the Xilinx Virtex II FPGA, where the delay is measured within the *OP* module and excludes the external logic within the *MODEL* module. We use this method for acquiring delays of all arithmetic and logical operations with precisions of 2, 4, 8, 16, 32, and 64 bits. We also consider predicated operations, i.e., those embedded within if-then-else statements. The delay values are plotted and a best-fit curve is found to approximate the delay model as a function of precision.

```
module OP( IN1, IN2, OUT );
  parameter [31:0] WIDTH = 32;
  input wire signed [WIDTH/2-1:0] IN1;
  input wire signed [WIDTH/2-1:0] IN2;
  output wire signed [WIDTH-1:0] OUT;
  assign OUT = IN1 * IN2;
endmodule

module MODEL( IN1, IN2, OUT );
  parameter [31:0] WIDTH = 32;
  input wire signed [WIDTH/2-1:0] IN1;
  input wire signed [WIDTH/2-1:0] IN2;
  output wire signed [WIDTH-1:0] OUT;
  OP oper( IN1, IN2, OUT );   // operation instance
endmodule
```

Figure 6.5. Verilog code for modeling delays.

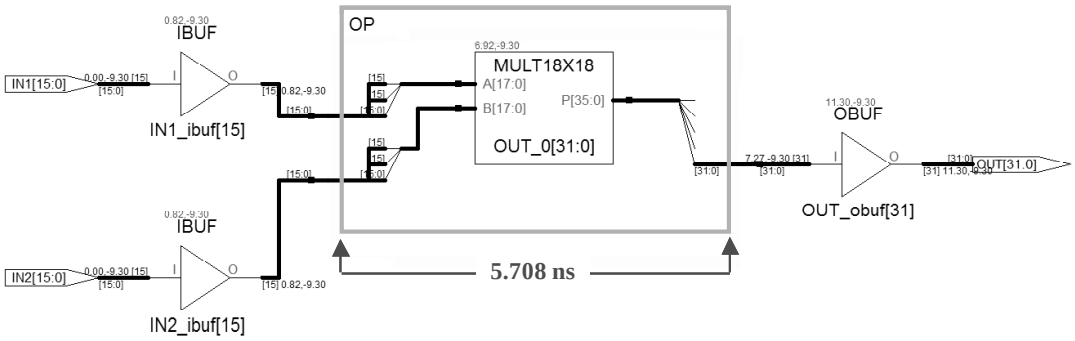


Figure 6.6. Measuring operation delays for FPGA designs.

Jiang et al. [30] have categorized their models into constant, linear, and quadratic equations. We have chosen to use a cubic equation in all delay models for the following reasons. We have found through experimentation that the delays of many operations in FPGAs were discontinuous linear functions of precision. These nonlinear characteristics are often due to high-level optimizations. For instance, a 2-bit *add* operation can be

replaced with a combination of simple logic gates, resulting in reduced delay for low precision operations. Similarly, operations with higher precisions often require additional levels of logic that increase the delay. Consequently, these optimizations produce varying slopes in the model, requiring cubic expressions to attain higher-accuracy in the delay models. Incidentally, operations that exhibit constant, linear, or quadratic properties can be modeled with cubic expressions as well.

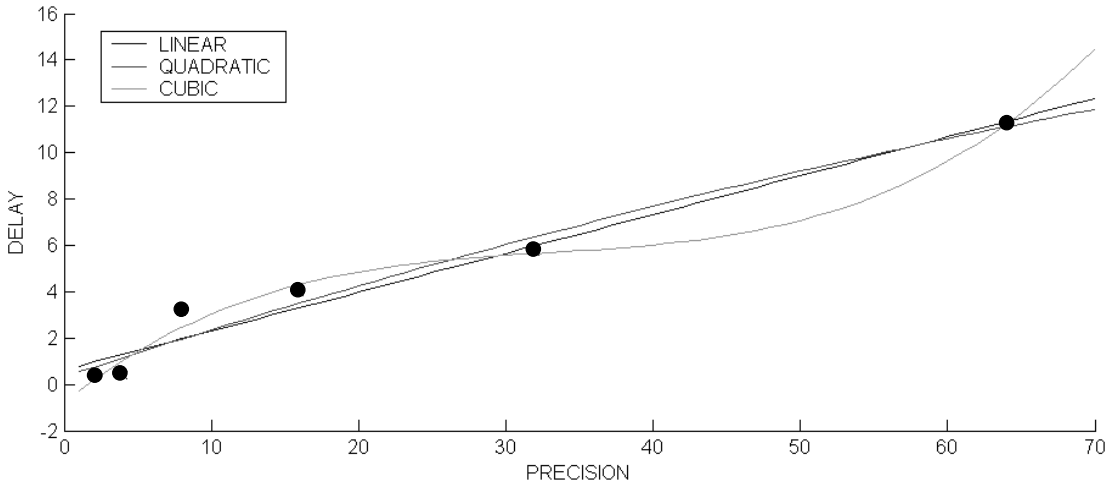


Figure 6.7. Comparison of delay models for a multiply operation.

Figure 6.7 illustrates a comparison of linear, quadratic, and cubic delay models for the multiply operation over a range of precisions, as described above. The best-fit curve model for the delays of each operation is obtained using the *POLYFIT* function in MATLAB. Clearly, the cubic model produces the highest accuracy.

In estimating the operation delays, we can expect a certain margin of error. If a consistent method was used in obtaining the delays for all operations, one can expect the margin of error to be relative among all the operation delay models. Therefore, we can justify that even with a margin of error, the critical path may nonetheless be identified correctly for determining the best chaining of operations.

6.3.2 Balanced Chaining Algorithm

In RTL VHDL and Verilog, operation chaining is accomplished by assigning the result of a computation using the blocking operator (=) rather than non-blocking operator (<=). This allows the resulting value to be used immediately instead of in the next clock cycle (state). Operation chaining is implemented on an operation node in a CDFG by assigning a cycle delay of zero. The cycle delay is used when determining a schedule for each node in a finite state machine.

Figure 6.8 presents our balanced chaining algorithm, whose objective is to minimize the clock cycles and critical path in the input graph, *G*, given an input target *frequency*. The target frequency is upper-bounded by the slowest operation in the design and lower-bounded by 1 MHz, which is essentially equivalent to running unconstrained chaining. This chaining method uses a uniform cycle delay between each operation and its successor nodes; it does not consider independent cycle delays for each outgoing edge. However, the algorithm can be easily adapted to handle varying cycle delays.

```
  Balanced_Chaining( Graph: G, frequency )
 1  if frequency < 1.0 then frequency = 1.0
 2  critical_delay = 1000 / frequency
 3  D is a mapping of each operation node to a delay
 4  for each block b in G do
 5    for each node n in b do
 6      D[n] = Get_Operation_Delay( n )
 7      if ( D[n] > critical_delay ) then
 8        critical_delay = D[n]
 9      if n has successors and n->getCycles() == 1
10        n->setCycles( 0 )
11  for each block b in G do
12    topologically sort nodes in b
13    for each node n in b in reverse order do
14      for each predecessor node p of n do
15        D[p] = Get_Operation_Delay( p )
16        total_delay = D[p] + D[n]
17        if p->getCycles() == 0 then
18          if total_delay > critical_delay then
19            p->setCycles( 1 )
20          else if total_delay <= critical_delay
21               and total_delay > D[p] then
22            D[p] = total_delay
```

Figure 6.8. Balanced chaining algorithm.

The algorithm begins in lines 3-10 by iterating through the nodes in the CDFG. In lines 6-8, each operation node is mapped to its predicted delay for the target architecture based on the model described above. While doing so, it also updates the *critical delay*, which is the worst-path delay of any sequence of chained operations in the CDFG. In lines 9-10, the CDFG is initialized to unconstrained chaining by assigning a cycle delay of zero for each operation node.

The chaining is performed in lines 11-22. The nodes in the CDFG are first topologically sorted, and then traversed from *bottom-up*. The delay of each node is recalculated based on the delay of its predecessor nodes in lines 15-16. If the combined

delay of a node and its predecessor is greater than the *critical delay*, the predecessor node is unchained by setting its cycle delay to one cycle in lines 18-19. Otherwise, the operation delay of the predecessor node is updated with the *total delay* in lines 20-22. It is evident that the algorithm runs in linear time with the number of nodes in the CDFG.

6.4 Experimental Results

In this section we present timing results for ten benchmarks using the scheduling routines discussed in this chapter. The benchmarks were originally available in C, and compiled to assembly code using the Texas Instruments *Code Composer Studio* software suite, targeting the C6211 DSP architecture. The assembly codes were compiled to RTL VHDL and Verilog using the FREEDOM compiler, and the CDFGs were unrolled several times to increase the design size. Using the scheduling techniques outlined here, the designs were synthesized to target two different FPGA architectures: the Xilinx Virtex II FPGA and the Altera Stratix FPGA. The RTL codes generated by the FREEDOM compiler were simulated using the *ModelSim v5.7e* simulation tool from Mentor Graphics. In each case, the design was verified to be bit-true accurate. The RTL codes were synthesized using the *Synplify Pro v8.2* logic synthesis tool from Synplicity. These synthesis results were used to obtain estimated frequencies for each benchmark. The execution times on the FPGAs were measured using the number of clock cycles in the simulation and the frequency results from the synthesis process. The estimated delay

models for balanced chaining were based on data gathered for this FPGA architecture using the *Synplify Pro v8.2* logic synthesis tool. Table 6.1 and Table 6.2 present the measured operation delays for each operation and their predicted delay model on the Xilinx Virtex II and Altera Stratix FPGAs, respectively. The delay model for each operation is in the format: $C_3*X^3 + C_2*X^2 + C_1*X + C_0$.

Table 6.1. Delay models for operations on the Xilinx Virtex II FPGA.

	Measured Operation Delays for Varying Bit-Widths						Constant Coefficients for Delay Models			
Operation	**2**	**4**	**8**	**16**	**32**	**64**	**C_3**	**C_2**	**C_1**	**C_0**
ADD	0.383	0.900	1.041	1.456	2.192	3.655	0.0000	-0.0019	0.1000	0.3414
SUB	0.383	0.900	1.041	1.456	2.192	3.655	0.0000	-0.0019	0.1000	0.3414
MULT	0.382	0.382	3.140	3.996	5.708	11.236	0.0001	-0.0143	0.5116	-0.7996
ASSGN	0.000	0.000	0.000	0.000	0.000	0.000	0.0000	0.0000	0.0000	0.0000
UNION	0.000	0.000	0.000	0.000	0.000	0.000	0.0000	0.0000	0.0000	0.0000
NEG	0.383	0.383	0.900	2.097	2.282	3.156	0.0000	-0.0056	0.2088	-0.2173
NOT	0.382	0.382	0.382	0.382	0.382	0.382	0.0000	0.0000	0.0000	0.3820
AND	0.382	0.382	0.382	0.382	0.382	0.382	0.0000	0.0000	0.0000	0.3820
OR	0.382	0.382	0.382	0.382	0.382	0.382	0.0000	0.0000	0.0000	0.3820
XOR	0.382	0.382	0.382	0.382	0.382	0.382	0.0000	0.0000	0.0000	0.3820
XNOR	0.382	0.382	0.382	0.382	0.382	0.382	0.0000	0.0000	0.0000	0.3820
NAND	0.382	0.382	0.382	0.382	0.382	0.382	0.0000	0.0000	0.0000	0.3820
NOR	0.382	0.382	0.382	0.382	0.382	0.382	0.0000	0.0000	0.0000	0.3820
CMPEQ	0.382	0.901	0.901	1.419	1.938	1.938	0.0000	-0.0015	0.0903	0.3459
CMPNE	0.382	0.901	0.901	1.419	1.938	1.938	0.0000	-0.0015	0.0903	0.3459
CMPLT	0.382	0.810	0.810	0.810	1.870	2.605	0.0000	0.0021	-0.0038	0.5908
CMPGT	0.382	0.810	0.810	0.810	1.870	2.605	0.0000	0.0021	-0.0038	0.5908
CMPLE	0.382	1.544	1.544	1.544	2.603	3.339	0.0000	-0.0015	0.0948	0.6766
CMPGE	0.382	1.544	1.544	1.544	2.603	3.339	0.0000	-0.0015	0.0948	0.6766
SLL	0.383	0.900	1.547	2.191	2.877	3.649	0.0000	-0.0055	0.2139	0.0652
SRL	0.383	0.900	1.547	2.205	2.799	3.803	0.0001	-0.0060	0.2224	0.0425
SRA	0.383	1.115	2.523	3.202	3.844	4.431	0.0001	-0.0114	0.3856	-0.2246
PRED	0.382	0.901	0.901	1.419	1.419	1.938	0.0000	-0.0040	0.1254	0.2560

