# Aviv: A Retargetable Code Generator for Embedded Processors

by

Silvina Zimi Hanono

S.M. (E.E.C.S) Massachusetts Institute of Technology (Feb. 1995)
B.S. (Elec. Engr.) Cornell University (Jan. 1992)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999



Author .....
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by .....
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

# Aviv: A Retargetable Code Generator for Embedded Processors

by

Silvina Zimi Hanono

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

## Abstract

Embedded systems are broadly defined as systems designed for a particular application. The functionality of an embedded system is divided into hardware and software components. Synthesis of the hardware component involves designing a custom circuit for the hardware portion of the input application. Synthesis of the software component consists of designing a processor that is suited for the software portion of the input application and generating code that implements the functionality of the software component on the designed processor. Short design cycles and increasing embedded system complexity make it impractical to perform manual processor architecture exploration and code generation. In order to effectively explore the design space for the software component of embedded systems, a retargetable code generator is required.

This thesis presents the AVIV retargetable code generator that generates optimized machine code for a specified target processor. AVIV is capable of compiling application code into machine code for a variety of architectures. In particular, it is focused on generating code for architectures with instruction-level parallelism.

The key problems in code generation are instruction selection, resource allocation, and scheduling. Most current code generation systems solve these problems sequentially. However, these problems are interdependent; therefore, solving them independently can lead to suboptimal code. In order to address the interdependencies, AVIV solves all three problems concurrently. AVIV uses a new graphical representation, the Split-Node DAG, to specify all possible ways of implementing the software component of the application on the target processor. A heuristic algorithm utilizes the information embedded in the Split-Node DAG to solve the various problems of code generation concurrently, thereby identifying a globally optimized solution.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

*To my parents, Saul and Sarita,*
*and my sisters, Judy and Anat,*
*with love.*

# Acknowledgments

I would like to begin by thanking my advisor, Professor Srinivas Devadas, for being an extremely supportive, helpful, friendly, and fun advisor to work with. His open door policy always made it easy to approach him about any matter, academic or non-academic. His enthusiasm made every problem seem more interesting, and his encouragement helped make it all come together. Professor Devadas first introduced me to the world of Computer-Aided Design through the graduate CAD course he teaches at MIT. There began my enthusiasm for this field, and my interest in working with Srini.

I would also like to thank my readers, Professor Anant Agarwal with whom I had the pleasure of pursuing my Master's degree, and Professor Saman Amarasinghe. I have enjoyed holding conversations with them about my research as well as other topics. Their insights often helped me realize issues that I had previously neglected. I thank them for their time and effort in helping me successfully complete my PhD.

During the course of my research, I have been a member of the SPAM project which is a joint academic and industrial effort in the area of retargetable code generation for embedded systems. SPAM initially consisted of Synopsys, Princeton University, Aachen University, and MIT. Since then it has grown to include several additional institutions. I thoroughly enjoyed being a member of the SPAM project, and I would like to extend my gratitude to all of its members for contributing to my understanding of the field of retargetable code generation.

It has been a pleasure working with and really getting to know the members of my research group. I am indebted to my officemate Daniel Engels who has meticulously edited all of my papers and this thesis, helped me incessantly whenever I needed to discuss any matter, and most importantly has been a truly wonderful friend. I look forward to many years of this continued friendship. I would like to thank George Hadjiyiannis with whom I have spent endless hours ironing out details about ISDL, examining problems in my code generator, and just hanging out. George is the member of the team with whom I have worked most closely, both in co-developing

ISDL and in integrating our research projects. His expansive knowledge in the field has been an asset to us all.

I would also like to thank the rest of the members of my group, Farzan Fallah, Sandeep Chatterjee, and Prabhat Jain as well as all of my close friends at MIT. They have helped make my work environment a pleasant and fun place to be. I have enjoyed their company on outings, during late nights in the lab, and on a daily basis.

A special thank you goes to my husband, Josh Wachman, who has always believed in me and pushed me to do my very best. He has been a wonderful partner and best friend. His interest in learning allowed us to discuss my work almost as soon as we met. His love, encouragement, and support helped make the entire process a much more pleasant experience.

Finally, I would like to thank my family. My parents, Saul and Sarita, and my sisters, Judy and Anat, for being the best family anyone could ever hope for. Their endless love, support, and encouragement has helped me make it through every problem I have yet to encounter. My parents always believed in providing the very best for us, and pushed us to strive for the same. Education has always been a number one priority in my family, and that feeling has become part of my identity. It has been difficult to spend so many years apart from my family while pursuing my undergraduate and graduate degrees. Nevertheless, we have remained as close as ever. For all their love, support, and dedication, I proudly dedicate this thesis to my parents and sisters whom I love very much.

# About the Author

**Silvina Zimi Hanono** was born on September 21, 1970 in Rosario, Argentina. September 21st is "el dia de la primavera", the first day of Spring in Argentina. At the young age of fourteen months, she moved with her family to Israel where she lived for the next five years. At the age of six she arrived in the United States for the first time. Since then she has moved back and forth between Israel and the United States a few more times. As a result of moving from country to country, Silvina and her family speak English, Hebrew, and Spanish fluently and often switch languages mid-sentence. This is a well known trait in the Hanono family. In trying to come up with a name for her thesis, it seemed appropriate to give it a name that would identify with her upbringing. She chose the name AVIV, the Hebrew word for Spring (la primavera), to symbolize her Israeli and Argentinian origins.

Silvina's higher education began at Cornell University where she received her B.S. in Electrical Engineering in January 1992. She then moved on to MIT where she received her S.M. in Electrical Engineering and Computer Science in February 1995. Silvina was a recipient of a National Science Foundation Fellowship (1992) for graduate study. She has served as a teaching assistant (1994) and recitation instructor (1996) for the undergraduate introductory course to Computer Architecture, *Computation Structures (6.004)*. Silvina is a member of several national honor societies including Eta Kappa Nu and Tau Beta Pi. She is also a member of the Institute of Electrical and Electronics Engineers (IEEE).

### Recent Publications

"Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator," S. Hanono, S. Devadas. In *Proceedings of the 35th Design Automation Conference,* pages 510-515, June 1998.

"ISDL: An Instruction Set Description Language for Retargetability," G. Hadjiyiannis, S. Hanono, and S. Devadas. In *Proceedings of the 34th Design Automation Conference,* pages 299-302, June 1997.

# Contents


**1 Introduction** **25**

1.1 The AVIV Retargetable Code Generation System . . . . . 28

1.1.1 Typical Compiler Structure . . . . . 29

1.1.2 AVIV's Code Generation Framework . . . . . 29

1.2 The Software Synthesis Loop . . . . . 33

1.3 Project ARIES . . . . . 35

1.3.1 Hardware-Software Co-Design . . . . . 36

1.3.2 The ARIES System Design Framework . . . . . 37

1.3.3 The ARIES System Design Methodology . . . . . 38

1.4 Contributions of this Thesis . . . . . 41

1.5 Organization of this Thesis . . . . . 42

**2 Related Work** **45**

2.1 Related Work on Addressing Multiple Tasks of Code Generation Concurrently . . . . . 46

2.1.1 Integrating Register Allocation and Instruction Scheduling . . 46

2.1.2 Performing Data Routing and Scheduling Concurrently . . . . 47

2.1.3 Unified Resource Allocation for Registers and Functional Units in VLIW Architectures . . . . . 48

2.1.4 Integrating Code Selection and Register Allocation into Instruction Scheduling . . . . . 49

2.1.5 Integrating Instruction Selection, Resource Allocation, and Scheduling for Heterogeneous Register Machines . . . . . 51

2.1.6 Summary and Comparison to AVIV . . . . . 52
2.2 Related Work in Retargetable Code Generation . . . . . 52
2.2.1 The Record Compiler Generator . . . . . 53
2.2.2 FlexWare . . . . . 54
2.2.3 Chess . . . . . 55
2.2.4 Wilson et al. . . . . 57
2.3 Related Work on Machine Description Languages for Embedded Processors . . . . . 58
2.3.1 Mimola . . . . . 59
2.3.2 nML . . . . . 59
2.3.3 Lisa . . . . . 60
2.3.4 Summary and Comparison to ISDL . . . . . 60

**3 ISDL: Instruction Set Description Language 61**
3.1 Definitions . . . . . 62
3.2 ISDL Model of the Instruction Set . . . . . 63
3.3 The Structure of an ISDL Description . . . . . 64
3.3.1 Instruction Word Format . . . . . 66
3.3.2 Global Definitions . . . . . 66
3.3.3 Storage Resources . . . . . 69
3.3.4 Instruction Set . . . . . 70
3.3.5 Constraints . . . . . 72
3.3.6 Optional Architectural Details . . . . . 73
3.4 An ISDL Example . . . . . 73
3.5 Using Constraints to Simplify the Machine Description . . . . . 80

**4 The Split-Node DAG 85**
4.1 Roadmap for Transforming High-Level Code into a Split-Node DAG . 87
4.2 The Compiler Front-End . . . . . 88
4.2.1 Overview of the SUIF Compiler . . . . . 88
4.2.2 Machine-Independent Optimizations . . . . . 88

4.2.3 Creating the Basic Block DAGs . . . . . 92
4.3 Matching SUIF Operations to ISDL Operations . . . . . 93
4.4 Additional Databases Created by AVIV . . . . . 95
4.5 Converting the Basic Block DAGs into Split-Node DAGs . . . . . 97

**5 Covering the Split-Node DAG** **103**
5.1 Code Generation Tasks . . . . . 103
5.2 Code Generation Using the Split-Node DAG . . . . . 108
5.2.1 Exploring the Split-Node Functional Unit Assignments . . . . 110
5.2.2 Adding the Required Transfers . . . . . 115
5.2.3 Maximal Groupings . . . . . 117
5.2.4 Selecting a Minimum-Cost Set of Maximal Cliques . . . . . 124
5.2.5 The Covering Solution . . . . . 127
5.2.6 Detailed Register Allocation . . . . . 128
5.2.7 Peephole Optimization . . . . . 128
5.3 Alternate Implementation . . . . . 129
5.4 Handling Complex Operations . . . . . 130
5.5 Handling Non-Uniform Operation Latencies . . . . . 132