Table 6.2. Delay models for operations on the Altera Stratix FPGA

	Measured Operation Delays for Varying Bit-Widths						Constant Coefficients for Delay Models			
Operation	2	4	8	16	32	64	C_3	C_2	C_1	C_0
ADD	0.527	1.078	1.462	1.702	2.182	3.142	0.0000	-0.0040	0.1437	0.4210
SUB	0.527	1.058	1.462	1.702	2.182	3.142	0.0000	-0.0041	0.1446	0.4108
MULT	0.244	0.527	6.065	6.065	6.065	4.974	0.0003	-0.0350	1.0330	-1.8792
ASSGN	0.000	0.000	0.000	0.000	0.000	0.000	0.0000	0.0000	0.0000	0.0000
UNION	0.000	0.000	0.000	0.000	0.000	0.000	0.0000	0.0000	0.0000	0.0000
NEG	0.244	0.527	1.342	1.462	1.702	2.182	0.0001	-0.0059	0.1855	-0.0524
NOT	0.000	0.000	0.000	0.000	0.000	0.000	0.0000	0.0000	0.0000	0.0000
AND	0.244	0.244	0.244	0.244	0.244	0.244	0.0000	0.0000	0.0000	0.2440
OR	0.244	0.244	0.244	0.244	0.244	0.244	0.0000	0.0000	0.0000	0.2440
XOR	0.244	0.244	0.244	0.244	0.244	0.244	0.0000	0.0000	0.0000	0.2440
XNOR	0.244	0.244	0.244	0.244	0.244	0.244	0.0000	0.0000	0.0000	0.2440
NAND	0.244	0.244	0.244	0.244	0.244	0.244	0.0000	0.0000	0.0000	0.2440
NOR	0.244	0.244	0.244	0.244	0.244	0.244	0.0000	0.0000	0.0000	0.2440
CMPEQ	0.527	1.058	1.485	2.017	2.444	2.836	0.0000	-0.0048	0.1785	0.2999
CMPNE	0.527	1.058	1.485	2.017	2.444	2.836	0.0000	-0.0048	0.1785	0.2999
CMPLT	0.527	1.342	1.462	1.702	2.182	3.142	0.0000	-0.0037	0.1314	0.5554
CMPGT	0.527	1.342	1.462	1.702	2.182	3.142	0.0000	-0.0037	0.1314	0.5554
CMPLE	0.527	1.342	1.462	1.702	2.182	3.142	0.0000	-0.0037	0.1314	0.5554
CMPGE	0.527	1.342	1.462	1.702	2.182	3.142	0.0000	-0.0037	0.1314	0.5554
SLL	0.527	1.590	1.587	2.368	2.532	3.634	0.0001	-0.0069	0.2178	0.3928
SRL	0.527	1.590	1.587	2.368	2.532	3.634	0.0001	-0.0069	0.2178	0.3928
SRA	0.527	1.590	2.244	2.939	3.577	4.860	0.0001	-0.0087	0.2996	0.2380
PRED	0.527	0.971	1.435	2.314	3.192	4.169	0.0000	-0.0038	0.1842	0.2187

Table 6.3 and Table 6.4 show timing results for the Xilinx Virtex II and Altera Stratix FPGAs using ASAP, ALAP, and balanced scheduling with unconstrained chaining. These results produce the minimum number of cycles possible among the

scheduling routines, which are consistent for both architectures. The objective is then to maximize the frequency within this time frame.

With balanced scheduling, a uniform distribution of operations among the cycles improves the frequency dramatically over ASAP and ALAP for nearly all benchmarks. This is due to the fact that the scheduling has partitioned the large critical paths in the design. Variations in frequency for *laplace* results for the Xilinx Virtex II FPGA, as well as *ellip* and *sobel* results for the Altera Stratix FPGA are due to the naïve partitioning of the critical path. Consequently, minor variations in the data path can cause the back-end synthesis tool to make substantial changes in design implementations that can affect the overall frequency.

Table 6.3. Comparison of scheduling routines for Xilinx Virtex II FPGA.

		ASAP		ALAP		BALANCED	
Benchmark	Cycles	Freq (MHz)	Time (µs)	Freq (MHz)	Time (µs)	Freq (MHz)	Time (µs)
dot_prod	1204	54.8	**22.0**	68.5	**17.6**	111.5	**10.8**
iir	2704	54.6	**49.5**	50.1	**54.0**	58.2	**46.5**
matmul_32	111909	103.0	**1086.5**	70.9	**1578.4**	103.0	**1086.5**
gcd	66	215.6	**0.3**	211.1	**0.3**	215.6	**0.3**
diffeq	58	20.4	**2.8**	26.6	**2.2**	42.4	**1.4**
ellip	53	124.9	**0.4**	146.8	**0.4**	166.9	**0.3**
laplace	5528	150.5	**36.7**	109.7	**50.4**	131.8	**41.9**
fir16tap	14948	103.7	**144.1**	71.3	**209.6**	103.7	**144.1**
fir_cmplx	2852	54.4	**52.4**	47.8	**59.7**	78.7	**36.2**
sobel	18891	108.8	**173.6**	80.6	**234.4**	111.7	**169.1**

Table 6.4. Comparison of scheduling routines for Altera Stratix FPGA.

Benchmark	Cycles	ASAP		ALAP		BALANCED	
		Freq (MHz)	Time (µs)	Freq (MHz)	Time (µs)	Freq (MHz)	Time (µs)
dot_prod	1204	56.4	**21.3**	56.4	**21.3**	89.6	**13.4**
iir	2704	56.5	**47.9**	46.0	**58.8**	55.3	**48.9**
matmul_32	111909	89.6	**1249.0**	42.9	**2608.6**	89.6	**1249.0**
gcd	66	159.8	**0.4**	176.2	**0.4**	159.8	**0.4**
diffeq	58	11.1	**5.2**	13.8	**4.2**	26.7	**2.2**
ellip	53	119.6	**0.4**	165.3	**0.3**	147.7	**0.4**
laplace	5528	89.3	**61.9**	100.4	**55.1**	100.8	**54.8**
fir16tap	14948	89.6	**166.8**	42.9	**348.4**	89.6	**166.8**
fir_cmplx	2852	51.0	**55.9**	36.5	**78.1**	57.8	**49.3**
sobel	18891	94.1	**200.8**	67.5	**279.9**	86.7	**217.9**

Table 6.5 and Table 6.6 show comparisons of the same benchmarks for the two FPGA architectures using no chaining, unconstrained chaining, and balanced chaining. For the latter case, a target frequency of 500 MHz was chosen, resulting in the maximum allowed frequency for the design. As expected, the clock cycles increased over that of the unconstrained chaining due to reduced chaining of operations. For the Xilinx Virtex II FPGA, we see dramatic increases in frequency as well as significant improvements in timing over that of an unconstrained chaining approach. The delay estimations for *ellip*, *laplace*, and *sobel* were hampered by extra logic inserted by the synthesis tool. However, the difference was within a 10% margin of error as compared to the best results. It is interesting to note that the frequency results for balanced chaining are very close to those reported for scheduling without chaining, which is the maximum capable frequency expected for each benchmark. This strengthens our claim that it is indeed possible to

effectively predict the critical delays of a design at an abstract level in order to optimize the chaining of operations during scheduling.

The results for the Altera Stratix FPGA seemed to be less consistent than that of the Xilinx Virtex II. The resulting frequency estimations for *dot_prod*, *matmul_32*, *diffeq*, and *fir16tap* were much lower than the best case (with no chaining). After further investigation, it seems the synthesis tool optimized away registers between cycles for the MAC operations, which increased the overall critical path. These transformations were unpredictable and could not be disabled. While these frequency results were significantly less than the best case, it is interesting to note that they were no worse than that of unconstrained chaining.

Table 6.5. Comparison of chaining routines for Xilinx Virtex II FPGA.

Benchmark	NONE			UNCONSTRAINED			BALANCED		
	Cycles	Freq (MHz)	Time (µs)	Cycles	Freq (MHz)	Time (µs)	Cycles	Freq (MHz)	Time (µs)
dot_prod	1654	145.1	**11.4**	1204	111.5	**10.8**	1304	146.2	**8.9**
iir	6306	103.8	**60.8**	2704	58.2	**46.5**	4204	136.0	**30.9**
matmul_32	171528	146.1	**1174.0**	111909	103.0	**1086.5**	120101	146.2	**821.5**
gcd	118	187.8	**0.6**	66	215.6	**0.3**	67	215.6	**0.3**
diffeq	156	143.1	**1.1**	58	42.4	**1.4**	94	143.3	**0.7**
ellip	66	168.1	**0.4**	53	166.9	**0.3**	59	161.4	**0.4**
laplace	9221	189.5	**48.7**	5528	150.5	**36.7**	6638	173.1	**38.3**
fir16tap	23386	145.1	**161.2**	14948	103.7	**144.1**	15912	145.2	**109.6**
fir_cmplx	3924	102.3	**38.4**	2852	78.7	**36.2**	3012	102.3	**29.4**
sobel	33563	148.0	**226.8**	18891	111.7	**169.1**	22611	132.0	**171.3**

Table 6.6. Comparison of chaining routines for Altera Stratix FPGA.

Benchmark	NONE			UNCONSTRAINED			BALANCED		
	Cycles	Freq (MHz)	Time (µs)	Cycles	Freq (MHz)	Time (µs)	Cycles	Freq (MHz)	Time (µs)
dot_prod	1654	146.7	**11.3**	1204	89.6	**13.4**	1304	89.6	**14.6**
iir	6306	55.5	**113.6**	2704	56.5	**47.9**	4204	131.6	**31.9**
matmul_32	171528	135.4	**1266.8**	111909	89.6	**1249.0**	120101	89.6	**1340.4**
gcd	118	141.8	**0.8**	66	176.2	**0.4**	67	180.8	**0.4**
diffeq	156	124.0	**1.3**	58	26.7	**2.2**	94	75.7	**1.2**
ellip	66	152.1	**0.4**	53	165.3	**0.3**	59	162.7	**0.4**
laplace	9221	104.4	**88.3**	5528	100.8	**54.8**	6638	112.6	**59.0**
fir16tap	23386	127.3	**183.7**	14948	89.6	**166.8**	15912	89.6	**177.6**
fir_cmplx	3924	53.1	**73.9**	2852	57.8	**49.3**	3012	138.1	**21.8**
sobel	33563	89.8	**373.8**	18891	94.1	**200.8**	22611	96.0	**235.5**

6.5 Summary

When good delay estimations are not available for FPGAs, high-level synthesis tools often use unconstrained chaining in scheduling to reduce the number of cycles in the design. In this chapter we present a balanced scheduling routine that uniformly distributed operations among states. This effectively breaks up large critical paths in the design and improves the frequency. Results indicate that in general, this technique performs better than ASAP or ALAP scheduling with unconstrained chaining for different FPGA architectures.

With good delay estimation and modeling methods, better-quality chaining is possible. Towards this effort, we have developed precision-based delay models to estimate operation delays in FPGAs. This technique was incorporated in our balance

chaining routine. Given a target frequency, balanced chaining uses these delay models to reduce the cycles and critical path in the design by chaining operations within the given critical delay. Results show significant improvements in frequency and run times using our balanced chaining routine, even with ASAP and ALAP scheduling. Furthermore, our method for modeling operation delays is shown to accurately identify the critical paths of complex designs during high-level synthesis for different FPGA architectures. Consequently, when using balanced chaining, the balanced scheduling technique is no longer essential.

Chapter 7

Resource Sharing

As part of the translation process of DSP assembly code into hardware descriptions by the FREEDOM compiler, a grammar is provided to map each operation of the source ISA to a set of MST instructions. The MST instructions are then converted into a CDFG that is composed of nodes representing inputs, outputs, and operations, which are optimized for performance in terms of frequency, power, timing, and area. It is interesting to observe that such methods of translation in high-level synthesis compilers often produce reoccurring patterns in the resulting CDFG. As an example, suppose a design targeted for high-level synthesis contains a number of calls to a multiply-accumulate (MAC) function. The grammar might translate each of these calls to a sequence of multiply-add operations in the CDFG. We surmise that it is possible to re-associate these reoccurring patterns of instruction sequences into collections of operation sets, or templates, to be shared in a hardware implementation to reduce area

and facilitate better placement and routing. Additionally, the templates may be used for IP blocks and other specialized complex computational units.

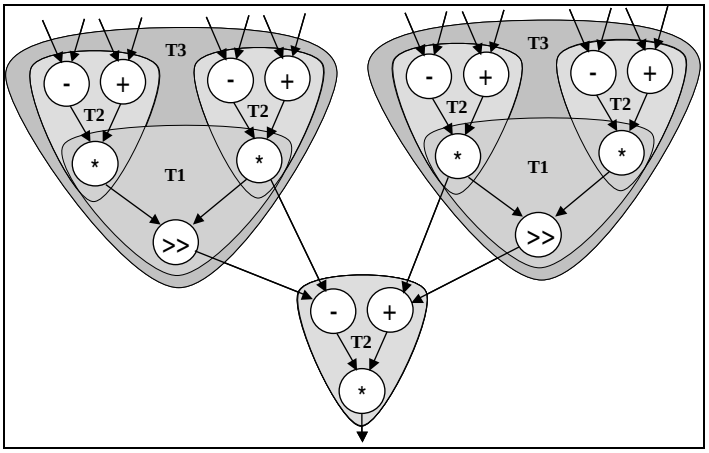


Figure 7.1. Reoccurring patterns in a CDFG.

Figure 6.6 illustrates a CDFG with reoccurring patterns of operations, allowing several possible resource sharing schemes. A simple implementation would share only individual operations, such as adders and multipliers, although more complex patterns may also be extracted, namely T1, T2, or T3. Sharing individual operations alone will result in the removal of 13, considering the added cost of implementing each template. However, the cost of routing and steering logic necessary to share a single operator for an entire design is unacceptably high. Similarly, one might assume it is preferable to use a small number of large templates in resource sharing. However, growing template T3 only removes 7 operations. Consequently, implementing template T2 would produce the best resource coverage. We see from this example two important aspects in template

generation: a cost function is essential in the decision factor for growing templates, and the template selection and growth has a direct impact on future decisions. For instance, selecting the template T1 will prevent the growth of template T2 due to overlapping structures, thus resulting in a suboptimal area reduction.

Regularity extraction is the term generally used for extracting reoccurring patterns from CDFGs. This approach is related to the technology mapping and the graph covering problems. However, in the classical technology mapping problem, a static library of modules is provided with the goal of completely covering the CDFG with a mapping of each operation to a gate in the technology library. At higher levels of abstraction, these static libraries may not be readily available. Consequently, regularity extraction requires a more dynamic approach.

The contribution of this work is an area optimization algorithm that uses regularity extraction to generate templates of reoccurring patterns to enable resource sharing. We use a heuristic method to grow templates dynamically based on a cost function and the frequency of reoccurring patterns within the CDFG. Additionally, we implement backtracking and vary the look-ahead depth when growing templates to evaluate the efficiency of the algorithm in obtaining near-optimal solutions in practice. Preliminary work on this research has been published [73].

7.1 Related Work

Many approaches have been evaluated for resource sharing. The graph-covering problem may be solved optimally using a binate covering formulation. However, the problem is NP-Complete; the optimal binate covering approach is intractable for large problem instances. Villa et al. [66] proposed methods for obtaining results more quickly by implementing better bounding techniques. However, even at its best, using binate covering to cover large CDFGs can take an extremely long time to find an optimal covering solution. Many researchers have resorted to heuristics in order to obtain solutions quickly. Memik et al. [39] proposed a method of resource sharing for FPGAs using a combination of five heuristics to merge resources across basic blocks in a CDFG. They assume a technology library is provided for the target FPGA and the operation nodes have been annotated with area and delay measurements. We present a more general approach that may be used when this information is not readily available. Our method utilizes regularity extraction to organize reoccurring operation networks in a CDFG into separate templates to enable resource sharing.

There has been a significant amount of work on regularity extraction. In general, two methods are used. The first method uses a static library of templates. The second method, which we have implemented, dynamically generates templates on the fly. Callahan et al. [6] proposed a method for mapping operations to target FPGAs given a library of templates to match. Prior to performing graph covering, the DAGs are

partitioned into trees. This approach, however, can be very costly when trying to minimize area, as the number of operations will increase significantly. Furthermore, they only allow single-output templates, whereas our proposed method supports multiple fanout. Chowdhary et al. [8] have proposed dynamic template generation that considers the complete set of templates for trees and single-output DAGs and multiple fanouts. The complexity of considering all possible templates can be high. Kastner et al. [34] and Rao and Kurdahi [56] and have proposed methods for extracting templates from DAGs based on incremental template matching. However, this greedy approach is subject to ill-fated decisions that may prevent future template growth or become trapped in local minima. Our proposed method includes a varying look-ahead to consider more levels of hierarchy simultaneously in order to avoid this problem. Guo et al. [25] proposed a similar method for the MONTIUM architecture that allows a limit to the generated template sizes. However, templates are grown based on frequency of occurrence; they do not consider a cost function. Additionally, there is no discussion of the degree to which their solutions deviate from optimality. In our method, we have implemented a cost function and backtracking to determine the point of diminishing returns for a number of benchmarks, allowing near-optimal solutions to be efficiently generated.

7.2 Dynamic Resource Sharing

This section presents our heuristic algorithm for dynamic resource sharing. We discuss our methods of building template matches using regularity extraction, and selecting which templates to implement for resource sharing via a cost function.

7.2.1 Linear DAG Isomorphism

The key to regularity extraction is identifying common patterns in a graph. CDFGs composed of directed acyclic graphs (DAGs) pose a more difficult problem than trees. Unlike trees, in which dynamic programming may be used to find optimal solutions for template matching, DAGs are commonly approached using heuristics to identify isomorphic patterns. This process becomes more complicated for large DAGs and with multiple fanouts. Consider the example in Figure 7.2, where two circuits are structured differently, but perform the same function (assuming the inputs for the two are the same). Evidently, the second structure may be used in place of the first structure if the same inputs are used in both multipliers. However, in some cases these structures may in fact not be identical due to other properties, such as sign and precision.

There has been much work on DAG isomorphism. Gemini [21] was developed as a means of validating circuit layout through graph isomorphism using a graph coloring technique. Rao and Kurdahi [56] proposed a string matching technique for comparing graphs, but they use very simple string formulas (called *K-formulas*).

Zibin et al. [77] have described efficient methods of linear isomorphism by utilizing the commutative and associative properties of operations. We have implemented a similar approach using string matching that encapsulates additional properties of each operation, including predicates, sign and precision. The algorithm is limited to single-rooted DAGs, but allows for multiple fanouts as described later in section 7.2.6.

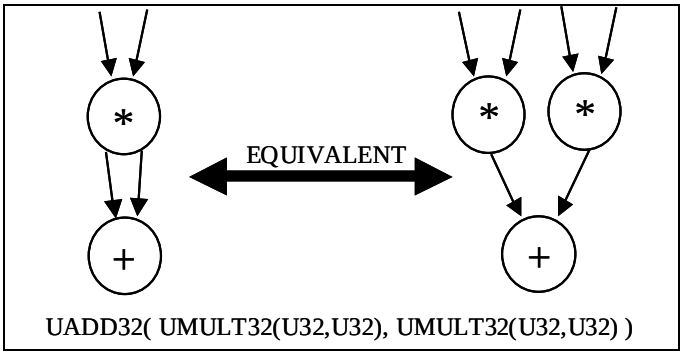


Figure 7.2. Equivalent DAG structures.

Figure 7.3 describes the procedure *GenerateExprs()*, which takes as input a *nodeset* from a single-rooted DAG, and a hash table, *N*, that maps string expressions to nodes. The *nodeset* is topologically sorted in order of leaves to the root. Beginning with the leaves, the algorithm calls the *HashNode()* function on each node in the *nodeset* to generate a string value based on the properties of the operation and the hash expressions of its input nodes. In Figure 7.2, the hash value *UMULT32* is used to represent an unsigned 32-bit multiplication operation, while the inputs, *U32*, are represented abstractly using only sign and precision properties. Similarly, the 32-bit unsigned add operation is represented as a function of its inputs. The functionality may be expanded to

handle other properties, such as bit-select and predicated operations. *HashNode()* utilizes the associative and commutative properties of symmetric operations by sorting the input hash expressions alphanumerically. Two DAGs are said to be isomorphic if the hash values of the root nodes are equivalent, i.e., *N[root1] = N[root2]*.

```
 GenerateExprs(nodeset, N)
1  Topologically sort the nodes in nodeset
2  for each node n in nodeset
3    for each incoming edge p of n do
4      add N[p] to expr_list
5    N[n] = HashNode(n, expr_list)
```

Figure 7.3. Pseudo-code for DAG expressions.

7.2.2 Growing Templates

Templates are grown by first building *nodesets* and then matching structures as described above. A *nodeset* is built by performing a depth-first traversal of a DAG beginning at a root node, and incrementally adding the nodes at each stage of the hierarchy. In Figure 7.4, we present a recursive function, *BuildNodeSet()* that grows a nodeset. The function takes as input the current *node*, the *nodeset* to which the nodes are added, the current *level* in the hierarchy, a *max_level* of hierarchy, and a table *P* that maps each node to its current parent nodeset. In each stage, *node* is added to *nodeset* if *max_level* has not been reached and no reconverging paths are found, as described in the next section. A nodeset may only be combined with another via its *root* node. Consequently, the procedure checks that *node* is a root prior to combining sets. The

procedure then recursively calls itself on each input to *node*. The recursion is bounded by *max_level* look-ahead, or the height of the DAG if it is the smaller of the two. Figure 7.5 illustrates the nodesets generated from a DAG with varying look-ahead depths of 0, 1, and 2.

```
 BuildNodeSet(node, nodeset, level, max_level, P)
 1  if level > max_level then return
 2  else if PathReconverges(node, nodeset, false)
 3    return
 4  pset = GetParentSet(node, P)
 5  if pset != NULL then
 6    if node != pset->root then return
 7    else add pset->nodes to nodeset
 8  else add node to nodeset
 9  for each node n in nodeset do
10    for each input edge p of n outside nodeset do
11      BuildNodeSet(p,nodeset,level+1,max_level,P)
```

Figure 7.4. Pseudo-code for growing nodesets.

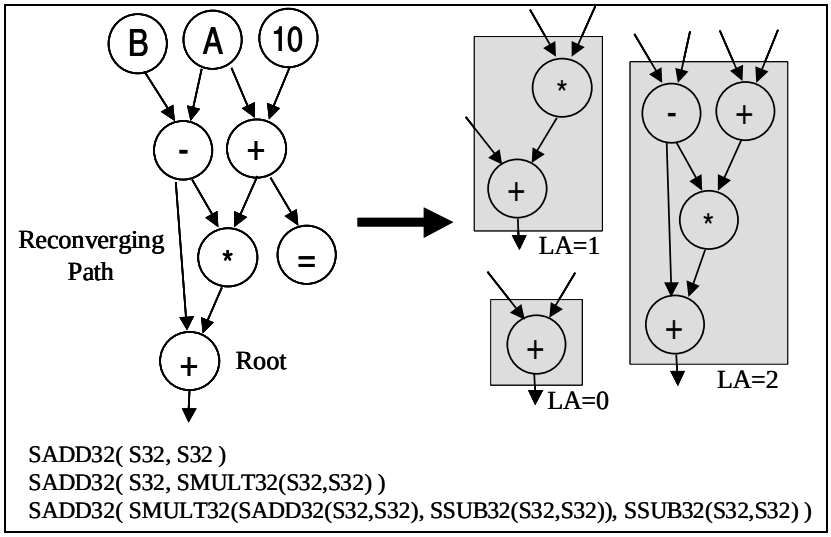


Figure 7.5. Generated nodesets and expressions.

7.2.3 Reconverging Paths

The order in which templates are grown is important. If there exists a secondary path between a nodeset and a joining node in which an intermediate node lies, the node may not be added to the set unless the node on the intermediate path is first included. Otherwise, a cycle is produced. Referring to Figure 7.5, we see that the root node's first input, the subtract node, has a reconverging path through the multiply operation. Had the subtract node been added first, the multiply node would become both an input and output to the nodeset, resulting in a cycle. Consequently, only the multiply node is added in the second stage, while the subtract node is added in the third stage.

```
 PathReconverges(node, nodeset, diverged)
 1  for each outgoing edge s of node do
 2    if nodeset contains s then
 3      if diverged == false then
 4        continue
 5      else if diverged == true then
 6        return true
 7    else if s has not been visited then
 8      if PathReconverges(s, nodeset, true) then
 9        return true
10  return false
```

Figure 7.6. Pseudo-code for reconverging paths.

Figure 7.6 describes a recursive algorithm for determining if *node* has a reconverging path to *nodeset*. The procedure performs a depth-first traversal of all paths from *node* by calling itself on each successor node. If a node on a path does not belong

to *nodeset*, the node is said to have diverged. Consequently, the presence of any subsequent nodes along the same path to *nodeset* indicates a reconverging path.

7.2.4 Template Matching

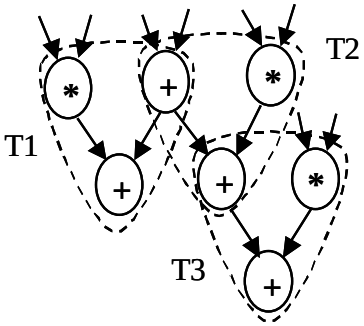
We have described a method of growing nodesets for templates and a means of generating expressions for nodeset equivalence checking. We now describe a technique for generating template matches that employs these methods. Figure 7.7 presents a procedure *GenerateExprTable()* for generating template matches. It takes as input a graph *G*, a hash table *E* that maps a list of equivalent nodesets to a hash expression, a table *P* that maps a node to its parent nodeset, and a *look-ahead* factor for generating nodesets.

The procedure iterates through each node in *G* and builds nodesets based on the look-ahead value provided. It considers all permitted levels of look-ahead simultaneously. For instance, a look-ahead value of 2 produces nodesets for look-aheads of 0, 1, and 2. Similarly, a look-ahead of infinity (∞) considers all possible nodeset combinations for each node in the graph. Once a nodeset is built, a hash expression is generated for the DAG. This expression is used as the key for adding the nodeset to table *E*. To bound the size of the table, the procedure terminates if an empty hash value is found or if the hash value is the same as that in a previous stage. Figure 5 shows the nodesets generated for a look-ahead of 2 and the corresponding hash expressions generated at each level.

```
  GenerateExprTable(G, E, P, look_ahead)
 1  for each block b in G
 2    for each node n in b
 3      for i = 0 to look_ahead do
 4        nodeset is a collection of nodes
 5        N is a hash table of string to node
 6        nodeset->root = node
 7        BuildNodeSet(node, nodeset, 0, i, P)
 8        GenerateExprs(nodeset, N)
 9        key = N[nodeset->root]
10        if nodeset is empty then break
11        else if key was already seen then break
12        else add nodeset to E[key]
13  for each element e in E do
14    GetMaximalIndependantSets(e, P)
15  PruneUnmatchedSets(E, P)
```

Figure 7.7. Pseudo-code for template matching.

The result of the procedure is a complete mapping of expressions to a list of equivalent nodesets. It is, however, necessary to remove any conflicts that arise locally within each set of template matches. Villa et al. [66] describe a method of obtaining a close approximation of the maximal independent set using an adjacency matrix, in which the number of nodesets is maximized by first selecting those that conflict with the least number of nodesets, and then eliminating any rows/columns that intersect with a ‘1’. Templates that do not have at least two matches are pruned, since there is no benefit to instantiating single-match templates. Figure 7.8 illustrates a set of conflicting template matches and the corresponding adjacency matrix. Selecting either nodeset T1 or T3 first would result in the maximum attainable matches. Conversely, selecting T2 first would have eliminated matches T1 and T3, thereby producing suboptimal results.



	T1	T2	T3
T1	1	1	0
T2	1	1	1
T3	0	1	1

Figure 7.8. Adjacency matrix for overlapping sets.