**6 Control Flow** **135**
6.1 The If-Then-Else Control Flow Construct . . . . . 135
6.2 The For-Loop Control Flow Construct . . . . . 141
6.3 Possible Control Flow Optimizations . . . . . 143
6.3.1 Trace Scheduling . . . . . 143
6.3.2 Code Motion . . . . . 143
6.3.3 Loop Optimizations . . . . . 145

**7 The Constraints Checker** **147**
7.1 The Constraints Syntax . . . . . 148
7.2 Using the Constraint Checker . . . . . 153
7.2.1 Removing Invalid Non-Terminal Options . . . . . 153

7.2.2 Testing Maximal Cliques Prior to Instruction Selection . . . . 155
7.2.3 Testing the Final Selected Instructions After Register Allocation 156
7.2.4 Checking for Time-Shifted Constraints . . . . . 157
7.3 The Constraint Checker . . . . . 158
7.3.1 Technique I . . . . . 159
7.3.2 Technique II . . . . . 163
7.4 Performance Comparison of Techniques I and II . . . . . 169

**8 The Assembler Generator** **171**
8.1 Lex and Yacc . . . . . 171
8.2 ISDL Information Required for the Creation of an Assembler . . . . . 173
8.3 The Generated Lex File . . . . . 174
8.4 The Generated Yacc File . . . . . 175

**9 Results** **181**
9.1 Code Generation Results . . . . . 182
9.2 Analysis of AVIV . . . . . 186
9.3 Modifying the Target Architecture . . . . . 187
9.4 Exploring the Performance Capabilities of the AVIV Code Generator . 189

**10 Conclusions** **197**
10.1 A Complete Working System . . . . . 198
10.2 The Split-Node DAG . . . . . 199
10.3 The Quality of the Code Produced Using AVIV's Heuristics . . . . . 199
10.4 Generating Maximal Cliques and Verifying their Validity . . . . . 201
10.5 The Covering Algorithm . . . . . 203
10.6 Using AVIV in the Software Synthesis Loop . . . . . 204
10.7 Summary of Directions for Future Work . . . . . 205

**A An Example ISDL Description** **207**

**B An Example ISDL Description Including Control Flow** **213**

**C Example Code Segments** **221**

C.1 Example 1 . . . . . 221

C.2 Example 2 . . . . . 222

C.3 Example 3 . . . . . 222

C.4 Example 4 . . . . . 223

C.5 Example 5 . . . . . 224

# List of Figures


1-1 A heterogeneous system-on-a-chip . . . . . 27
1-2 Retargetable code generation framework using AVIV . . . . . 30
1-3 Software synthesis methodology . . . . . 34
1-4 A generic hardware–software co-design methodology . . . . . 36
1-5 ARIES framework . . . . . 37
1-6 ARIES design methodology . . . . . 39

3-1 The design flow for an ASIP . . . . . 62
3-2 ISDL model of instructions . . . . . 65
3-3 The example VLIW architecture . . . . . 74
3-4 The instruction word of the example VLIW architecture . . . . . 74
3-5 Constraints help simplify the machine description . . . . . 81

4-1 Retargetable code generation framework using AVIV . . . . . 86
4-2 Converting a basic block of code into a basic block DAG . . . . . 92
4-3 Expression trees for add operation . . . . . 94
4-4 Expression tree for branch on false operation . . . . . 95
4-5 Example basic block DAG . . . . . 98
4-6 Example target architecture . . . . . 99
4-7 Splitting the basic block DAG nodes into multiple nodes . . . . . 100
4-8 The Split-Node DAG . . . . . 102

5-1 Instruction selection . . . . . 104
5-2 Optimizing the schedule by reordering the instructions . . . . . 106

5-3 Splitting a DAG into multiple trees . . . . . 107
5-4 Algorithm for covering the Split-Node DAG . . . . . 109
5-5 The Split-Node DAG . . . . . 110
5-6 Algorithm for selecting split-node functional unit assignments . . . . 112
5-7 Pruning the search space of split-node assignments . . . . . 113
5-8 The Split-Node DAG after functional unit assignment . . . . . 115
5-9 Algorithm for selecting the best transfer nodes for functional unit assignment . . . . . 117
5-10 Generating the maximal cliques . . . . . 119
5-11 Pseudo-code for generating maximal cliques . . . . . 121
5-12 Algorithm for covering the nodes in an assignment using a minimum-cost set of cliques . . . . . 125
5-13 Inserting loads and spills into the Split-Node DAG . . . . . 127
5-14 Cycles in the clique selections . . . . . 130
5-15 Incorporating complex operations into the Split-Node DAG . . . . . 132

6-1 Low-level SUIF representation of if-then-else statement . . . . . 136
6-2 Basic block DAGs for if-then-else statement . . . . . 137
6-3 Control flow graph for if-then-else example code . . . . . 138
6-4 Example target architecture with support for control flow . . . . . 138
6-5 Split-Node DAGs for if-then-else example code . . . . . 139
6-6 Control flow graph for a for-loop . . . . . 142

7-1 Pseudo-code for technique I of constraints checker . . . . . 161
7-2 Pseudo-code for technique II of constraints checker . . . . . 166

9-1 The ARIES1-ARIES5 target architectures . . . . . 183
9-2 Data flow of a four-way parallel application . . . . . 191
9-3 The effects of block size and data parallelism on runtime . . . . . 192
9-4 Data flow graph example II . . . . . 193
9-5 Schedule for data flow graph II . . . . . 194

9-6 Input program structure can produce different DAGs . . . . . 196

# List of Tables

7.1 Sample results of constraint checker . . . . . 151
7.2 Performance comparison of techniques I and II of the constraint checker 169

9.1 Code generation experiments for ARIES1 . . . . . 184
9.2 Comparison of code generation results on ARIES1 and ARIES3 . . . . 185
9.3 Code generation experiments for ARIES2 . . . . . 185
9.4 Code size produced for ARIES1 versus ARIES2 architectures . . . . . 186
9.5 Breakdown of CPU time . . . . . 187
9.6 Utilization statistics for example 4 . . . . . 188
9.7 Utilization statistics for example 5 . . . . . 188

# AVIV: A Retargetable Code Generator for Embedded Processors

Silvina Zimi Hanono

# Chapter 1

# Introduction

Consumer electronic products have become an integral component of modern daily life. In recent years, society has witnessed a proliferation of electronic devices such as cellular phones, personal digital assistants, and laptop computers. The complexity of these new products is increasing as designers attempt to provide additional functionality while meeting strict design constraints. In addition, market competition and the increasing demand for electronic equipment are pushing designers to shorten the design cycles of new products.

Modern electronics are controlled by complex digital systems that must meet strict constraints in terms of performance, cost, size, and power consumption. In a competitive marketplace, performance and cost are critical in differentiating one product from another. In addition, low cost and superior performance increase the likelihood of broad consumer acceptance of new electronic products. Size constraints limit the amount of functionality that can be incorporated into a product design. Finally, low power consumption is necessary for portable electronic equipment that is battery-operated.

Designing an entire complex system as an Application Specific Integrated Circuit (ASIC) is neither economical nor practical. ASICs exhibit low design flexibility because they do not support design changes at late stages of the design cycle. Furthermore, ASICs require long design cycles which incur a high design cost. Therefore, ASICs are not well suited to meet the short time-to-market requirements of new elec-

tronic products. As a result, design trends have migrated away from full ASIC implementations towards systems that include programmable components. Programmable processors increase flexibility by allowing design changes to occur at late stages of the design cycle. In addition, systems containing programmable components are easier to implement than ASICs. As a result, they require shorter design cycles and reduced design costs.

An *embedded system* is a computer system whose hardware (ASIC) and software (programmable processor and application code) are designed for a specific application rather than for general-purpose computing. Examples of embedded systems include engine management units, dishwasher controllers, and cellular phones. The design of embedded systems addresses the design constraints discussed above. One of the main characteristics of embedded systems is that the design of the software component is as important as the design of the hardware component. The software component of embedded systems is referred to as the *embedded software*, and the processor on which the software is executed is referred to as an *embedded processor.*

In order to meet the design constraints of embedded systems, it is common to integrate the entire system on a single integrated circuit (IC) [20]. This is enabled by current deep submicron processing technology which supports the integration of over 200 million transistors on a single IC [57]. Figure 1-1 illustrates a possible system-on-a-chip architecture consisting of a processor core, program ROM, memory, application specific circuitry, and peripherals. The processor core may be an off-the-shelf processor core or an in-house Application Specific Instruction-Set Processor (ASIP). An ASIP is a processor that is designed to efficiently execute the software for a specific application. Regardless of whether a newly designed ASIP or a preexisting processor core is used, the selected processor should be well suited for the given application.

Although incorporating a complete system on a single IC may improve performance, cost, and power consumption requirements, such a high level of integration constrains the size of the system components. Furthermore, the cost of an integrated circuit increases exponentially with its size [29]. As a result, it is imperative to design

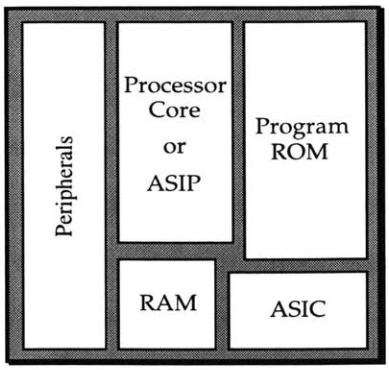



Figure 1-1: **A heterogeneous system-on-a-chip**

both the hardware and software components with minimum size in mind. This implies that, not only the ASIC and embedded processor need to be designed within the size constraints of the given IC, but the remainder of the system components must meet the strict size constraints as well. The program ROM, which stores the application code that is to be executed on the embedded processor, is a good candidate for size reduction efforts. Reducing the size of the application code reduces the size of the program ROM. The application code size can be reduced by developing a code generator geared towards producing code that is optimized for minimum code size.