7.2.5 Selecting Templates

Once the table of possible template matches is constructed, it is necessary to determine the best templates to implement. Typically, each choice has a direct impact on future choices, as there may yet exist conflicts between nodesets of different templates that must be eliminated. Figure 7.9 describes a procedure, *SelectTemplateMatches(),* that iteratively selects the best template matches from *E* using a cost function based on the number of operations covered, less the cost of implementing the template with replicated hardware (see section 7.2.6). The corresponding list of matches is placed into *T*. All remaining nodesets in *E* that conflict with these matches are removed, and unmatched sets are pruned. This process continues until *E* is either empty or no suitable template matches are available. If a single template match is found in *T* after the other matches were combined into larger templates, the template is pruned and its nodes are added back to the pool for future template matching.

Different techniques may be considered when choosing a cost function. If precise area measurements are available for operations, it may be feasible to obtain nearly minimal area solutions. For our purposes, we chose a wiring cost for several reasons. First, it is often difficult to approximate the area cost at an abstract level, since we must predict how the resulting CDFG will be translated to hardware. Second, we wish to give preference to large, complex networks in order to reduce interconnect and simplify routing. Finally, reducing the nets effectively reduces the area as well.

```
 SelectTemplateMatches(E, T, P, costfunc)
 1  changes = true
 2  while( changes )
 3    changes = false
 4    best_expr = GetBestCostExpr(E, costfunc)
 5    if best_expr != NULL then
 6      nodesets = E[best_expr]
 7      remove best_expr from E
 8      AddTemplate(nodesets, T, P)
 9      RemoveIntersectingSets(nodesets, E, P)
10      PruneUnmatchedSets(E, P)
11      delete nodesets
12      changes = true
13  PruneUnmatchedSets(T, P)
```

Figure 7.9. Pseudo-code for template selection.

7.2.6 Building Template Structures

To ensure that a template performs the same function for all isomorphic DAG structures, it must have a tree structure, as shown in Figure 7.2. The disadvantage of this approach is that it may result in an increase in area. However, this increase is bounded because it is possible to prune a template if it is deemed too costly to implement for the

number of available matches. The template tree structure can be constructed from any of the matched nodesets by decomposing the DAG into a tree using a post-order traversal, beginning at the root node. For each template match, we identify the nodes with edges to external operations, adding them to the fanout list. Note that it may be necessary to replicate DAG input edges to split nodes in the expanded tree structure. Figure 7.10 illustrates the template tree structure for the DAG in Figure 7.5 with replicated input edges and multiple fanouts. The subtract operation has been replicated during conversion from DAG to tree. The nodeset is extracted into a separate CDFG, and replaced with a single call node to the function *Template1(B,A,B,A,A,10).*

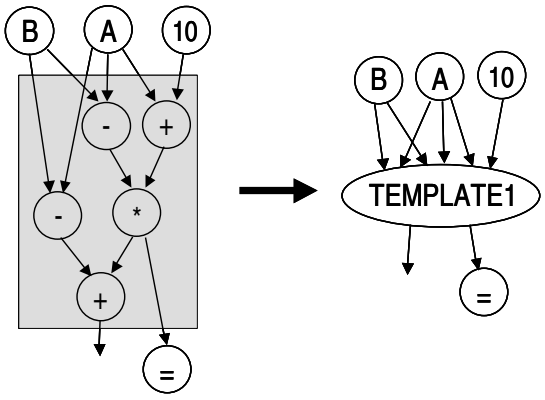


Figure 7.10. Generated template for the DAG in Figure 7.5.

7.2.7 Resource Sharing Algorithm

In this section we present a *Resource Sharing* algorithm, shown in Figure 7.11, comprised of the techniques described in this chapter. The procedure takes as input a graph *G*, a *look-ahead* value for generating the table of template expressions, and a

backtracking value. As previously described, each chosen template match has a direct impact on future template selections. Thus, the following question arises: if we had chosen the second, third, or fourth best template expression first, how would this have affected the future results?

```
   ResourceSharing(G, look_ahead, backtrack)
 1   best_T = NULL
 2   best_P = NULL
 3   best_cost = 0
 4   changes = true
 5   while ( changes ) do
 6      changes = false
 7      E maps exprs to nodeset lists
 8      T maps templates to nodeset lists
 9      P maps nodes to their parent nodeset
10      GenerateExprTable(G, E, P, look_ahead)
11      for i = 0 to min(backtrack+1, E.size) do
12        SelectTemplateMatches(E, T, P, i)
13        for each template t in T do
14          template_cost += GetCost(t,costfunc)
15        if template_cost > best_cost then
16          best_cost = template_cost
17          best_T = T
18          best_P = P
19          changes = true
20    BuildTemplates(G, best_T, best_P)
```

Figure 7.11. Pseudo-code for resource sharing.

Backtracking prevents the algorithm from becoming trapped in suboptimal local minima as a result of ill-fated decisions. However, deep backtracking can dramatically increase the CPU time required for template extraction. Therefore, we have evaluated the relationship between look-ahead depth and gains in solution quality in order to determine the point of diminishing return. By selecting the i^{th} best template expression

first, we can evaluate how well the solution compares with optimality. For instance, with a backtracking value of 1, the best expression is chosen in the first iteration and second-best expression is chosen in the second iteration. Similarly, setting the backtracking value to infinity (∞), although it is really bounded by the size of the expression table *E*, will evaluate the cost of implementing each template match first. In each iteration of the backtracking loop, we compare the cost of the resulting template table *T* to the best cost and update the values accordingly. Ultimately, the best table of templates is built.

7.3 Experimental Results

This section reports results on ten DSP benchmarks using the proposed resource sharing algorithm. The CDFGs were generated using the FREEDOM compiler, and were unrolled several times to increase the design complexity. Table 7.1 presents the number of templates generated and the maximum template size for varying look-ahead and backtracking depths. Table 7.2 presents the number and percentage resources reduced. The percentage resource reduction is the cost of the operations removed, less that required for implementing the templates. Table 7.3 presents the total run time for the resource sharing with varying look-ahead and backtracking depth.

In examining the template sizes and resource reduction for different look-ahead depths, there are apparent ambiguities in the results. These variations are likely due to the order in which templates are grown, which effects future template growth.

Additionally, the templates did not include memory operations, which limited the maximum allowable coverage. It is interesting to note the number of templates generated decreased as the look-ahead depth increased. This is expected, since larger templates are discovered initially. However, we found that increasing the look-ahead depth caused an exponential growth in the size of the hash values, requiring $O(2^d)$ time for string matching. This explains the exponential increase in execution times with increased look-ahead depth.

Overall, the results show quality gains saturate at look-ahead of approximately 3, with reduction in resource usage ranging from 40-80%. The last set in each table shows results for a backtracking depth of 10 to quantify the optimality of the algorithm. The results show that reduction in resource usage only varied within a 5% margin, indicating that our resource sharing algorithm produces efficient results, even using small, incremental template growth. Consequently, small, incremental template growth is preferable for resource sharing to obtain near-optimal results in reduced CPU time.

Interestingly, when applying this resource sharing technique to a hardware implementation in an RTL model, the multiplexer and glue logic inserted by the logic synthesis tool for sharing the hardware resulted in poor quality gains. In fact, the added glue logic cost more than the overall benefit of removing the extraneous hardware, thus counteracting the affect of resource sharing. However, one can argue that this is the fault of the logic synthesis tool. In order to properly test this resource sharing algorithm in a hardware implementation, the FREEDOM compiler would need to be adapted to directly

translate the CDFGs to structural hardware models (complete netlist of gates) for the target architecture instead of RTL models to be synthesized by third party tools.

Table 7.1. Number of templates generated and maximum template sizes for varying look-ahead and backtracking depths.

	LA = 1		LA = 3		LA = 5		LA = 7		LA = INF		BT = 10 LA = INF	
Benchmark	# Tmpls	Max Size	# Tmpls	Max Size	# Tmpls	Max Size	# Tmpls	Max Size	# Tmpls	Max Size	# Tmpls	Max Size
dot_prod	5	24	4	18	4	18	4	18	4	18	4	18
iir	9	18	6	48	6	48	6	48	6	48	6	48
matmul_32	10	24	9	18	6	18	6	18	6	18	9	20
gcd	7	5	6	5	5	5	5	5	5	5	5	5
diffeq	18	9	8	93	9	10	9	10	9	10	9	10
ellip	4	3	2	3	2	3	2	3	2	3	2	3
laplace	9	5	7	5	7	5	7	5	7	5	7	5
fir16tap	9	24	7	18	5	18	5	18	5	18	5	18
fir_cmplx	15	21	12	37	8	37	8	37	8	37	12	37
sobel	13	8	11	8	11	13	11	13	10	13	11	13

Table 7.2. Number and percentage resources reduced with varying look-ahead and backtracking depth.

Benchmark	LA = 1		LA = 3		LA = 5		LA = 7		LA = INF		BT = 10 LA = INF	
dot_prod	202	69.2%	220	75.3%	220	75.3%	220	75.3%	220	75.3%	220	75.3%
iir	531	78.4%	520	76.8%	520	76.8%	520	76.8%	520	76.8%	520	76.8%
matmul_32	219	61.3%	231	64.7%	214	59.9%	214	59.9%	214	59.9%	220	61.6%
gcd	51	43.6%	51	43.6%	48	41.0%	48	41.0%	48	41.0%	48	41.0%
diffeq	161	55.5%	133	45.9%	146	50.3%	146	50.3%	146	50.3%	146	50.3%
ellip	104	64.6%	101	62.7%	101	62.7%	101	62.7%	101	62.7%	101	62.7%
laplace	229	67.4%	215	63.2%	215	63.2%	215	63.2%	215	63.2%	215	63.2%
fir16tap	197	61.8%	215	67.4%	208	65.2%	208	65.2%	208	65.2%	208	65.2%
fir_cmplx	514	71.3%	474	65.7%	500	69.3%	500	69.3%	500	69.3%	518	71.8%
sobel	894	79.2%	870	77.1%	837	74.1%	868	76.9%	846	74.9%	844	74.8%

Table 7.3. Timing results in seconds for resource sharing with varying look-ahead and backtracking depth.

Benchmark	LA = 1	LA = 3	LA = 5	LA = 7	LA = INF	BT = 10 LA = INF
dot_prod	3.6	4.8	5.9	9.0	23.7	31.5
iir	19.5	29.8	40.4	47.6	70.8	120.6
matmul_32	4.2	7.1	8.7	11.3	20.7	36.9
gcd	0.8	0.8	0.8	0.8	0.8	2.1
diffeq	4.4	7.3	9.0	17.3	23.6	31.4
ellip	1.6	1.6	1.8	1.9	1.9	3.2
laplace	6.5	8.4	11.9	15.2	20.8	31.3
fir16tap	4.9	6.3	7.3	9.7	17.5	29.3
fir_cmplx	25.1	36.8	45.3	65.7	156.6	231.2
sobel	38.3	49.4	54.0	54.9	48.0	164.9

7.4 Summary

This chapter presented a regularity extraction algorithm for resource sharing in CDFGs. We use a heuristic to dynamically grow templates based on a cost function and the frequency of reoccurring patterns, while applying backtracking and varying look-ahead depth to evaluate the trade off between solution quality and run time. Experimental results on ten benchmarks indicate that small, incremental template growth is preferable for resource sharing to obtain near-optimal results in reduced CPU time.

Chapter 8

Hardware-Software Partitioning of Software Binaries

Partitioning plays an increasingly important role in the design process of hardware/software co-designs of embedded systems. The decisions of how and where to partition a design are crucial in obtaining optimal performance trade-offs. Often times, synthesis and simulation tools cannot handle the complexity of the entire system under development, or designers may want to concentrate on critical parts of a system to speed-up the design cycle. Thus, the present state of design technology often requires a partitioning of the system.

Generally, designers will profile functions in an application to determine the bottlenecks in the design, or locate the portions of code that are the most computationally intensive portions in the design. These structures, which generally consist of entire procedures or loop structures, are usually selected to move to hardware in order to obtain performance speedups.

Partitioning is usually performed at high levels of abstraction since it is usually easier to analyze the design structure. Partitioning software binaries poses a more complicated problem since the structures are not often identifiable, or they may be highly pipelined and cannot be easily partitioned. Sometimes the loop structures are too small and larger structures are required, or perhaps the designer may wish to move large blocks of sequential code to hardware instead.

This chapter presents a general scheme for hardware/software partitioning of software binaries, in which structures are extracted from procedures at different levels in the design hierarchy. The structures are self-contained and can be optimized independently, moved to hardware as a stand-alone procedure, or implemented as part of a hardware/software co-design.

8.1 Related Work

There are many important aspects to design partitioning, some of which has been discussed by Johannes [31]. Partitioning can be implemented on all different levels of granularity, generally at the functional or structural levels, and at different stages of a high-level synthesis process. In the early stages of design, partitioning can be more complicated as the critical decisions for optimal performance tradeoffs are often based on incomplete knowledge. Many researches have implemented partitioning at the FSM level. Feske et al. [23] presented an approach for FSM partitioning and state encoding

targeting FPGA architectures. Vahid et al. [61][62] showed that partitioning a design at the functional level can produce better I/O and size constraints than at the structural netlist level. Furthermore, partitioning a design before synthesis can yield an order of magnitude improvement in synthesis runtimes.

Other work has shown that partitioning a design into smaller sections can be used to decrease power consumption. Chung et al. [9] described work on partitioning of behavioral-level descriptions prior to scheduling and allocation, such that each partition is controlled by an independent gated clock, which can be turned off to reduce power consumption for the entire functional block. The optimality of the partition is defined in terms of area and power under a given global timing constraint, based on high-level estimations. Hwang et al. [29] proposed a power reduction technique by partitioning an FSM, and then shutting down the inactive processes and datapaths to eliminate unnecessary switching activity. Venkataraman et al. [64] described the GALLOP tool for FSM partitioning. Using a genetic algorithm they perform state assignment and partitioning simultaneously to reduce power consumption.