In order to design an embedded system, the embedded system application is partitioned into hardware and software components. An ASIC is designed to implement the functionality of the hardware component. The design of the software component consists of: (1) designing a processor (ASIP) to execute the software component, and (2) generating code that implements the functionality of the software component on the designed processor. In order to produce a highly optimized software component, it may be necessary to explore multiple target processors. This implies that the code that is generated for the software component would have to be compiled for various target processor architectures.

The difficulty in producing code for an ASIP is that compilers for high-level languages, such as C or C++, are not readily available for newly designed ASIPs. Therefore, the traditional approach to developing embedded software code has been to write

the application code in assembly language [44]. As the complexity of embedded systems increases, programming in assembly language and optimizing the code manually become impractical. Furthermore, hand coding virtually eliminates the possibility of changing the embedded processor architecture since the code must be rewritten every time the processor architecture is modified. In order to be able to explore various target processors and automatically generate code for each different architecture, a *retargetable* code generator – a code generator that can automatically generate code for different target architectures – is required.

This thesis focuses on the task of retargetable code generation. The major contribution of this thesis is the presentation of the AVIV retargetable code generator [26] which automatically produces optimized machine code for a variety of target processor architectures. In addition, this thesis presents the Instruction Set Description Language (ISDL) [23, 22] which is a machine description language that was developed to support the communication between the ASIP design and the retargetable code generation systems.

## 1.1 The AVIV Retargetable Code Generation System

The AVIV retargetable code generator supports the exploration of the processor design space by producing optimized machine code for target processors with various instruction set architectures. It focuses on architectures exhibiting instruction-level parallelism (ILP), including Very Long Instruction Word (VLIW) architectures. The code generated by AVIV is optimized for minimum code size in order to reduce the size requirements of the program ROM that stores the generated code.

Code generation consists of the following three tasks:

- *Instruction Selection* - Selecting the target processor instructions that will best implement the functionality of the input code.
- *Resource Allocation* - Assigning target processor resources including functional

units, registers, and memories to each selected instruction.

- *Scheduling* - Ordering the selected instructions so that all data dependencies are satisfied and the resulting schedule length is minimized.

Most current code generation systems address these tasks sequentially. However, previous studies have shown that optimizing one task of code generation without taking into account the effect on the other tasks leads to suboptimal results [20]. AVIV avoids this problem by addressing the various code generation tasks concurrently, thereby increasing the likelihood of finding a *globally* optimal solution.

The remainder of this section describes the structure of the AVIV retargetable compiler. It begins with a general overview of a typical compiler structure and continues with a high-level presentation of the AVIV code generation framework.

### 1.1.1 Typical Compiler Structure

Compilers typically consist of a front-end and a back-end [3]. The input to the compiler *front-end* is the application program written in a high-level language. The front-end transforms the high-level code into a machine-independent intermediate representation of the program. During this process, the front-end may perform machine-independent optimizations on the input code. The compiler *back-end* converts the intermediate representation into assembly or binary code specific to the target processor. Ordinary compilers assume a specific target architecture and optimize the code for that architecture only. A *retargetable* compiler's back-end must infer the capabilities of the target processor from an input description of the processor and retarget its optimization functions to the given target processor.

### 1.1.2 AVIV's Code Generation Framework

AVIV's code generation framework is illustrated in Figure 1-2. Its front-end consists of the SUIF[1] [60] and SPAM [49] compilers. The SUIF compiler is a research compiler de-

[1]Stanford University Intermediate Format

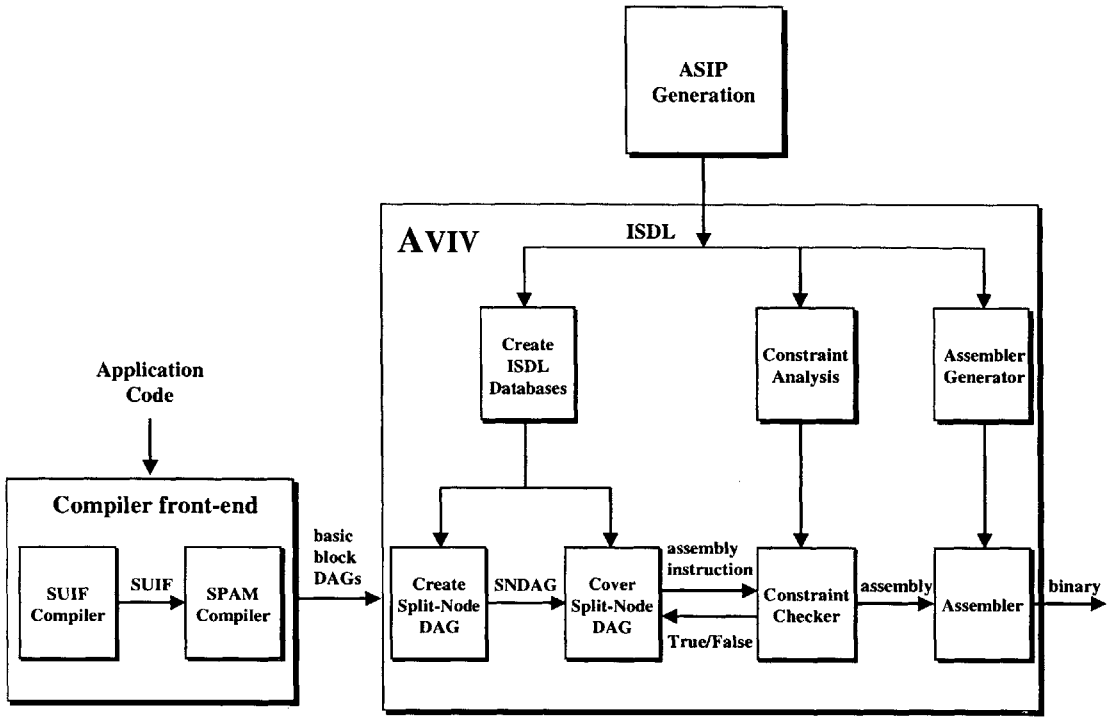



Figure 1-2: **Retargetable code generation framework using AVIV**

signed to facilitate the experimentation and development of new compiler algorithms. It receives the application source code, written in a high-level language such as C or Fortran, as input, and converts it to an intermediate representation named SUIF [50]. The SUIF compiler performs a series of machine-independent transformations in order to improve the quality of the code. Some examples of machine-independent optimizations are dead-code elimination and loop transformations such as loop-invariant code motion and loop unrolling. Transformations such as loop unrolling result in an increase in available parallelism within a segment of code.

After performing machine-independent optimizations, the optimized SUIF intermediate representation of the application code is passed to the SPAM compiler. The SPAM compiler is a retargetable code generation and optimization framework for embedded Digital Signal Processors (DSPs). It was designed to be a *developer retargetable* compiler. This means that using SPAM, a developer can build an optimizing

compiler for a new architecture by making function calls to various routines within the SPAM compiler using appropriate machine-specific parameters. Within the AVIV code generation framework, the SPAM compiler is used to convert the SUIF intermediate representation into a set of machine-independent basic block Directed Acyclic Graphs (DAGs) that are connected through control flow information. Each node of the DAG represents an operation in the input application, and each edge specifies a precedence relation between a pair of nodes.

In order to convert the machine-independent representation of the application code into a machine-specific representation, the back-end of the AVIV compiler must receive a description of the target processor. This description is provided using the ISDL machine description language. The ISDL description of the target processor is produced by the ASIP design system or by a high-level specification module in the case of an off-the-shelf processor.

The AVIV retargetable code generator receives the machine-independent basic block DAGs and the ISDL description of the target processor as input. It converts the basic block DAGs into machine code that can be executed on the target processor by following the steps listed below:

1. Parse the ISDL description and store the extracted information in various databases (Chapter 4).

2. Convert the basic-block DAGs into Split-Node DAGs (SNDAGs) (Chapter 4).

3. Cover the nodes of the Split-Node DAG with target processor instructions (Chapter 5).

4. Ensure that each selected instruction is valid using the constraint checker (Chapter 7).

5. Assemble the assembly format of the selected instructions into binary format (Chapter 8).

The first step of the AVIV compiler is to parse the ISDL description and extract all of the information relevant to the various steps of code generation. This infor-

mation is stored in databases that assist the code generator in subsequent steps of the code generation process. For example, one database stores all of the operations that the target processor can execute, while another database stores all of the storage capabilities of the target processor.

The second step of the AVIV compiler is to convert the machine-independent DAGs into a machine-dependent graphical representation called the *Split-Node DAG*. The Split-Node DAG explicitly represents all possible ways of implementing the application code on the target processor. The structure of the Split-Node DAG and the process by which it is created are described in detail in Chapter 4.

Before describing the remaining steps of AVIV's code generation methodology, the distinction between target processor *operations* and *instructions* should be clarified. The AVIV code generator and the ISDL description language both support architectures with instruction-level parallelism. Instruction-level parallelism implies that a single target processor instruction may represent multiple operations that are to be executed in parallel.

The third step of the compilation process is to select a set of target processor instructions that will optimally *cover* the nodes of the Split-Node DAG. *Covering* a node refers to selecting a target processor operation whose functionality is equivalent to the functionality of the node. Trade-offs between available resources and potential parallelism on the target processor are considered when covering the nodes.

During the third step, the selected operations are merged into target processor instructions. However, not all possible mergings of operations into instructions are valid (e.g., some operation groupings may result in resource conflicts). In order to ensure the validity of a potential instruction, the fourth step of AVIV's compilation methodology passes each potential instruction through a *constraint checker*. The constraint checker compares each instruction to all of the target processor constraints specified in the ISDL description. If the instruction satisfies all of the constraints then it is a valid instruction, and it may be selected to cover the corresponding nodes of the Split-Node DAG. The instructions selected to cover the Split-Node DAG are output in the assembly language corresponding to the target processor.

The fifth and final step of the compilation process assembles the selected instructions into binary (machine) code. The assembler for the target processor is automatically generated using the assembler generator.

As illustrated in Figure 1-2, within the AVIV compiler, the ISDL description is used to create the databases that assist the code generation process, to extract the constraints of the target processor for the constraint checker, and to automatically create an assembler for the target processor.

## 1.2 The Software Synthesis Loop

The AVIV retargetable code generator is one element of a software synthesis system that supports the automatic exploration of potential target processor architectures. Such a system can be used to produce an ASIP and corresponding code that are best suited for the software component of the input application. In this system, the ASIP generation module produces an ASIP for the input application. AVIV then generates code for the input application using instructions available on the target ASIP. Once the code has been generated, an instruction-level simulator analyzes the generated code and provides feedback to the ASIP generation module about the appropriateness of the ASIP for the input application. Based on this analysis, the ASIP generation module refines the ASIP in order to better satisfy the requirements of the input application. AVIV then regenerates code for the application using instructions available on the modified processor. The ASIP generation (or modification) and code generation loop, referred to as the *software synthesis loop*, is iterated until an optimized ASIP and corresponding code are generated. This design loop is highlighted in Figure 1-3.

In the first iteration of the software synthesis loop, the ASIP generation module analyzes the input application and estimates the utilization of the operations (e.g., addition and subtraction) present in the application code. Based on the utilization estimates, the ASIP generation module proposes an initial target architecture. The proposed architecture may be an ASIP or an off-the-shelf processor. If an ASIP is selected as the target architecture, then it is described to the code generator using

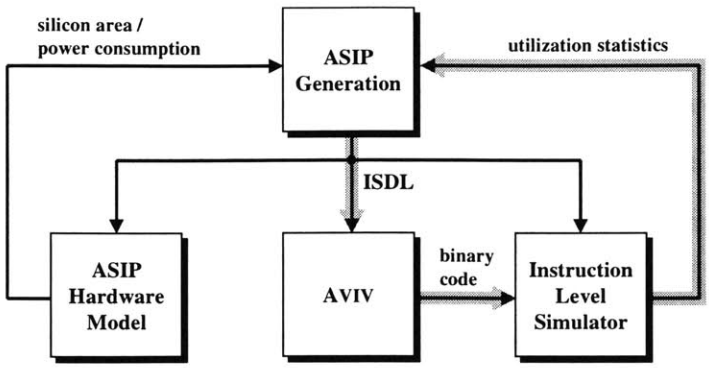



Figure 1-3: **Software synthesis methodology**

ISDL. The software synthesis loop is then iterated until an acceptable ASIP and corresponding code are produced. If an off-the-shelf processor is selected, then the ASIP generation step is skipped. Furthermore, if a compiler for this processor already exists, then the software synthesis task simply consists of compiling the software component using the preexisting compiler. If a compiler does not exist, then the off-the-shelf processor is described using ISDL. AVIV then generates code for the processor. In this scenario, no iteration of the software synthesis loop is required because an off-the-shelf processor cannot be modified.

In addition to being used by the code generator to compile the application code into machine code, the ISDL description is also used to automatically produce an instruction-level simulator (ILS) and a hardware model of the target ASIP [25]. The ILS executes the compiled machine code and produces an execution address trace. From the address trace, the ILS generates dynamic instruction frequencies for the input application. The dynamic instruction frequencies together with the compiled machine code determine the dynamic operation frequencies. Finally, from the dynamic operation frequencies, the ILS derives utilization statistics for the operations and functional units available in the target ASIP architecture. The hardware model of the ASIP determines the silicon area and power consumption requirements of each

operation and functional unit.

The ASIP generation module modifies the target architecture based on the utilization statistics produced by the ILS. Operations or functional units with low utilization may be redundant and can potentially be removed. Alternatively, a functional unit with very high utilization may denote a bottleneck requiring the allocation of additional resources. For example, if the only functional unit that contains a multiply operation has a high utilization and multiply operations comprise a substantial percentage of the operations for that unit, then multiplication resources are a bottleneck and should be expanded. The silicon area and power consumption costs, provided by the ASIP hardware model, are considered whenever operations or functional units are added or removed.

Additionally, the AVIV code generator may provide information that is used to improve the ASIP architecture. In particular, information about the number of load and spill operations required for the target application helps determine the optimum size of the register files.

After modifying the target architecture based on the statistics generated by the ILS and the hardware model of the previous target processor, the code generation and evaluation processes are repeated. The software synthesis loop is iterated until a suitable ASIP and corresponding optimized machine code are obtained. The optimized ASIP and its corresponding machine code comprise the software component of the embedded system being designed.

## 1.3 Project ARIES

The hardware and software components of embedded systems are highly interrelated. In order to effectively consider the trade-offs in embedded system design, the hardware and software should be designed and evaluated in a unified environment. Hardware-software co-design methodologies [21, 30, 53, 40, 10] have evolved to support concurrent and unified development of the hardware and software components.

This section begins with a high-level description of hardware-software co-design.

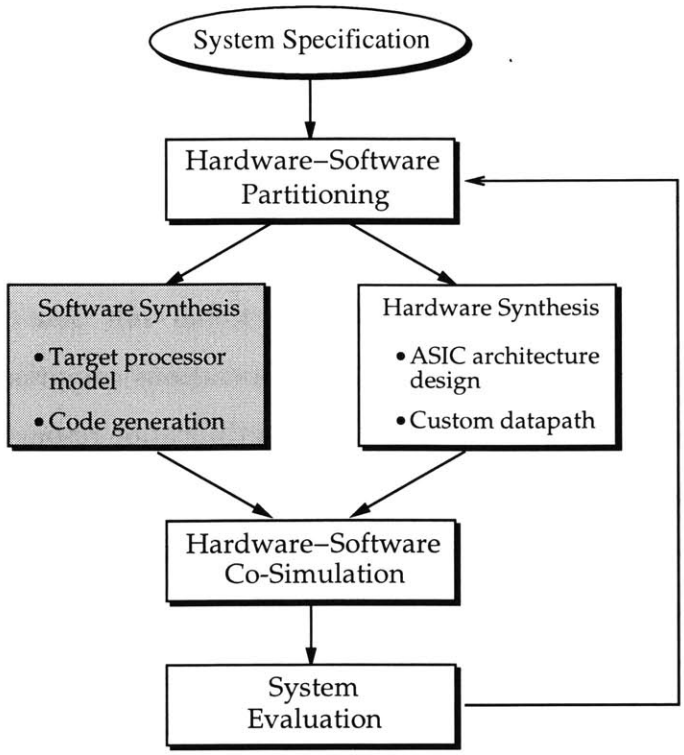



Figure 1-4: **A generic hardware–software co-design methodology**

It then presents Project ARIES as one possible implementation of a hardware-software co-design system. ARIES is an automated design system that enables the generation of mixed hardware and software embedded systems. The software synthesis system described in Sections 1.1 and 1.2 is a component of the ARIES [12] hardware-software co-design project.

### 1.3.1 Hardware-Software Co-Design

A simplified view of a generic hardware-software co-design methodology is shown in Figure 1-4. In this design flow, the system functionality is partitioned into hardware and software components. Hardware synthesis consists of designing application specific circuitry (ASIC) for the hardware component. Software synthesis involves: (1) selecting a target processor to execute the software component, and (2) generating code that implements the functionality of the software component on the selected target processor. Once the hardware and software syntheses are completed, the ASIC,

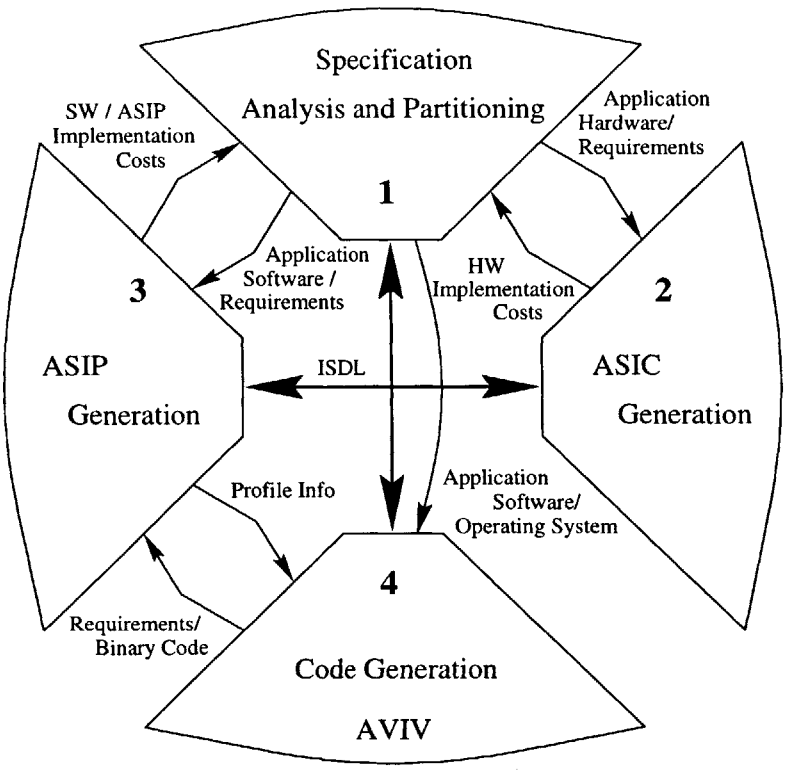



Figure 1-5: **ARIES framework**

processor, and generated code are combined into a complete system model. The entire system is then evaluated using a hardware-software co-simulator. If the design constraints are not satisfied, a new hardware-software partition is created, and the design process is repeated until an acceptable design is determined.