Partitioning software binaries is often more complicated than partitioning a high-level application due to the lack of functional boundary information present in the design. Software binaries generally require coarse-grain partitioning first at the procedural level, or procedure extraction. Cifuentes and Simon [13] have reported algorithms for identifying function calls from assembly programs using predefined procedure call interfaces. They further described a procedure abstraction language that

can be used to specify the calling conventions for different architectures, and discuss how data flow analysis can be used to identify function arguments. Mittal et al. [43] described a method of automatically extracting function bodies from linked software binaries using procedure-calling conventions along with limited control and data flow information. This work was incorporated as part of the FREEDOM compiler.

There has been very little work reported on structural partitioning of software binaries at a fine-grain level. Most work reported focused on simple loop structures or the entire function [10][11][12][60][59]. Our approach allows for both fine and course grain partitioning of software binaries at the procedural and structural levels. The partitioned kernels of the design can then be optimized individually, and selectively moved to hardware as a stand-alone procedure or as a hardware/software co-design.

8.2 Structural Extraction

This section introduces the concept of *structural extraction* as a means of partitioning software binaries for a hardware/software co-design. Structural extraction was implemented in the FREEDOM compiler as a means of extracting fine-grain and course-grain structures from MST procedures at different hierarchical levels. This includes loops, if-then-else blocks, etc. The designer is able to extract the identified structures using the GUI interface. The extracted structures are treated as independent procedures, and therefore can be easily optimized and moved to hardware. Input and

output ports are automatically generated during the CDFG optimizations as described in Section 5.2.1.

8.2.1 Discovering Extractable Structures

Structural extraction requires one to first identify those structures that are feasible for partitioning. A manual approach is often used to profile the design for computational bottlenecks. Alternatively, heuristics may be used to determine the best sections of code to move to hardware, such as those loops with the large code size. In our approach, we leave the structural selection to the designer. The objective here is to identify structures that can be partitioned within a maximum inter-procedural cut-size for data dependencies. As described earlier (Section 5.2.1), we limit the cut-size of the I/O interface to the size of the number of physical registers in the source general-purpose processor architecture. Consequently, we do not allow virtual registers to cross inter-procedural boundaries.

Figure 8.1 describes the procedure *Discover_Structures()* for discovering structures to be extracted, whose inputs consist of an MST procedure, *P*, and a Boolean mapping, *M*, of each structure to signify if it is extractable. The procedure begins by generating a linearized CFG from the MST procedure, as described in Chapter 4. Using the methods described in sections 5.1.1 and 5.1.2, reaching definitions are generated for data flow analysis, followed by structural analysis to generate a minimized structural graph. *Discover_Extractable_Structures(),* described in Figure 8.2, is recursively called

on the each structure in the hierarchical tree to determine if it may be extracted. A structure may only be extracted if no virtual register has data dependencies that cross the structural boundaries. Figure 8.3 describes the procedure *Can_Extract_Structure()*, which utilizes data dependency analysis to identify such operands. The procedure iterates through the instructions in each basic block within the structure and checks that no virtual operand has data dependencies that exist outside the structure. A structure that contains inter-procedural virtual data dependencies may not be extracted independently from its parent structure.

```
  Discover_Structures( P, M )
1  CFG = Build_CFG( P )
2  Generate_Reaching_Definitions( CFG )
3  SG = Structural_Analysis( CFG )
4  struct_list = SG->GetStructures()
5  Discover_Extractable_Structures( CFG, M, struct_list )
6  return SG
```

Figure 8.1. Procedure for discovering structures for extraction.

```
  Discover_Extractable_Structures( CFG, M, struct_list )
1  for each structure s in struct_list do
2    if ( Can_Extract_Structure(CFG, s) ) then
3      M[ s ] = true
4    else
5      M[ s ] = false
6    Discover_Extractable_Structures(CFG, M, s->GetStructures())
```

Figure 8.2. Recursive procedure for identifying extractable structures.

```
  Can_Extract_Structure( CFG, s )
1  for each block b in s do
2    for each instruction i in block b do
3      for each virtual operand register r in i do
4         if r is has a definition outside of s then
5           return false
6         else if r has a use outside of s then
7           return false
8  return true
```

Figure 8.3. Procedure for determining if a structure can be extracted.

An analysis of the algorithm in Figure 8.1 requires *O(n)* for generating the CFG [74] in line 1, as described in Chapter 4. Generating the reaching definitions (DU-chains and UD-chains) [44] in line 2 requires $O(n^2)$ in the number of instructions. In line 3, structural analysis requires *O(b)* time in the number of blocks using the graph minimization technique with depth first search [44]. *Discover_Extractable_Structures()* in line 5 (see Figure 8.2) recursively iterates through the structure hierarchy in the graph. At each level in the hierarchy, it iterates through all instructions within the structure to find any definition that crosses structural boundaries, as described in the function *Can_Extract_Structure()* in Figure 8.3. Searching the DU-chains for definitions that cross structure boundaries has a worst case time complexity of *O(n)* for each structure. A worst case scenario results when each structure is composed of two structures in which one has a single instruction, requiring $O(n^2)$ time complexity to traverse the entire structure hierarchy. Consequently, the complete algorithm has a worst case time complexity of $O(n^2)$.

8.2.2 Selecting Structures for Extraction

Once the extractable structures in the procedure are identified, they may be selected for extraction manually using the GUI interface. A dialog box presents a tree model of the structural hierarchy in the procedure that can be explored. A structure (and its children) may be selected for extraction by clicking the checkbox next to it. However, the checkbox is disabled for structures that may not be extracted independently of its parent structure. While exploring the structures in the procedure, the window on right side presents useful information about the highlighted structure, such as the type (loop, if-then-else, etc.) and the number and percentage of operations within.

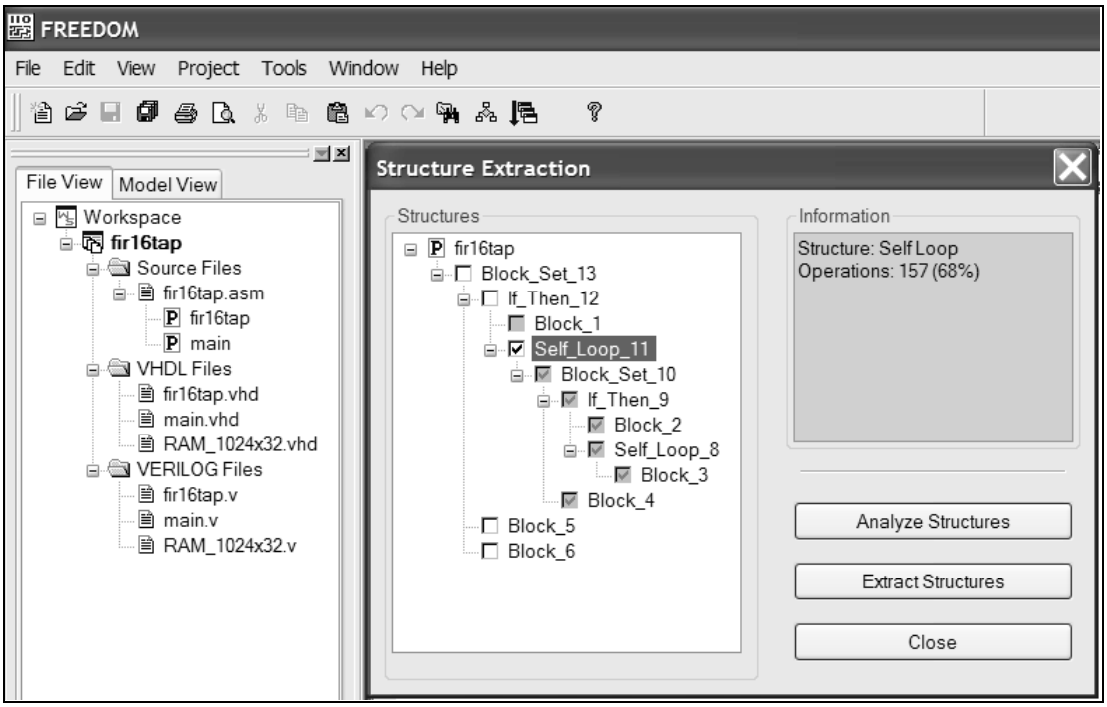


Figure 8.4. GUI interface for selecting structures for extraction.

8.2.3 Extracting Structures

Figure 8.5 describes a recursive procedure for extracting structures from a procedure. It takes as input the procedure *P*, a Boolean mapping *M* for each structure to be extracted, and *struct_list*, the current list of structures at each hierarchical level. If the map *M* returns *true* for a structure, it is extracted along with all its children. Otherwise, the function is recursively called on its children structures. Structures are extracted by replacing the basic blocks and instructions that lie within them with a call to a new procedure. The algorithm has a worst case run time complexity of *O(s)* in the number of structures.

```
  Extract_Structures( P, M, struct_list )
1  for each structure s in struct_list do
2    if ( M[ s ] == true ) then
3      extract structure s from procedure P
4    else
5      Extract_Structures( P, M, s->GetStructures() )
```

Figure 8.5. Recursive procedure fore extracting structures.

In Figure 8.6, the outer loop of the *fir16tap* procedure was extracted; instructions with PC values 0x0704 to 0x0770 have been replaced with the call instruction to the procedure *fir16tap_struct_0*. Figure 8.7 describes the Verilog code for calling the *fir16tap_struct_0* procedure from its parent in a FSM. The inputs and reset are set in the first state, and then it waits in the second state until the process is complete. In the third state the output values from the procedure are read.

```
22.00000000  0x00000700  L1: NOP(0,u') 0x00000000(s'32)
22.00100000  0x00000700  CALL(0,u') fir16tap_struct_0
23.00000000  0x00000700  GOTO(0,u') L4
64.00000000  0x00000774  L4: NOP(0,u') 0x00000000(s'32)
```

Figure 8.6. Extracted MST instructions replaced with a procedure call.

```
FIR16TAP_FSM_PROCESS_STATE_1 : begin
  fir16tap_struct_0_instance_0_reset <= 1'b1;
  fir16tap_struct_0_instance_0_A10_IN <= SR3;
  fir16tap_struct_0_instance_0_A11_IN <= SR5;
  fir16tap_struct_0_instance_0_A2_IN <= SR9;
  fir16tap_struct_0_instance_0_A3_IN <= SR11;
  fir16tap_struct_0_instance_0_A4_IN <= SR13;
  fir16tap_struct_0_instance_0_A5_IN <= SR14;
  fir16tap_struct_0_instance_0_A6_IN <= SR17;
  fir16tap_struct_0_instance_0_A7_IN <= SR18;
  fir16tap_struct_0_instance_0_A8_IN <= SR20;
  fir16tap_struct_0_instance_0_A9_IN <= SR21;
  fir16tap_ctrl <= FIR16TAP_FSM_PROCESS_STATE_2;
end

FIR16TAP_FSM_PROCESS_STATE_2 : begin
  fir16tap_struct_0_instance_0_reset <= 1'b0;
  if (fir16tap_struct_0_instance_0_done == 1'b1) begin
    fir16tap_ctrl <= FIR16TAP_FSM_PROCESS_STATE_3;
  end
  else begin
    fir16tap_ctrl <= FIR16TAP_FSM_PROCESS_STATE_2;
  end
end

FIR16TAP_FSM_PROCESS_STATE_3 : begin
  SR0 <= fir16tap_struct_0_instance_0_A0_OUT;
  SR2 <= fir16tap_struct_0_instance_0_A1_OUT;
  SR9 <= fir16tap_struct_0_instance_0_A2_OUT;
  SR14 <= fir16tap_struct_0_instance_0_A3_OUT;
  SR15 <= fir16tap_struct_0_instance_0_A5_OUT;
  SR18 <= fir16tap_struct_0_instance_0_A7_OUT;
  SR21 <= fir16tap_struct_0_instance_0_A9_OUT;
  fir16tap_ctrl <= FIR16TAP_FSM_PROCESS_STATE_4;
end
```

Figure 8.7. Verilog code for calling the extracted procedure in a FSM.

8.3 Summary

This chapter presented a fine-grain hardware/software partitioning scheme for software binaries at the structural level, such as loop bodies. While discovering structures for extraction, we minimize the cut size of the graph by only allowing partitioning of a structure that does not have any virtual registers with dependencies that cross the structure boundaries. Rather, only physical registers may have dependencies across structural boundaries. The extracted structures are therefore self-contained and can be optimized independently, or moved to hardware as a stand-alone procedure or in a hardware/software co-design.

Chapter 9

A Case Study: MPEG-4

This chapter presents a case study on the MPEG-4 (Motion Picture Encoding Group) video decoder, which was used as a means of quantifying the work presented in this dissertation. The MPEG-4 video decoder was originally available in C code and compiled for the TI C6211 DSP architecture. The software binary was translated to hardware using the FREEDOM compiler, targeting the Xilinx Virtex II FPGA. The following sections provide an overview of the MPEG-4 software architecture and a comparison of implementation results for the general-purpose processor and the FPGA architecture.

9.1 Overview of the MPEG-4 Decoder

The MPEG-4 standards began development in 1995, and is now composed of 16 parts, shown in Table 9.1. The visual coding standard is detailed in part 2. In part 10, an advanced video codec (AVC) is described, which was added as a means of backwards

compatibility for the older MPEG standards. The implementation described here is only concerned with the standard detailed in part 2. Each part in the standard is subdivided into profiles, which define different technology criteria. For instance, the *simple* profile operates on rectangular *I-frames* and *P-frames*, while the *core* profile also allows for *B-frames*. Within each profile are levels that define the scope or magnitude of the parameters within the profile, such as bit rate and typical frame size. The software architecture for the MPEG-4 video decoder used in this case study was implemented using the *simple* profile. The following subsections describe the software model for the decoder, as is illustrated in Figure 9.1.

Table 9.1. MPEG-4 standard.