ARIES is an implementation of a hardware-software co-design methodology. The remainder of this section describes its framework and design methodology.

### 1.3.2 The ARIES System Design Framework

The ARIES design framework, shown in Figure 1-5, consists of four interrelated subsystems addressing the various tasks of embedded system design.

1. The high-level specification analysis and partitioning subsystem is responsible for analyzing the system specification and aiding in the task of partitioning the system into a *hardware component* and a *software component.*

2. The ASIC generation subsystem synthesizes the hardware component into an Application Specific Integrated Circuit (ASIC).

3. The ASIP generation subsystem receives the software component of the input application and produces a customized Application Specific Instruction-Set Processor (ASIP) for the given application software. The ASIP design is then transmitted to all subsystems using the Instruction Set Description Language (ISDL).

4. The AVIV code generator receives the ISDL description of the target ASIP and the software component of the input application as input. It generates binary code that implements the software component on the target ASIP.

This framework mimics the typical embedded system structure which consists of an embedded processor, custom software, and custom hardware.

### 1.3.3 The ARIES System Design Methodology

ARIES uses a hardware–software co-design methodology that supports custom ASIP development. A block diagram of the ARIES design methodology is presented in Figure 1-6. The shaded blocks correspond to the software synthesis portion of the design methodology.

A behavioral description of the input application is provided to ARIES using a high-level system specification language such as Scenic [36]. The system specification is analyzed in order to estimate the costs of different implementations of the specified system. Based on these estimates, several implementation options are explored. These options consider various partitions of the application into hardware and software components. The partitioning of the system specification into hardware and software components yields: (1) a behavioral description of the hardware component functionality, and (2) C code describing the functionality of the software component. In addition, an analysis of the software component may provide an initial ASIP architecture specification to the ASIP generation (synthesis) subsystem.

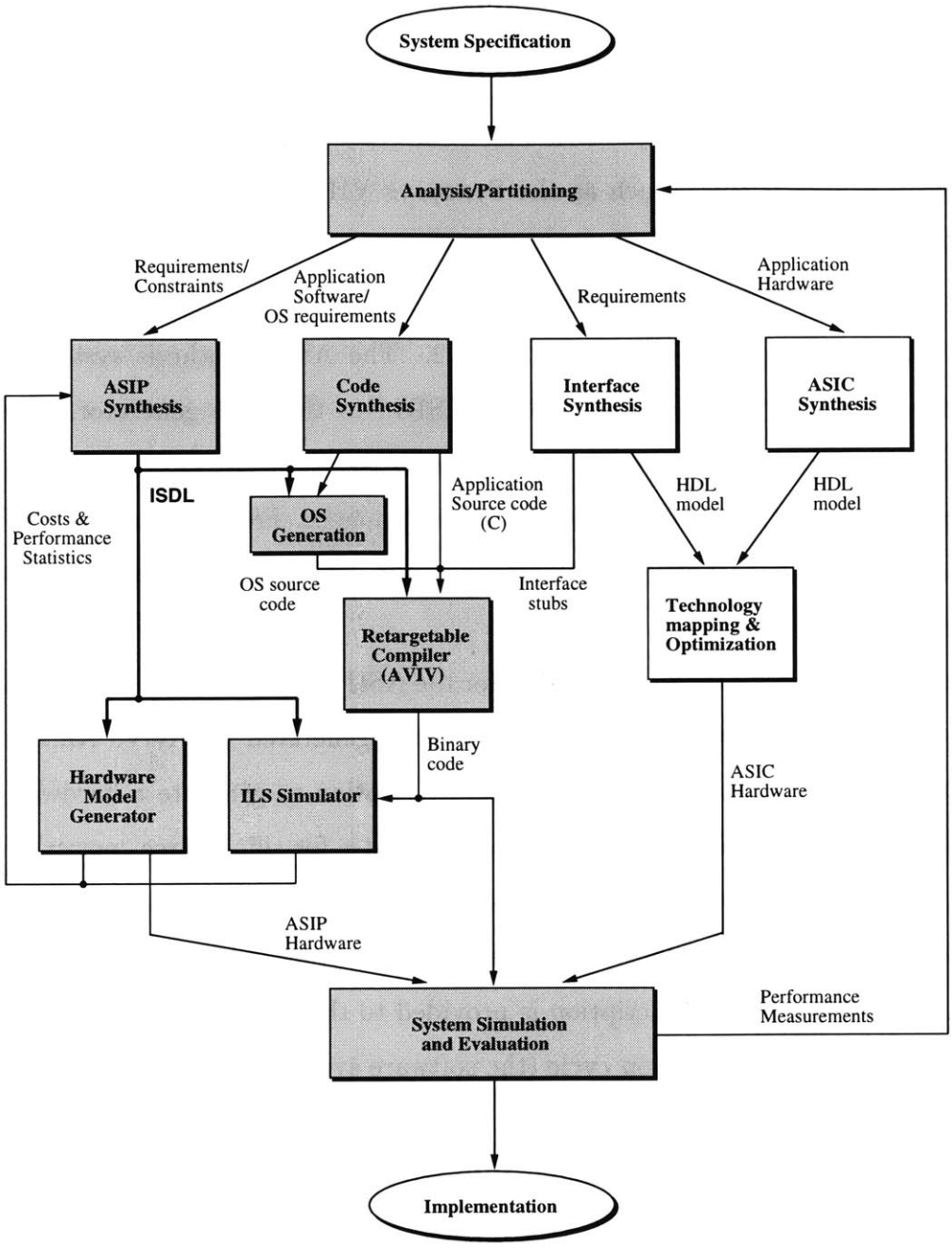



Figure 1-6: **ARIES design methodology**

The behavioral description of the hardware component's functionality is input to a hardware synthesis tool that produces an optimized ASIC implementation of the hardware component. The behavioral description is written in a hardware description language (HDL) such as Verilog [54] or VHDL [48]. The description is then mapped onto a set of hardware library elements specified by the integrated circuit manufacturers. Hardware synthesis has been studied in great depth, and standard hardware synthesis tools, such as the Synopsys VHDL Compiler(TM) or Synopsys HDL Compiler for Verilog [58], may be used to produce the ASIC.

The processor (ASIP) synthesis and code generation steps are intimately related and were described in Sections 1.1 and 1.2. The ASIP synthesis system provides a potential target ASIP, described using ISDL, to the code generator. The AVIV code generator receives the ISDL description of the target processor and the C code describing the functionality of the software component. From this input, it generates optimized code that implements the functionality of the software component on the target ASIP.

An instruction-level simulator (ILS) for the ASIP is created from the ISDL description. The ILS simulates and evaluates the code generated by AVIV. An automated hardware model generator uses the ISDL description to generate a hardware model of the ASIP. The hardware model derives estimates for silicon area, power consumption, and cycle length. The hardware model estimates and the simulation evaluation statistics are used by the ASIP synthesis module in order to refine the target architecture. The refined ASIP description is provided to the code generator, and the ASIP synthesis and code generation cycle (the software synthesis loop) is repeated until no further improvements can be made to the ASIP design.

There are several additional modules that complete the embedded system design. These include an interface synthesis module which creates hardware that communicates between the hardware and software components. The interface hardware is combined with the hardware component ASIC to produce a complete hardware model. In addition, an operating system (OS) generation module creates an OS for the application code. The complete software model includes the operating system

software, the ASIP, and the generated code.

The resulting ASIC, ASIP, and generated code are combined into a complete system model, and the entire design is evaluated to determine if the design constraints have been satisfied. If the design constraints are not satisfied, a new hardware-software partition is created, and the overall design process is repeated until an acceptable design is obtained.

## 1.4 Contributions of this Thesis

This thesis presents the AVIV retargetable code generator. The AVIV code generator is a working system that automatically generates optimized code that implements the functionality of a given input application on a specified target processor. AVIV focuses on producing high quality code for architectures with instruction-level parallelism. Its optimization functions target minimum code size in order to reduce the size requirements of the on-chip program ROM of the embedded system.

The most significant contribution of AVIV is its approach to addressing the instruction selection, resource allocation, and scheduling tasks of code generation concurrently. AVIV focuses on this problem because solving each code generation task independently produces suboptimal results. The code generation steps are interdependent; thus, only the concurrent consideration of the trade-offs between these tasks has the potential of finding a globally optimal solution.

In order to address the various tasks of code generation concurrently, AVIV first transforms each block of code in the input application into a new graphical representation called the Split-Node DAG. The Split-Node DAG explicitly represents all possible ways that the input application can be implemented on the target processor. In order to reduce the complexity of the code generation tasks, AVIV uses a heuristic covering algorithm to produce optimized machine code from the Split-Node DAG.

In addition to presenting the AVIV retargetable code generator, this thesis also presents the ISDL machine description language which was developed to support the communication between the ASIP generation module and the AVIV retargetable code

generator.

## 1.5 Organization of this Thesis

This chapter began with an introduction to embedded system design and motivation for the study of retargetable code generation. It motivated the study of retargetable code generation by describing the software component of embedded systems in detail, stressing that in order to be able to explore the architectural design space of the ASIP, a retargetable code generation environment is required. Next, an overview of the AVIV code generation framework and methodology was provided in order to briefly illustrate the various components of the code generation process and to provide a glimpse into the topics covered in this thesis. A description of an iterative design solution, referred to as the software synthesis loop, was then presented. This iterative solution addresses the close relationship between the design of the target ASIP architecture and the code generation process. Finally, the ARIES system was presented as a unified framework for the automated design of mixed hardware and software embedded systems. The AVIV retargetable code generator is one element of this system. The rest of this thesis describes each component of the AVIV code generator in greater detail.

Work related to the issues addressed by the AVIV retargetable code generator is presented in Chapter 2. The related work is divided into three areas. The first examines previous attempts to address multiple tasks of code generation concurrently. The second describes related work on retargetable code generation systems and how they compare to AVIV. The third presents previous work on machine description languages.