Part	Description
1	Systems
2	Visual
3	Audio
4	Conformance testing
5	Reference software
6	DMIF
7	Optimized software of MPEG-4 tools
8	MPEG-4 over IP networks
9	Reference hardware description
10	AVC
11	Scene description and application engine
12	ISO media file format
13	IP management and protection extensions
14	MP4 file format
15	AVC file format
16	AFX

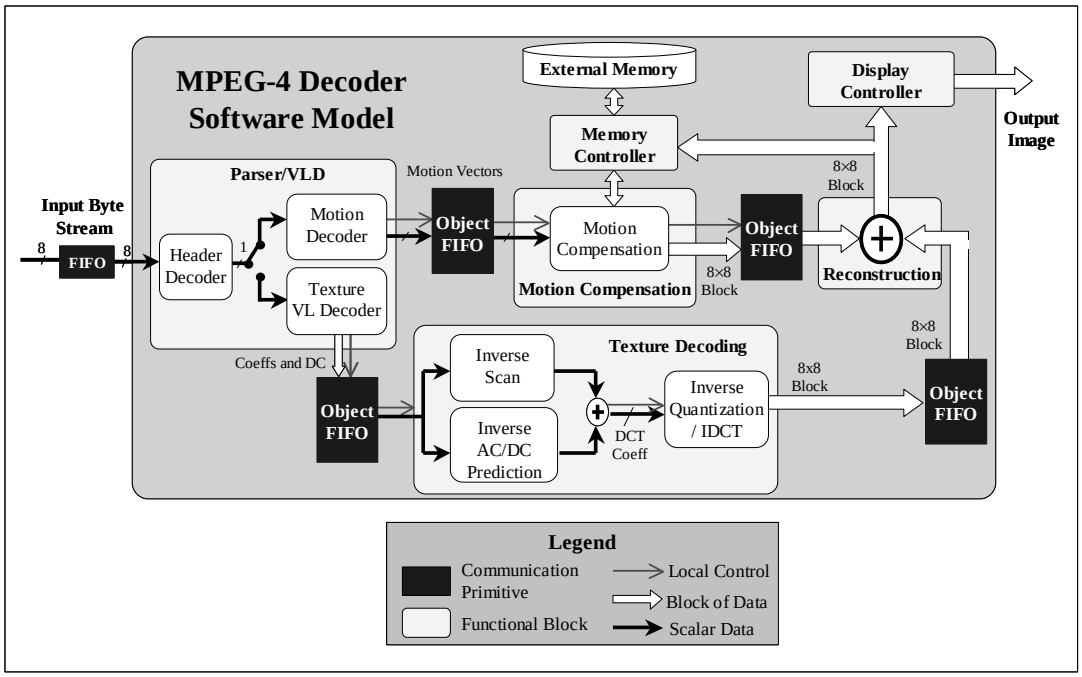


Figure 9.1. Software model for the MPEG-4 decoder.

9.1.1 Parser

The *parser* module is the front-end to the MPEG-4 decoder, in which the input byte stream is partitioned and decoded. The parser decodes visual object layer (VOL) headers for multi-layer streams and video object plane (VOP) headers for different frame types. The simple profile supports VOP headers made up of *I-frames*, which are intra-coded frames that do not use motion estimation, and *P-frames*, which are predictive-coded frame that use motion estimation based on the previously encoded frame. The frame data is organized into macroblocks. The *I-frames* macroblocks contain the texture data, while the *P-frame* macroblocks are composed of the motion vectors. A variable length decoder is used to decode both texture data and motion vectors.

9.1.2 Texture Decoding

The *texture decoding* module performs complete decoding of *I-frames* for intra-coded pictures. The texture decoding works on a block level (8x8) and is made of zigzag encoding, inverse quantization, AC/DC prediction, and Inverse Discrete Cosine Transform (IDCT). The software architecture for the textured decoding module is illustrated in Figure 9.2. The values in parentheses are the total number of function calls in the procedure and the edge values are the number of calls to individual procedures.

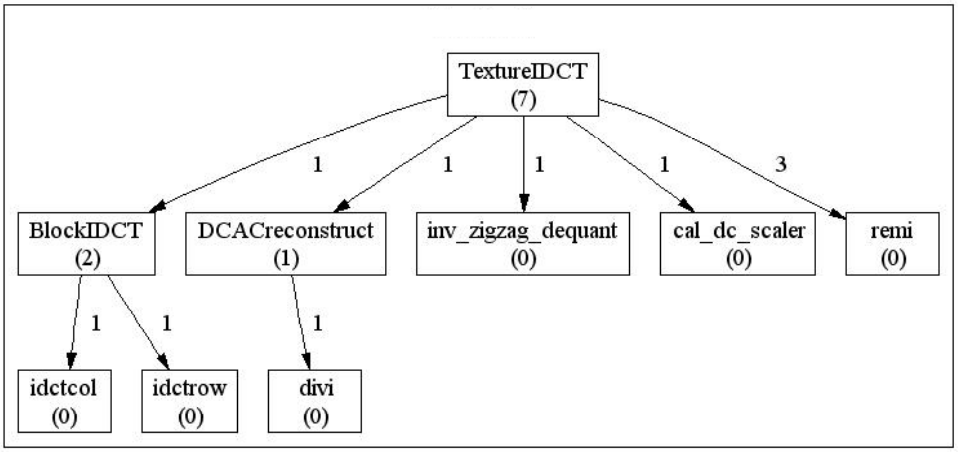


Figure 9.2. Software implementation for the texture decoding module.

9.1.3 Motion Compensation

The *motion compensation* module operates on macroblocks of *P-frames*. This module reconstructs frames based on motion vectors, which are added to the reference frame that is stored in the external memory. The reference frame is streamed into a buffer via the memory controller, and operations are performed on a block level

(8x8). The software architecture for the motion compensation module is illustrated in Figure 9.3.

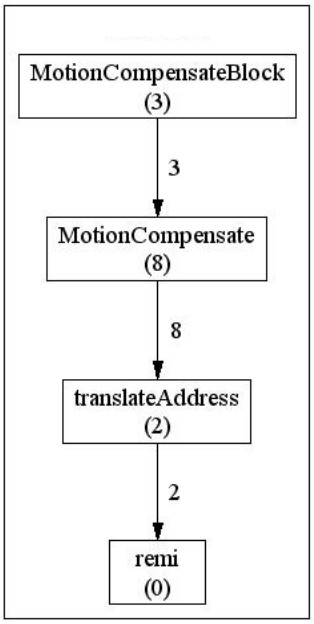


Figure 9.3. Software implementation for the motion compensation module.

9.1.4 Reconstruction

The *reconstruction* module combines the movement information carried by the decoded motion vector with the reference frame produced by the texture decoder. The resulting image is stored in the external memory and streamed out to video for viewing.

9.1.5 Memory Controller

The *memory controller* module provides the motion compensation module access to the external memory where the reference frame is stored. Additionally, the

reconstructed frame is stored to the external memory via the memory controller. The software architecture for the memory controller module is illustrated in Figure 9.4.

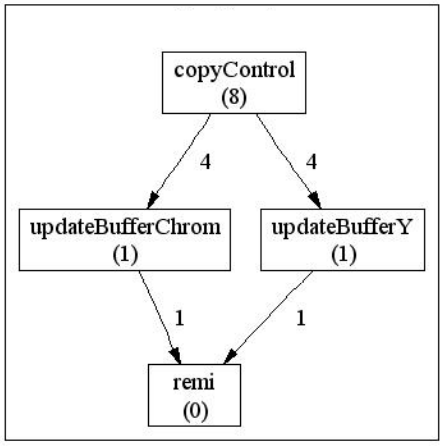


Figure 9.4. Software implementation for the memory controller module.

9.1.6 Display Controller

The *display controller* module performs YUV to RBG conversion on the reconstructed frame. The RGB frame data is then streamed out to video.

9.2 Experimental Results

This section reports comparison of results for several modules in the MPEG-4 decoder software that were implemented on a general-purpose DSP and then translated and optimized for an FPGA architecture using the FREEDOM compiler. The MPEG-4 decoder modules chosen were the *texture decoder*, *motion compensation*, *reconstruction*, and *memory controller*. The *parser* and *display controller* modules were left out because

their software implementations were highly integrated with file I/O functions that posed a problem in the hardware/software translation.

Table 9.2 compares the performance of the MPEG-4 decoder modules on the TI C6211 DSP and Xilinx Virtex II FPGA platforms. The first function listed for each module is the top-level function. Results indicate speedups between 14-67x in terms of cycles and 6-22x in terms of time for the FPGA implementation over that of the DSP.

Table 9.2. Comparison of MPEG-4 decoder modules on DSP and FPGA platforms.

	TI C6211 DSP			Xilinx Virtex II FPGA					
Function	**Avg. Cycles**	**Freq. (MHz)**	**Time (µs)**	**Avg. Cycles**	**Freq. (MHz)**	**Time (µs)**	**Area (LUTs)**	**Speedup (cycles)**	**Speedup (time)**
Texture Decoder									
TextureIDCT	275156	300.0	917.19	4088	100.1	40.84	36029	**67.31**	**22.46**
DCACreconstruct	4860	300.0	16.20	132	100.1	1.32	*	36.82	12.29
cal_dc_scaler	71	300.0	0.24	12	100.1	0.12	*	5.92	1.97
inv_zigzag_dequant	55198	300.0	183.99	1380	100.1	13.79	*	40.00	13.35
BlockIDCT	200654	300.0	668.85	2203	100.1	22.01	*	91.08	30.39
Idctcol	13980	300.0	46.60	224	100.1	2.24	*	62.41	20.82
Idctrow	10915	300.0	36.38	48	100.1	0.48	*	227.40	75.87
Divi	30	300.0	0.10	47	100.1	0.47	*	0.64	0.21
Remi	22	300.0	0.07	46	100.1	0.46	*	0.48	0.16
Motion Compensate									
MotionCompBlock	73447	300.0	244.82	5266	124.5	42.30	9851	**13.95**	**5.79**
MotionCompensate	71658	300.0	238.86	5245	124.5	42.13	*	13.66	5.67
translateAddress	854	300.0	2.85	74	124.5	0.59	*	11.54	4.79
Remi	14	300.0	0.05	46	124.5	0.37	*	0.30	0.13
Memory Controller									
copyControl	202294	300.0	674.31	5248	107.6	48.77	9675	**38.55**	**13.83**
updateBufferY	65132	300.0	217.11	1705	107.6	15.85	*	38.20	13.70
updateBufferChrom	33588	300.0	111.96	887	107.6	8.24	*	37.87	13.58
Remi	20	300.0	0.07	46	107.6	0.43	*	0.43	0.16
Reconstruction									
TextureUpdate	34888	300.0	116.29	1348	134.2	10.04	5781	**25.88**	**11.58**

9.3 Summary

This chapter presented a case study on the MPEG-4 video decoder in order to quantify the work presented in this dissertation. An overview of the MPEG-4 software architecture was provided. Comparison of implementation results for the TI C6211 DSP and the Xilinx Virtex II FPGA architectures show speedups between 14-67x in terms of cycles and 6-22x in terms of time for the FPGA implementation over that of the DSP. These results indicate the successfulness of the FREEDOM compiler in translating and optimizing even large software binary applications for FPGA architectures to obtain greater performances.

Chapter 10

Conclusions and Future Work

Recent advances in embedded communications and control systems are pushing the computational limits of DSP applications. Often times, the performance of these complex applications is impeded due to limitations in the computational bandwidth of the processor. The conventional way to address the computational bottleneck has been to migrate sections of the application or its entirety to hardware as part of a hardware/software co-design system. However, the problem is much more complicated when implementing a hardware/software co-design from software binaries. The challenge in translating instructions from a fixed processor architecture to hardware is in undoing all the architecture-specific optimizations, and then exploiting the fine-grain parallelism in the design using a much larger number of functional units, embedded multipliers, registers and on-chip embedded memories.

10.1 Summary of Contributions

The contribution of this work is in developing a methodology for translating software binaries of general-purpose processors to hardware descriptions for FPGAs, and to evaluate the efficiency of such methods as compared to the performance of the general-purpose processor. Towards this effort, we have developed the FREEDOM compiler, which automatically translates DSP software binaries to hardware descriptions for FPGAs. The multiple levels of the compiler architecture were discussed in detail.

We have presented our methodology for translating scheduled and pipelined software binaries to hardware and described an array of optimizations that were implemented in the FREEDOM compiler. The effects of these optimizations on performance were evaluated in terms of clock cycle count, frequency, and area. Through new techniques for balanced scheduling and operation chaining we have shown even greater improvements in performance.

We employed methods of fine-grain and course-grain extractions of instructions to further improve design feasibility. Our resource sharing technique uses a method known as regularity extraction to generate templates of reoccurring patters in a design. The reoccurring patterns are then extracted and replaced with template calls in order to reduce resource utilization. Our structural extraction technique identifies different types of structures in a design, such as loops and if-then-else blocks. These structures can be

extracted as separate procedures in functional hierarchy, allowing for partitioning of a design or implementation of a hardware/software co-design.

These concepts outlined in this work were tested in a case study for an MPEG-4 decoder in which we compare performances between a DSP and an FPGA implementation. Results indicate speedups between 14-67x in terms of cycles and 6-22x in terms of time for the FPGA implementation over that of the DSP.

10.2 Comparison with High-Level Synthesis Performances

One may speculate that there is a loss of information when translating software binaries to hardware rather than from a high-level language. Alternatively, one might assume in order to obtain quality results the software binaries would need to be decompiled into a high-level language, such as C, and then use a behavioral synthesis tool to generate hardware.