Chapter 3 provides a detailed description of the ISDL machine description language. Chapter 4 describes the structure of the Split-Node DAG. Chapter 5 then explains the method used to cover the Split-Node DAG with target processor instructions. Chapter 6 describes the handling of control flow in AVIV.

In the course of AVIV's development, several tools were built to assist in the code generation process. These include the constraints checker, described in Chapter 7,

and an assembler generator, described in Chapter 8.

Chapter 9 presents results obtained by running the AVIV code generator on various code segments with varying target processor architectures. Chapter 10 concludes this thesis and proposes directions for future enhancements of the retargetable code generator.

# Chapter 2

# Related Work

This chapter presents previous work related to retargetable code generation. It focuses on work in three problem areas. The first area concerns addressing multiple tasks of the code generation problem concurrently. This problem has been approached using various techniques. Most techniques only group a subset of the code generation tasks (e.g., register allocation and scheduling, but not instruction selection). However, a few methods attempt to group all tasks into a single unified solution environment.

The second area of related work addresses the design of retargetable code generators. Various systems have been designed to address the problem of retargeting a code generator to new architectures. In order to address the complexity of the various tasks involved in code generation, each system uses a set of heuristics to solve the code generation problem in a reasonable amount of time. The systems developed for retargetable code generation vary in: (1) the format in which the target processor and input application are described, (2) the internal data structures used to model the code generation problem, and (3) the heuristics used to produce optimized code.

The third area of related work concerns the format in which the capabilities of the target processor are described to the code generator. Many machine description languages have been developed for this task. However, the goals of the various languages are not always the same. As a result, the languages differ in the level of specification (e.g., structural versus behavioral), in the structure of the descriptions, and in the details that are explicitly provided.

## 2.1 Related Work on Addressing Multiple Tasks of Code Generation Concurrently

Previous studies have indicated that performing the various phases of code generation as independent steps leads to inefficient code because the phases are interdependent [20]. This suggests that in order to generate optimal code, the various phases of code generation should be solved concurrently. Numerous efforts that attempt to couple multiple code generation phases exist in the literature and a representative set is summarized below.

### 2.1.1 Integrating Register Allocation and Instruction Scheduling

Bradlee *et al.* [6] studied the effect of integrating register allocation and instruction scheduling for RISC (Reduced Instruction Set Computer) processors. The study compared the performance of three strategies: (1) a simple *postpass* strategy in which global register allocation is performed prior to instruction scheduling; (2) a variation of Integrated Prepass Scheduling (IPS) in which the scheduler is invoked prior to register allocation, and it must schedule within a local register limit; (3) a new technique called RASE (Register Allocation with Schedule Estimates) that integrates register allocation and instruction scheduling by giving the register allocator cost estimates that quantify the effect of its allocation choices on the subsequently generated schedule. The three strategies were developed as part of the Marion Code Generator Construction System [7] which was designed to produce retargetable code generators for RISC processors.

The postpass strategy cleanly separates register allocation and instruction scheduling. It serves as a basis for comparison to techniques that couple these two phases.

The second strategy is a variation of IPS. In IPS [19], the scheduler attempts to find a schedule that does not exceed a specified local register limit. If the register limit cannot be satisfied, then spills must be inserted by the register allocation step which

follows the scheduling step. There are several differences between the original IPS and the variation used by Bradlee *et al.* The first distinction is that the original IPS technique performs global register allocation before running IPS; therefore, only local register allocation is performed after scheduling. In Bradlee's version, the register limit used by the scheduler corresponds to a global register limit rather than a local limit. The second difference is that in the original IPS technique spills are scheduled by the register allocator. In Bradlee's version, a postpass (second) scheduler is invoked after register allocation to reoptimize the schedule with the added spill code.

The third strategy Bradlee *et al.* considered, RASE, couples register allocation and instruction scheduling more closely than the IPS strategy. The close coupling of the two phases is achieved by providing the register allocator with schedule cost estimates that allow the allocator to quantify the effect of its choices on the scheduler. In contrast, IPS only uses a heuristic to attempt to minimize the scheduler's impact on the register allocator.

Bradlee's study concluded that strategies that treat register allocation and scheduling independently result in inefficient schedules. Both RASE and IPS are significantly better than the two phase strategy. These results indicate that some level of integration is necessary to produce efficient schedules.

### 2.1.2 Performing Data Routing and Scheduling Concurrently

In [28], Hartmann presents an algorithm for performing data routing and scheduling concurrently in programmable ASIC systems. Typically, automatic generation of microcode has two phases. The first phase generates sequential vertical microcode that includes the routing of results of an operation from the output of one functional unit to the input of another functional unit (i.e., data routing). In particular, the first phase decides whether an intermediate result is kept in a register or in memory. The second phase compacts the sequence of operations. However, the two phases are interdependent; thus, Hartmann proposes a new scheduler that integrates data routing into scheduling.

Hartmann's scheduler is embedded in the framework of the Cathedral-II [56] sys-

tem. The Cathedral system is used to transform all operations in the application into operations available on the target processor. This includes converting operations not available on the target processor into multiple operations whose combination is functionally equivalent to the original operation. In addition, Cathedral decides whether or not complex operations should be used to cover the input application. Hartmann's scheduler then performs scheduling and data routing on the representation it receives from the Cathedral system. This scheduler assumes that each operation has already been assigned to a particular functional unit.

Hartmann's scheduling scheme is complicated by the fact that poor scheduling decisions can lead to deadlock[1]. Therefore, a check for deadlock must be made for each decision made by the data router. The scheduler loops through all of the operations, giving priority to operations whose operands are not located in memory (these operations utilize scarce register resources). The operation is scheduled in the current cycle provided that: (1) the required resources for the current operation are available, (2) scheduling the operation will not result in deadlock, and (3) the data router was successful in finding a route. If no further operations can be scheduled in the current cycle, then the process is repeated for the next cycle until all operations have been scheduled.

### 2.1.3 Unified Resource Allocation for Registers and Functional Units in VLIW Architectures

URSA (Unified ReSource Allocator) [5] is a resource allocator that addresses register and functional unit allocation in a unified manner. The URSA technique operates on a DAG representation of the application program and consists of three phases. The first phase measures the resource requirements of the DAG and identifies regions with excess requirements (requirements that cannot be supported by the target processor).

[1] An example of a situation that leads to deadlock is when two operations require a load of their input variables into the same location. If one of the operands of the first operation gets loaded together with one of the operands from the second operation, then neither operation can continue and deadlock occurs.

The second phase applies transformations that reduce the resource requirements to levels supported by the target processor. The third phase carries out the assignment of resources.

URSA is primarily concerned with the allocation of resources and not their actual assignment. It uses a DAG that represents reuse information to measure the program's resource requirements and to identify regions that have excess requirements. Transformations are applied on these regions to remove the excess resource requirements. In order to reduce register requirements, two possible transformations can be utilized. The first transformation inserts additional dependence edges into the DAG. These edges sequentialize the register usage, resulting in reduced register requirements. The second possible transformation is to introduce load and spill nodes into the DAG. This reduces the number of live variables that must be stored in the registers. In order to reduce the functional unit requirements, the only possible transformation is to insert dependence edges that reduce the parallelism available in the DAG. This results in a reduction in the number of functional units required. Since all three types of transformations operate on the same DAG, they may be applied in an integrated manner. Once the DAG transformations have been incorporated, the resource assignment and code generation phases may be executed.

### 2.1.4 Integrating Code Selection and Register Allocation into Instruction Scheduling

Mutation scheduling [43] provides a unified compiler-based approach for exploiting the functional unit and storage capabilities of a target processor. It integrates instruction selection and register allocation into the instruction scheduling phase of code generation. These tasks are integrated in an attempt to dynamically adapt a given program to optimally match the characteristics of the target architecture.

Mutation scheduling is a "value-oriented" approach to instruction scheduling. This means that it allows the computation of any given value to change dynamically during scheduling in order to conform to varying resource constraints and availabili-

ties. Mutation scheduling accomplishes this by associating each value, defined in the program, with a *mutation set.* A mutation set includes all functionally-equivalent expressions that can compute the given value using different resources of the target architecture. When attempting to schedule the *expression* that currently generates a given value, if the resources associated with that expression are unavailable, then another functionally-equivalent expression can substitute the current expression. The alternate expression, or *mutation*, is selected from the mutation set. The alternate expression selected should better suit the available resources.

Mutation sets are also used to integrate register allocation into the scheduling process by allowing a mutation set to change dynamically during scheduling. This ability permits the addition of new expressions that calculate the given value to the mutation set. Thus, if a value has already been computed and is resident in a register, then a reference to that register will become one of the expressions in the mutation set associated with that value.

The mutation sets and their transformations are integrated into an existing Global Resource-constrained Percolation (GRiP) [42] scheduler to yield a mutation scheduling system. Using GRiP, operations are progressively scheduled earlier until the scheduler is blocked by resource dependencies, true data dependencies, or false data dependencies (e.g., when no free registers are available). Whenever one of these dependencies is encountered, the mutation set is used to attempt to remove the dependency. When trying to find a new mutation for values that cause true data dependencies or functional resource dependencies, the new mutation should be one that can be scheduled earlier. Scheduling the value earlier allows expressions that depend on that value to be scheduled more easily. The mutation set transformations can also be used to free registers when the register resources become unavailable.

### 2.1.5 Integrating Instruction Selection, Resource Allocation, and Scheduling for Heterogeneous Register Machines

In [59], Wess presents a new methodology for the efficient code generation of expression trees for heterogeneous register set machines. This methodology uses *trellis* trees in which each operation node of the expression tree is replaced by its equivalent trellis diagram. A trellis diagram represents an instruction as a connection between all its source and destination operand nodes. A trellis diagram explicitly models all possible register sources, destinations, and combinations thereof. The source and destination nodes define the state of a trellis diagram. Each state can represent a single register, an arbitrary memory location, or a combination of possible registers. A state that contains multiple heterogeneous registers designates that the operand can come from (in the case of a source operand) or go to (in the case of a destination operand) any of the storage locations represented by that state.

By traversing a trellis tree bottom-up, optimal normal form programs can be generated in time that is linearly dependent on the size of the trellis tree. This is achieved by first calculating the minimal state cost for each state. The cost of a state is defined as the cost of the instruction that utilizes the operands corresponding to that state. Then, all edges and connections which do not result in minimal state cost are removed. Optimal normal form programs correspond to paths in the reduced trellis tree.

This method of code generation integrates scheduling, register allocation, and instruction selection into one optimization algorithm that produces optimized vertical code. However, compaction (the process of merging multiple operations into single parallel instructions) is performed after the previous portions of code generation have been completed. It is possible that instruction selection decisions made in covering the trellis trees with vertical code may not be the best decisions for a machine with a high degree of parallelism. Thus, this algorithm is not well suited for orthogonal architectures such as VLIW architectures.

### 2.1.6 Summary and Comparison to AVIV

The work described in this section summarized several prior attempts to address multiple code generation tasks in a unified manner. The first three projects provided motivation and algorithms for concurrently addressing two subtasks of the code generation process. In particular, Bradlee *et al.* found that some integration of the register allocation and scheduling tasks is necessary to produce efficient schedules. Hartmann introduced an algorithm for performing data routing and scheduling concurrently. The URSA project addressed the problem of performing register and functional unit allocation in a unified manner. The last two projects addressed the tasks of instruction selection, resource allocation, and scheduling concurrently. The mutation scheduling scheme is similar to AVIV in that it addresses the various phases of code generation concurrently, but it uses different heuristics. Wess's work using trellis trees, however, does not address the compaction phase of code generation until after vertical code has been generated.

AVIV builds on this previous work by addressing all subtasks in a unified manner. This includes addressing instruction selection, register and functional unit resource allocation, data routing, scheduling, and compaction concurrently. AVIV focuses on generating optimized code for architectures with instruction-level parallelism. In order to make use of the parallelism available on such architectures, it is imperative to address the compaction task of code generation together with the other code generation tasks.

## 2.2 Related Work in Retargetable Code Generation

Several projects have been developed to respond to the growing need for retargetable code generators in the design of embedded systems. Retargetable code generators attempt to fully automate the process of translating a high-level software application into optimized code that can run on a specified target processor. Issues involved in

the design of retargetable code generators range from the level of specification of the target processor (e.g., structural or behavioral descriptions) to the type of heuristics used to try to solve the code generation problem optimally. A representative set of these research efforts is summarized in this section.

### 2.2.1 The Record Compiler Generator

The RECORD compiler generator [34, 38] compiles programs written in the DSP-specific programming language, DFL [39], into binary code. RECORD compilers are generated from a structural (netlist) description of the target processor provided in the MIMOLA HDL [4]. The benefit of a structural description, such as a register transfer level (RTL) netlist, as opposed to a behavioral description, is that a structural description simplifies the analysis of architectural trade-offs. However, in order to generate code from a structural description, the instruction set must first be extracted from the description. The RECORD compiler uses the Instruction Set Extraction (ISE) method described in [33] to generate an instruction set (behavioral) description from an RTL netlist. Once the instruction set description is available, instruction selection is performed. Instruction selection is the process of selecting a set of target processor instructions to cover the instructions of the input application. In order to perform instruction selection, the target processor and the input application instructions are represented as instruction patterns. The RECORD compiler then uses the IBURG [16] tool set to match the target processor instruction patterns to the patterns representing the input application. The IBURG tool set supports the automatic generation of pattern matchers for any given target instruction set. After instruction selection is completed, an optimization step that attempts to compact multiple operations into parallel instructions is performed.

There are several differences between the RECORD compiler and the AVIV code generator. The first is that the RECORD compiler begins with a structural description of the target processor rather than a behavioral description. Behavioral descriptions are preferable for code generation because they explicitly model the instructions available on the target processor. The RECORD compiler includes an additional phase of

code generation that extracts the instruction set from the structural description. Although behavioral descriptions are preferable for code generation, structural descriptions may simplify the synthesis of the target architecture hardware. Another distinction between the two compilers is that the RECORD compiler addresses the various phases of code generation independently (i.e., code compaction and register spill minimization are performed after instruction selection) whereas AVIV addresses them concurrently in order to find a globally optimized solution.

### 2.2.2 FlexWare

FlexWare [46] is a software/firmware development environment for ASIPs and commercial processors. It is composed of an instruction set simulator, INSULIN [52], and a retargetable code generator, CODESYN [45]. INSULIN provides a cycle-true VHDL-based simulation environment. CODESYN takes an algorithm written in a high-level language and maps it into the target instruction set to produce optimized machine code for ASIPs and commercial processor cores.

The target processor is described to CODESYN using a mixed-level model that includes a behavioral level description of all possible partial instructions (i.e., microinstructions), an abstract netlist describing the data path topology, and a definition of the register classes.

In CODESYN, code generation begins by converting the high-level source program into a hierarchy of Control-Data Flow Graphs (CDFGs). A pattern matching phase then determines the possible set of target processor instructions that can perform the functions of the CDFG. This is performed by trying to match the partial instructions (described as small CDFGs) to portions of the source CDFG. The number of attempted matches is minimized by organizing the partial operation CDFGs into a hierarchical tree-like structure [37]. This structure is organized such that if a partial operation pattern does not match a portion of the CDFG, then it is possible to prune the tree of possible matches at that point. It is guaranteed that no matches are possible beyond that point because any pattern further down the tree is a superset of the patterns above it.

After the set of all possible target processor matches have been identified, a dynamic programming [1] technique is used to select the best instructions to cover the source CDFG. Once instruction selection is completed, register allocation can be performed. At this stage, if any microoperation can be executed in parallel with another microoperation, then the multiple operations are compacted into a single horizontal instruction. The last step is to assemble the instructions into their binary representation.

The main difference between CODESYN and AVIV is that, like the RECORD compiler, the CODESYN methodology addresses the various phases of code generation independently. AVIV, on the other hand, focuses on determining a globally optimized solution by considering instruction selection, resource allocation, and scheduling concurrently. Furthermore, the target processor description used by CODESYN is not the same description used by INSULIN. ISDL, on the other hand, is used to describe the target processor to all the design tools in the software synthesis portion of Project ARIES.

Another distinction between the two approaches is that CODESYN does not consider the required register-to-register moves or spill-to-memory operations until after instruction selection has been completed. This implies that during register allocation it may be necessary to add such data transfer operations in order to place the data in the correct location. AVIV embeds all required data transfer operations directly into the Split-Node DAG representation (AVIV's version of the CDFG). As a result, in covering the Split-Node DAG, the required data transfer operations will automatically be considered when evaluating the trade-offs between multiple operations that execute the same function on different hardware resources. In addition, the data manipulation operations and data transfer operations are covered simultaneously. This optimizes the schedule from the start to include all required operations.

### 2.2.3 Chess

CHESS [31] is a retargetable code generation environment for fixed-point DSPs and ASIPs. It generates machine code for the target processor, described in the nML

language [14, 17], and provides feedback as to how well suited the target processor is for the given application.

The nML target processor description is translated into an Instruction-Set Graph (ISG) which is a mixed structural and behavioral representation of the processor. The skeleton of the ISG is defined by the storage resources of the target processor. It includes static storage (i.e., memories, and registers) and transitory storage (i.e., storage that only passes a value but does not retain it). The ISG models connectivity of the microoperations to the storage resources of the target processor. Each operation is linked to a predefined primitive that describes the behavior of the operation. In addition, each operation has a list of instruction strings associated with it. These strings represent the valid instruction-bit setting enabling that operation. Thus, the ISG models connectivity and encoding restrictions, as well as structural hazards. This model is used for all phases of code generation.

The code generation process first translates the input algorithm into a CDFG. It refines the CDFG so that all nodes in the CDFG correspond to microoperations that can be implemented by the target processor. Code selection then covers the CDFG with patterns, called bundles, that correspond to partial instructions supported by the instruction set. Rather than making an exhaustive list of all possible bundles, it bundles instructions on the fly by searching for valid paths in the ISG. A path is valid when the intersection of all the instruction strings along that path is non-empty. The code selection phase is followed by a register allocation phase that assigns storage locations to values and determines whether any data transfer operations are required. If there are insufficient register resources, then several intermediate values are spilled to memory. Next, a scheduling step is performed. It compacts the selected partial instructions, data transfer operations, and control-flow operations into a single microprogram. The scheduling step is followed by an assembly phase that converts the assembly instructions into machine code.

In CHESS each phase of code generation is performed separately. However, in order to ensure phase coupling, an intermediate scheduling view that accounts for the resource constraints imposed by the previous phase is constructed after each phase.

There are several differences between CHESS and AVIV. The first is that CHESS is driven by nML in which all legal groupings of operations must be explicitly listed. In AVIV, the target processor is described using ISDL. ISDL supports the explicit description of constraints that specify which operation groupings are illegal. As a result, it is not necessary to list all legal groupings of operations. Sections 2.3.2 and 3.5 compare these two description languages in greater detail. The second difference between the two code generators is that in CHESS each phase of code generation is performed separately, though not independently, whereas in AVIV, the phases are performed concurrently. The third difference is that although CHESS supports both orthogonal (e.g., VLIW) and encoded instruction formats, its optimization functions are geared towards encoded instruction formats. The optimizations utilized by AVIV focus on orthogonal instruction sets.

### 2.2.4 Wilson et al.

Wilson et al.'s Integer Linear Programming (ILP) based approach to code generation [61] is based on a behavioral model of the target processor. Similar to FLEXWARE and CHESS, code generation begins by translating a high-level source language into a Data-Flow Graph (DFG). Pattern matching is used to recognize complex instructions. Next, a potential schedule that attempts to minimize overhead costs, such as spills to memory and address calculations, is identified. Register assignment, including necessary spills to memory, is then performed.