In order to test this hypothesis we took ten example benchmarks in C, and used the PACT compiler [32] to generate a hardware implementation for the Xilinx Virtex II FPGA. Table 10.1 shows the best performance results for the PACT compiler as compared to the best performance results for our FREEDOM compiler in terms of area, frequency and cycles. It is clear that the results of the two approaches are comparable. This validates our claim that one does not need to decompile software binaries or assembly code to a high-level language in order to obtain quality results. Rather,

assembly and binary codes may be used as an intermediate language from any high-level language to generate an efficient hardware implementation. Consequently, results for the FREEDOM compiler show improvements of 3-22x in terms of clock cycles and 2-10x in terms of time over that of the DSP implementation.

Table 10.1. Performance comparison between the TI C6211 DSP and the PACT and FREEDOM compiler implementations on the Xilinx Virtex II FPGA.

	DSP			PACT				FREEDOM			
Benchmark	Cycles	Freq	Exec Time	Cycles	Area	Freq	Exec Time	Cycles	Area	Freq	Exec Time
dot_prod	12516	300.0	**41.7**	3357	2447	69.2	**48.5**	1354	1371	130.1	**10.4**
iir	22987	300.0	**76.6**	3010	5873	98.4	**30.6**	4255	2837	124.2	**34.3**
matmul_32	1799064	300.0	**5996.9**	277703	2155	70.2	**3955.9**	134502	1832	124.9	**1076.9**
gcd	268	300.0	**0.9**	48	322	158.2	**0.3**	76	283	181.6	**0.4**
diffeq	2318	300.0	**7.7**	79	1396	69.4	**1.1**	104	1259	141.9	**0.7**
ellip	335	300.0	**1.1**	43	1222	180.0	**0.2**	57	1649	147.7	**0.4**
laplace	74673	300.0	**248.9**	8467	10000	78.2	**108.3**	5919	3222	130.7	**45.3**
fir16tap	113285	300.0	**377.6**	115209	547	69.7	**1652.9**	17600	1458	127.4	**138.1**
fir_cmplx	72856	300.0	**242.9**	8499	6083	57.9	**146.8**	3316	3319	101.1	**32.8**
sobel	127495	300.0	**425.0**	81418	7873	57.6	**1413.5**	24501	4458	128.3	**191.0**

10.3 Future Work

The work presented in this dissertation has show great promise in high-level synthesis of software binaries. More optimizations are necessary to improve performance even further. For instance, most of the optimizations currently implemented in the compiler are local to a single basic block. Optimizations, such as inter-procedural constant propagation could significantly improve design performance by propagating

constant inputs to procedures at compile time. This would allow other optimizations to effectively reduce the design complexity and allow for better aliasing. Additionally, FPGA designs are capable of larger throughput, but are often hindered due to the significant number of memory operations that originate from the source binaries. It would be of great advantage to optimize or further remove extraneous memory operations in order to allow more parallelism and throughput in the hardware implementations.

Scheduling plays a particularly important role in high-level synthesis. Our new balanced scheduling and operation chaining methods are quite effective in increasing the overall frequency of the design. However, the current scheduling routines do not consider power or thermal effects for VLSI circuits. This is a crucial area of research that needs to be evaluated and integrated into the FREEDOM compiler when generating FPGA designs. Software pipelining is another important scheduling optimization that needs to be addressed which will have a significant impact on throughput performance.

The FREEDOM compiler currently assumes a FIFO buffer for communication in a hardware/software co-design. It does not consider other communication methods. The architecture description language can be used as a means of interface co-synthesis for such applications. There is also a need for investigating the implementation of streaming interfaces for hardware designs.

References

[1] W. Amme, P. Braun, F. Thomasset, and E. Zehendner, "Data Dependence Analysis of Assembly Code," in *International Journal of Parallel Programming*, vol. 28, issue 5, 2000.

[2] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," in *Proceedings for Conference on Programming Language Design and Implementation (PLDI)*, pp 1-12, June 2000.

[3] P. Banerjee, M. Haldar, A. Nayak, V. Kim, V. Saxena, S. Parkes, D. Bagchi, S. Pal, N. Tripathi, D. Zaretsky, R. Anderson, and J. R. Uribe, "Overview of a Compiler for Synthesizing MATLAB Programs onto FPGAs," in *IEEE Transaction on VLSI Systems*, vol. 12, no. 3, pp 312-324, March 2004.

[4] U. Banerjee, *Dependence Analysis for Supercomputers*. Kluwer Academic Publishers, Norwell, MA, 1988.

[5] O. Bringmann, C. Menn, and W. Rosenstiel, "Target Architecture Oriented High-Level Synthesis for Multi-FPGA Based Emulation," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp 326-332, Paris, France, 2000.

[6] T. Callahan, P. Chong, A. DeHon, J. Wawrzynek, "Fast Module Mapping and Placement for Datapaths in FPGAs," in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pp 123-132, Monterey, CA, 1998.

[7] D. Chen, J. Cong, and Y. Fan, "Low-power high-level synthesis for FPGA architectures," in *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pp 134-139, Seoul, Korea, 2003.

[8] A. Chowdhary, S. Kale, P. Saripella, N. Sehgal, and R. Gupta, "A General Approach for Regularity Extraction in Datapath Circuits," in *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design*, pp 332-339, San Jose, CA, Nov. 1998.

[9] K.-S. Chung, T. Kim, and C. I. Liu, “Behavioral-Level Partitioning for Low Power design in Control-Dominated Applications,” in *Proceedings of the 10th Great Lakes Symposium on VLSI*, pp 156-161, Chicago, Illinois, 2000.

[10] C. Cifuentes and K. Gough, “A Methodology for Decomposition,” in *Proceedings for XIX Conferencia Latinoamericana de Informatica*, pp 257-266, Buenos Aires, Argentina, 1993.

[11] C. Cifuentas and V. Malhotra, “Binary Translation: Static, Dynamic, Retargetable?” in *Proceedings for the International Conference On Software Maintenance (ICSM)*, pp 340-349, Monterey, CA, 1996.

[12] C. Cifuentes, D. Simon, and A. Fraboulet, “Assembly to High-Level Language Translation,” in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp 228-237, Washington, DC, 1998.

[13] C. Cifuentes and D. Simon, “Procedure Abstraction Recovery from Binary Code,” in *Proceedings of the Conference on Software Maintenance and Reengineering*, pp 55-64, 2000.

[14] K. Cooper, T. Harvey, and T. Waterman, “Building a Control-Flow Graph from Scheduled Assembly Code,” Technical Report 02-399, Department of Computer Science, Rice University, Houston, TX, 2002.

[15] G. De Micheli, D. C. Ku, F. Mailhot, and T. Truong, “The Olympus Synthesis System for Digital Design,” in *IEEE Design and Test Magazine*, pp 37–53, October 1990.

[16] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[17] G. De Micheli and R. K. Gupta, “Hardware/Software Co-Design,” in *Proceedings of the IEEE*, vol. 85, no. 3, pp 349–365, Mar. 1997.

[18] B. Decker and D. Kästner, “Reconstructing Control Flow from Predicated Assembly Code,” in *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems*, pp 81-100, Vienna, Austria, 2003.

[19] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, “Dynamic translation: The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges,” in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp 15-24, March 2003.

[20] W. E. Dougherty and D. E. Thomas, "Unifying behavioral synthesis and physical design," in *Proceedings of the 37th Conference on Design Automation*, pp 756-761, Los Angeles, CA, 2000.

[21] C. Ebeling and O. Zajicek, "Validating VLSI Circuit Layout by Wirelist Comparison," in *Proceedings of the IEEE International Conference on Computer Aided Design*, pp 172-173, 1983.

[22] R. Ernst, "Codesign of Embedded Systems: Status and Trends," in *IEEE Design & Test of Computers*, vol. 12, pp 45–54, Apr. 1998.

[23] K. Feske, S. Mulka, M. Koegst, and G. Elst, "Technology-Driven FSM Partitioning for Synthesis of Large Sequential Circuits Targeting Lookup-Table Based FPGAs," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pp 235-244, 1997.

[24] Z. Gu, J. Wang, R. Dick, and H. Zhou, "Incremental exploration of the combined physical and behavioral design space," in *Proceedings of the 42nd Annual Conference on Design Automation*, pp 208-213, San Diego, California, 2005.

[25] Y. Guo, G. Smit, H. Broersma, and P. Heysters, "A Graph Covering Algorithm for a Coarse Grain Reconfigurable System," in *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, pp 199-208, San Diego, CA, June 2003.

[26] R. Gupta and G. De Micheli, "Hardware-Software Co-Synthesis for Digital Systems," in *IEEE Design & Test of Computers*, vol. 10, pp. 29–41, Sept. 1993.

[27] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, "Dynamic and Transparent Binary Translation," in *IEEE Computer Magazine*, vol. 33, no. 3, pp 54-59, March 2000.

[28] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "A System for Synthesizing Optimized FPGA Hardware from MATLAB," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp 314-319, San Jose, CA, Nov. 2001.

[29] E. Hwang, F. Vahid, and Y.-C. Hsu, "FSMD Functional Partitioning for Low Power," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp 22-28, Munich, Germany, 1999.

[30] T. Jiang, X. Tang, and P. Banerjee, "Macro-models for high level area and power estimation on FPGAs," in *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pp 162-165, Boston, MA, Apr. 2004.

[31] F. Johannes, "Partitioning of VLSI Circuits and Systems," in *Proceedings of the 33rd Annual Conference on Design Automation*, pp 83-87, Las Vegas, NV, 1996.

[32] A. Jones, D. Bagchi, S. Pal, X. Tang, A. Choudhary, and P. Banerjee, "PACT HDL: a C compiler targeting ASICs and FPGAs with power and performance optimizations," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp 188-197, Grenoble, France, Oct. 2002.

[33] D. Kästner and S. Wilhelm, "Generic Control Flow Reconstruction from Assembly Code," in *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, vol. 37, issue 7, pp 46-55, Berlin, Germany, June 2002.

[34] R. Kastner, S. Memik, E. Bozorgzadeh, M. Sarrafzadeh, "Instruction Generation for Hybrid Reconfigurable Systems," in *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 7, issue 4, pp 605-627, Oct. 2002.

[35] D. Kerns and S. Eggers, "Balanced Scheduling: Instruction Scheduling when Memory Latency is Uncertain," in *ACM SIGPLAN Notices*, vol. 39, issue 4, pp 515-527, Apr. 2004.

[36] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," in *Proceedings of USENIX Security 2004*, pp 255-270, San Diego, CA, 2004.

[37] G. Lakshminarayana and N. Jha, "Synthesis of Power-Optimized Circuits from Hierarchal Behavioral Descriptions," in *Proceedings of the 35th Annual Conference on Design Automation*, pp 439-444, San Francisco, CA, 1998.

[38] B. Levine and H. Schmidt, "Efficient Application Representation for HASTE: Hybrid Architectures with a Single Executable," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp 101-107, Napa, CA, 2003.

[39] S. O. Memik, G. Memik, R. Jafari, E. Kursun, "Global Resource Sharing for Synthesis of Control Data Flow Graphs on FPGAs," in *Proceedings of the 40th Conference on Design Automation*, pp 604-609, Anaheim, CA, 2003.

[40] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-Software Co-Design of Embedded Reconfigurable Architectures," in *Proceedings of the 37th Conference on Design Automation*, pp 507-512, Los Angeles, CA, 2000.

[41] G. Mittal. *A Compiler Infrastructure for Compiling Assembly and Binary Programs onto FPGAS*. Ph.D. Dissertation, Northwestern University, Evanston, IL, Aug. 2004.

[42] G. Mittal, D. Zaretsky, X. Tang, and P. Banerjee, "Automatic Translation of Software Binaries onto FPGAs," in *Proceedings of the 41st Annual Conference on Design Automation*, pp 389-394, San Diego, CA, 2004.

[43] G. Mittal, D. Zaretsky, G. Memik, and P. Banerjee, "Automatic Extraction of Function Bodies from Software Binaries," in *Proceedings for the IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, Beijing, China, 2005.

[44] S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[45] R. Mukherjee, S. O. Memik, and G. Memik, "Temperature-aware resource allocation and binding in high-level synthesis," in *Proceedings of the 42nd Annual Conference on Design Automation*, pp 196-201, San Diego, CA, 2005.

[46] E. Musoll and J. Cortadella, "High-level Synthesis Techniques for Reducing the Activity of Functional Units," in *Proceedings of the International Symposium on Low Power Design*, pp 99-104, Dana Point, CA, 1995.

[47] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Accurate Area and Delay Estimators for FPGAs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp 862-869, Mar. 2002.

[48] M. Nemani and F. Najm, "Delay Estimation of VLSI Circuits from a High-Level View," in *Proceedings of the 35th Annual Design Automation Conference*, pp 591-594, San Francisco, CA, June 1998.

[49] M. Nourani and C. Papachristou, "False Path Exclusion in Delay Analysis of RTL-Based Datapath-Controller Designs," in *Proceedings of the Conference on European Design Automation*, pp 336-341, Geneva, Switzerland, Sept. 1996.

[50] P. Paulin and J. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASICs,” in *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 6, pp 661-679, June 1989.

[51] P. Paulin and J. Knight, “Scheduling and Binding Algorithms for High-Level Synthesis,” in *Proceedings of the Design and Automation Conference*, pp 1-6, Las Vegas, NV, June 1989.

[52] S. Peng, D. Fang, J. Teifel, and R. Manohar, “Automated synthesis for asynchronous FPGAs,” in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pp 163-173, Monterey, CA, 2005.

[53] M. Poletto and V. Sarkar, “Linear Scan Register Allocation,” in *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 895-913, Sept. 1999.

[54] N. Ramsey and M. Fernandez, “New Jersey Machine-Code Toolkit,” in *Proceedings of the 1995 USENIX Technical Conference*, pp 289-302, New Orleans, LA, 1995.

[55] N. Ramsey and M. Fernandez, “Specifying Representations of Machine Instructions,” in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, issue 3, pp 492-524, New York, NY, 1997.

[56] D. Rao and F. Kurdahi, “Partitioning by Regularity Extraction,” in *Proceedings of the Design and Automation Conference*, pp 235-238, Anaheim, CA, July 1992.

[57] A. Srinivasan, G. Huber, and D. LaPotin, “Accurate Area and Delay Estimations from RTL Descriptions,” in IEEE Transactions on VLSI Systems, vol. 6, no. 1, pp 168-180, March 1998.

[58] A. Stammermann, D. Helms, M. Schulte, A. Schulz, and W. Nebel, “Binding, Allocation and Floorplanning in Low Power High-level Synthesis,” in *Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design*, pp 544-550, 2003.

[59] G. Stitt and F. Vahid, “Dynamic Hardware/Software Partitioning: A First Approach,” in *Proceedings of the Design Automation Conference*, pp 250-255, Anaheim, CA, 2003.

[60] G. Stitt and F. Vahid, “Hardware/Software Partitioning of Software Binaries,” in *Proceedings of the International Conference of Computer Aided Design (ICCAD)*, pp 164-170, Santa Clara, CA, 2002.

[61] F. Vahid, T. Le, and Y.-C. Hsu, “A Comparison of Functional and Structural Partitioning,” in *Proceedings of the 9th International Symposium on System Synthesis*, pp 121-126, 1996.

[62] F. Vahid, T. Le, and Y.-C. Hsu, “Functional Partitioning Improvements Over Structural Partitioning for Packaging Constraints and Synthesis: Tool Performance,” in *ACM Transactions on Design Automation of Electronic Systems*, vol. 3, issue 2, pp 181-208, April 1998.

[63] K. S. Vallerio and N. K. Jha, “Task Graph Extraction for Embedded System Synthesis,” in *Proceedings of the International Conference on VLSI Design*, pp 480-486, Jan. 2003.

[64] G. Venkataraman, S. Reddy, and I. Pomeranz, “GALLOP: Genetic Algorithm Based Low Power FSM Synthesis by Simultaneous Partitioning and State Assignment,” in *Proceedings of the 16th International Conference on VLSI Design*, pp 533-538, Jan. 2003.

[65] W. Verhaegh, P. Lippens, E. Aarts, J. Korst, A. Werf, and J. Meerbergen, “Efficiency Improvements for Force-Directed Scheduling,” in *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*, pp 286-291, Nov. 1992.

[66] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli, “Explicit and Implicit Algorithms for Binate Covering Problems,” in *IEEE Transactions on Computer-Aided Design*, vol. 16, pp 677-691, July 1997.

[67] W. Wolf, A. Takach, C.-Y. Huang, R. Manno, and E. Wu, “The Princeton University Behavioral Synthesis System,” in *Proceedings of the 29th ACM/IEEE Conference on Design Automation*, pp 182-187, Anaheim, CA, 1992.

[68] W. H. Wolf, “An architectural Co-Synthesis Algorithm for Distributed, Embedded Computing Systems,” in *IEEE Transactions on VLSI Systems*, vol. 5, pp 218–229, June 1997.

[69] Y. Xie and W. Wolf, “Co-Synthesis with Custom ASICs,” in *Proceedings of the IEEE/ACM Asia and South Pacific Design Automation Conference*, pp 129-134, Yokohama, Japan, 2000.

[70] Y. Xie and W. Wolf, “Allocation and Scheduling of Conditional Task Graph in Hardware/Software Co-Synthesis,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp 620–625, Mar. 2001.

[71] M. Xu and F. Kurdahi, “Area and Timing Estimation for Lookup Table Based FPGAs,” in *Proceedings of the 1996 European conference on Design and Test*, pp 151-157, Mar. 1996.

[72] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee, “CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit,” in *Proceedings of the 27th International Symposium on Computer Architecture*, pp 225-235, Vancouver, Canada, 2000.

[73] D. Zaretsky, G. Mittal, R. Dick, and P. Banerjee, “Dynamic Template Generation for Resource Sharing in Control and Data Flow Graphs,” in *Proceedings of the International Conference on VLSI Design*, Hyderabad, India, Jan. 2006.

[74] D. Zaretsky, G. Mittal, R. Dick, and P. Banerjee, “Generation of Control and Data Flow Graphs from Scheduled and Pipelined Assembly Code,” in *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, Hawthorne, NY, Oct. 2005.

[75] D. Zaretsky, G. Mittal, X. Tang, and P. Banerjee, “Overview of the FREEDOM Compiler for Mapping DSP Software to FPGAs,” in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp 37-46, Napa, CA, 2004.

[76] D. Zaretsky, G. Mittal, X. Tang, and P. Banerjee, “Evaluation of scheduling and allocation algorithms while mapping assembly code onto FPGAs,” in *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pp 397–400, Boston, MA, 2004.

[77] Y. Zibin, J. Gil, and J. Considine, “Efficient Algorithms for Isomorphisms of Simple Types,” in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp 160-171, New Orleans, LA, Jan. 2003.

Appendix A

MST Grammar

```
Design <= Design Design_Name { Symbol_Table {Procedure}
          [{Function}] [{Library}] }

Symbol_Table <= {Symbol_Declaration}

Symbol_Declaration <= register Register_Name [Precision]
                      (signed | unsigned) [virtual]

Procedure <= Procedure Procedure_Name { {Instruction} }

Function <= Function Function_Name { {Instruction} }

Library <= Library Library_Name {}

Instruction <= Reference_Line Timestep PC [Label:] [Predicate] Operator
               [SRC1_Operand,] [SRC2_Operand,] [DST_Operand]

Predicate <= [[!]Operand]

Operator <= (Logical_Operator | Arithmetic_Operator |
             Branch_Operator | Assignment_Operator |
             Compare_Operator | General_Operator) (Delay, Sign)

Logical_Operator <= AND | NAND | NOR | NOT | OR | SLL | SRA | SRL |
                    XNOR | XOR

Arithmetic_Operator <= ADD | DIV | MULT | SUB | NEG

Branch_Operator <= BEQ | BGEQ | BGT | BLEQ | BLT | BNEQ |
                   CALL | GOTO | JMP

Assignment_Operator <= LD | MOVE | ST | UNION

Compare_Operator <= CMPEQ | CMPNE | CMPGT | CMPGE | CMPLT | CMPLE
```

General_Operator <= **NOP**

Operand <= (Immediate | Register | Memory | Label)

Immediate <= value **(**Sign Precision**)**

Register <= *Register*_Name [BitRange]

Memory <= ***[***Address*_Operand [**+** *Offset*_Operand]**]** [BitRange]

Label <= *Label*_Name

Name <= string

BitRange <= **[**(integer | integer:integer)**]**

Timestep <= double

Delay <= integer

Reference_Line <= integer

PC <= value

Precision <= integer

value <= hexadecimal

Sign <= **u’** | **s’**

Appendix B

HDL Grammar

```
design <= {entity}

entity <= entity entity_name()
            port_symbol_table
            global_symbol_table
            {component}
            {process}
          end entity

component <= component component_name()
               component_symbol_table
               {component_instance}
             end component

component_instance <= component_name : instance_name
                      ( {port_connection} )

port_connection <= port_symbol <= expression

process <= process process_name ({symbol_name[,]})
             [local_symbol_table]
             {statement}
           end process

comment <= // comment_string

statement <= assignment_statement |
             if_statement |
             switch_statement |
             case_statement |
             set_statement |
             read_statement |
             write_statement |
             report_statement |
```

```
            for_loop |
            while_loop |
            fsm |
            state

assignment_statement <= variable (<=|:=) expression

if_statement <= if ( expression ) {statement} [ else {statement} ] end

switch_statement <= switch ( expression ) {case_statement} end switch

case_statement <= (case expression | default) : {statement} end case

set_statement <= variable (<=|:=) expression ? expression : expression

read_statement <= variable (<=|:=) memory

write_statement <= memory (<=|:=) expression

report_statement <= report( "report_string" {, expression} )

null_statement <= NULL

for_loop <= for ( statement; expression; assignment_statement )
               {statement}
            end

while_loop <= while ( expression ) {statement} end

fsm <= fsm ( control_var_name ) {state} end fsm

state <= state state_id : {statement} end state

state_id <= integer | idle | init | done | reset | default

operator <= binary_operator | unary_operator

binary_operator <= logical_operator |
                   shift_operator |
                   arithmetic_operator |
                   relational_operator

logical_operator <= and | or | xor | nand | nor | xnor | &

shift_operator <= SLL | SRL | SRA

arithmetic_operator <= + | - | * | \ | %
```

```
relational_operator <= == | != | < | <= | > | >=

unary_operator <= - | ~

expression <=  binary_expression |
               unary_expression |
               function_call |
               primary |
               (expression)

binary_expression <= expression binary_operator expression

unary_expression <= unary_operator expression

function_call <= [$]function_name({expression[,]})

primary <= variable |
           memory |
           number |
           z

name <= {[A-Za-z0-9_]}

variable <= symbol_name [range]

memory <= symbol_name [expression] [range]

number <= precision'[sign]binary_value |
          precision'[sign]octal_value |
          precision'[sign]decimal_value |
          precision'[sign]hexidecimal_value

binary_value <= b{0-1}

octal_value <= o{0-7}

decimal_value <= d[+|-]{0-9}

hexadecimal_value <= h{0-9a-fA-F}

z <= precision'[sign][b|d|o|h]{z|Z|x|X}

precision <= integer

sign <= s | u | signed | unsigned

range <= [integer] | [integer:integer]

symbol_table <= {symbol_declaration}
```

```
symbol_declaration <= variable_declaration |
                      array_declaration |
                      port_declaration

variable_declaration <= symbol_type sign value_type [register]
                        [sensitive] range symbol_name

array_declaration <= variable_declaration range

port_declaration <= io_type array_declaration

symbol_type <= signal | variable | constant

value_type <= integer | decimal | binary | octal | hexadecimal

io_type <= input | output | inout
```

Appendix C

Verilog Simulation Testbench

```
`timescale 1 ps / 1 ps

module testbench();

   parameter
      clock_high = 50000,
      clock_low = 50000;

   reg   [0:0]    clock;
   reg   [0:0]    reset;
   wire  [0:0]    done;
   reg   [31:0]   tb_data_out;
   wire  [0:0]    ACK;
   wire  [31:0]   Errors;
   wire  [0:0]    EOF;
   reg   [0:0]    EOS = 0;
   reg   [0:0]    Action = 0;
   wire  [31:0]   mem_d;
   wire  [31:0]   mem_q;
   wire  [9:0]    mem_addr;
   wire           mem_we;
   integer        i, j;

   // main procedure module
   main main_func
   (
      .clock( clock ),
      .reset( reset ),
      .done( done ),
      .mem_d(mem_d),
      .mem_addr(mem_addr),
      .mem_we(mem_we),
      .mem_q(mem_q)
   );
```

```
// memory module
RAM_1024x32 RAM_1024x32_instance_0
(
   .clock(clock),
   .d(mem_d),
   .addr(mem_addr),
   .we(mem_we),
   .q(mem_q)
);

// file I/O module
file_out file_outdata
(
    .reset( reset ),
   .action( Action ),
   .eos( EOS ),
   .data( tb_data_out ),
   .errors( Errors ),
   .acknowledge( ACK ),
   .eof( EOF )
);

// clock process
always begin : clock_process
   clock = #clock_low 1'b1;
   clock = #clock_high 1'b0;
end

// reset process
always begin : reset_process
   $write( "@ %0t: beginning simulation...\n", $time );
   reset = 1'b1;
   @(posedge clock);
   @(negedge clock);
   reset = 1'b0;
   @(posedge done);
   @(posedge clock);
   @(negedge clock);
   wait( 0 );
end

// interrupt process
always @(posedge done) begin
   @(posedge clock);
   @(negedge clock);
   @(posedge clock);
   $write( "@ %0t: simulation completed.\n", $time );
   $write( "Total simulation cycle count: %0d\n\n",
            $time/(clock_high+clock_low) );
```

```
      // write memory data to file
      for ( j=0; j<1024; j=j+1 ) begin
         Action <= 1;
         tb_data_out = RAM_1024x32_instance_0.mem[j];
         @(posedge ACK)
         @(posedge clock);
         Action <= 0;
         @(negedge ACK);
         @(negedge clock);
      end
      EOS <= 1;
      $stop;
   end

   // memory initialization
   initial begin
      for ( i=0; i<1024; i=i+1 ) begin
         RAM_1024x32_instance_0.mem[i] = 0;
      end
      $readmemh("mem.txt", RAM_1024x32_instance_0.mem );
   end

endmodule


module file_out( reset, action, acknowledge, eof, eos, errors, data );

   parameter      filename = "output_tb.txt";
   parameter      precision = 32;
   parameter      resolution = 0;

   input        reset;
   input        action;
   input       eos;
   input signed [precision+resolution-1:0] data;

   output [31:0] errors;
   output reg    acknowledge = 0;
   output reg    eof = 1;

   integer      lineno;
   integer      final_errors = 0;
   integer      writer_errors = 0;

   assign      errors = final_errors + writer_errors;

   integer      f, v, r;

   reg         open_file = 0;
   reg [1024*8:1] fn;
```

```
   always @(posedge reset) begin
      $fclose( f );
      acknowledge <= 0;
      @(negedge reset);

      f = $fopen( filename, "w" );

      if ( f == 0 ) begin
          $write( "ERROR: failed to open output file: %0s\n", fn );
          $stop;
      end
   end

   always @(posedge eos) begin
      $fclose( f );
   end

   always @( posedge action ) begin
      if ( reset == 0 && eos == 0 ) begin
         $fwrite( f, "%0x\n", data );
      end

      acknowledge <= 1;
      @(negedge action);
      acknowledge <= 0;
   end

endmodule
```