The cornerstone of all optimizations is an ILP solver that can simultaneously perform scheduling, instruction selection, register assignment, and compaction. In addition, the ILP solver can choose between alternative spill and addressing candidates. Since all constraints are considered simultaneously in the ILP formulation, trade-offs can be made between the various optimizations leading to a globally optimal solution.

Running an ILP model on a large problem may result in a runtime that is longer than the potential benefits warrant. In order to shorten the runtime, the ILP solver is only applied after considerable preprocessing. Furthermore, it is only applied in a strategic manner. This is accomplished by: (1) making preliminary decisions that are

too expensive for the ILP to consider; (2) considering only a limited number of blocks at a time; and (3) having the ILP find a number of successively more constrained feasible solutions rather than attempting to solve the entire problem at once.

The problem with using ILP solvers for code generation is that finding the optimal solution is far too CPU intensive. Since the ILP solver does not have sufficient information about the structure of the problem, it cannot make intelligent decisions about how to prune the search space. In general, user-supplied hints are required to produce good code within a reasonable amount of CPU time.

Like ILP solvers, AVIV addresses the problem of solving instruction selection, resource allocation, and scheduling in a unified manner in order to find a globally optimal solution. This is achieved by converting the input application into a set of Split-Node DAGs that model all possible ways of implementing the operations of the input application on the target processor. The Split-Node DAG representation together with AVIV's heuristics for covering the Split-Node DAG allow the three tasks of code generation to be addressed simultaneously. The Split-Node DAG representation of the input application provides sufficient information to make intelligent decisions in pruning the search space of possible code generation solutions. As a result, AVIV is able to identify globally optimized solutions within a reasonable amount of CPU time.

## 2.3 Related Work on Machine Description Languages for Embedded Processors

A machine description language is needed in order to describe the capabilities of the target processor to a retargetable code generator. The languages that have been developed for this task differ in their level of specification, as well as in the information that is explicitly described versus the information that must be inferred from the description. In order to illustrate these differences, this section presents a brief overview of three representative machine description languages: MIMOLA, nML, and LISA.

### 2.3.1 Mimola

The MIMOLA [4] hardware description language uses a structural (netlist) rather than behavioral (instruction-set) model to describe the target processor. A structural model describes the processor at a substantially lower level than a behavioral model. MIMOLA follows the structure of standard hardware description models that explicitly define the connections between multiple modules. Each module can be defined at the RTL or gate level. The benefit of a structural model is that it can be easily simulated with an RTL structural simulator. In addition, it is much simpler to synthesize hardware from such a low-level description. However, it is difficult to describe the instruction-set of a target processor using a low-level description model. As a result, it is difficult to compile code from a MIMOLA description of the target processor.

### 2.3.2 nML

The nML language [13] is attractive because it allows the user to specify the target architecture in a way that parallels instruction-set descriptions found in a user's manual. In contrast to MIMOLA, the machine description contains behavioral as well as structural information. The structural information describes the storage capabilities, and the behavioral information describes the instruction-set. The instruction-set description is partitioned into rules. These rules form a grammar that represents all valid instructions of the target processor.

nML is very similar to the ISDL description language in the way it defines valid instructions. The main difference between the two languages is that nML does not support the description of constraints that specify which operation combinations lead to invalid target processor instructions. ISDL, on the other hand, explicitly lists all of the constraints of the target processor. The lack of support for the explicit description of constraints in nML implies that only valid instructions can be described. This means that nML can result in significantly larger and less intuitive machine descriptions than ISDL.

### 2.3.3 Lisa

The LISA language [62] was developed as a unified description language that could be used for compilation, simulation, and hardware generation. However, its focus has been on supporting the generation of fast compiled-code simulators that are bit-true and cycle-accurate. Its main characteristic is its operation-level description of the pipeline that is able to model complex interlocking and bypassing techniques. Although LISA is well suited for generating simulators, its description of available operations is not explicit. Instead, instructions are partitioned into schedulable units. In order to determine the admissible operations, precedence and resource constraints must be analyzed. This makes it more difficult to infer the capabilities of the target processor, and therefore makes the code generation process more complex.

### 2.3.4 Summary and Comparison to ISDL

None of the languages mentioned above provide support for explicit constraints. Without explicit constraints, descriptions for architectures with Instruction Level Parallelism (ILP) become very tedious to write because every legal combination of operations must be explicitly listed. The ISDL machine description language (presented in Chapter 3) explicitly defines all of the constraints of the target processor. Section 3.5 elaborates on the use of constraints to simplify the machine description of a target processor. In addition to the fact that constraints simplify the machine description, deriving a set of constraint clauses, in a form usable by the compiler, is very difficult. This implies that it is preferable to explicitly list the constraints in the machine description.

# Chapter 3

# ISDL: Instruction Set Description Language

The ISDL machine description language [23, 24, 22] was designed to aid in the generation of tools required for automated embedded system design. As illustrated in Figure 3-1, ISDL is the focal point of all the tools used for ASIP architecture exploration. In order to support the automated design of the software component of embedded systems, ISDL must:

- specify a wide variety of processor architectures including VLIW (Very Long Instruction Word) architectures,
- explicitly support constraints that define valid instructions,
- be easily understandable and modifiable by a compiler developer or hardware architect,
- support automatically retargetable code generation,
- support the automatic generation of an assembler and disassembler,
- support the automatic generation of a cycle-accurate instruction-level simulator,
- support the generation of an implementation of the architecture from the machine description, and

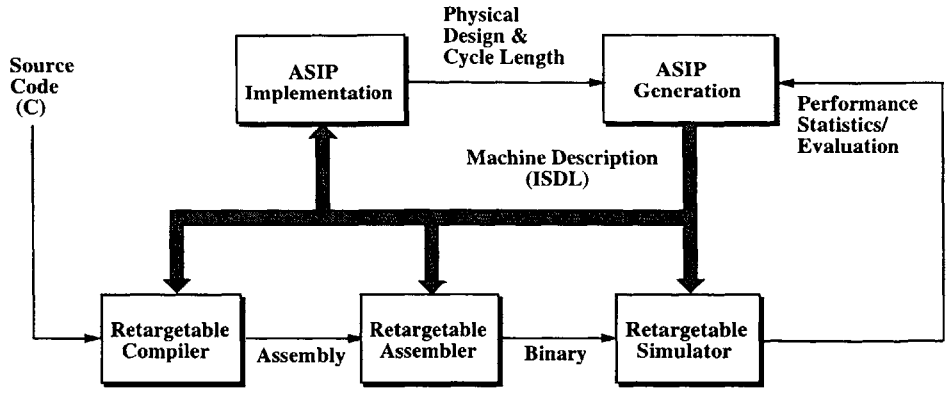



Figure 3-1: **The design flow for an ASIP**

- provide adequate information to allow for code optimizations.

ISDL was developed to include all of the features listed above. It has been used to describe ASIPs, as well as commercial Digital Signal Processor (DSP) cores. In particular, a powerful ASIP VLIW architecture and the Motorola 56000 DSP [41] have been described in ISDL.

This chapter begins by describing how ISDL models an instruction set. A detailed description of the structure of an ISDL description is then provided. The ISDL structure is clarified through a step-by-step explanation of a sample ISDL description. The chapter concludes with a detailed example of how constraints simplify machine descriptions.

## 3.1 Definitions

The following definitions are useful in clarifying the presentation of ISDL.

**VLIW (Very Long Instruction Word)** - A VLIW architecture is an architecture that exhibits instruction-level parallelism (i.e., a single instruction controls multiple functional units, and the functional units can operate independently and in parallel).

The VLIW class of architectures does not include architectures in which parallelism is *not* explicit in the instruction set (e.g., super-scalar architectures). VLIW encom-

passes architectures whose parallelism includes parallel data transfers in addition to parallel data manipulations. Each functional unit in a VLIW architecture typically has its own field in the instruction resulting in very long instruction words.

**Unifunctional** - A unifunctional architecture is an architecture that exhibits no instruction-level parallelism, (i.e., the instruction can only control one functional unit at a time).

This class of architectures includes architectures in which parallelism is *not* reflected in the instruction set.

**Operation** - An operation is the smallest part of a program that can specify a data manipulation independently.

In the case of VLIW architectures, a separate operation controls each of the functional units, and the VLIW instruction may consist of multiple operations.

**Instruction** - An instruction is the smallest unit of control that can be fetched and issued to the hardware.

In the case of VLIW architectures, an instruction may consist of a number of generally independent *operations*, each of which controls a functional unit.

**Operation Orthogonality** - An instruction set exhibiting operation orthogonality is one in which the presence or form of one operation in an instruction is independent of the presence or form of any other operation in the instruction.

Although many operations in the instruction set of a VLIW processor are orthogonal, they are generally not completely orthogonal.

## 3.2 ISDL Model of the Instruction Set

Conceptually, the instruction set of a processor consists of: (1) the state available in the architecture, and (2) the instructions that modify this state. In order to describe the instruction set of the processor, a machine description language must describe

these two components. In ISDL there is a straightforward mapping from the storage capabilities of the processor to its state. The description of the instructions is not as straightforward and is described below.

The processor is capable of recognizing and executing a number of complete instructions. However, these complete instructions are not explicitly listed in ISDL. Instead, the instructions are described in terms of their parts, or operations. In order to determine the set of instructions that the processor supports, it is necessary to group operations from the ISDL description into valid instructions.

ISDL divides the set of available operations into fields. The operations in a given field correspond roughly to the various functions that a single functional unit can perform. Thus, operations defined within the same field are mutually exclusive and cannot appear in the same instruction. Fields correspond roughly to the functional units of the processor, and, as a general rule, these units can operate in parallel. Therefore, each instruction consists of a group of operations – one taken from each field.

The set of all possible combinations of operations formed by taking one operation from each field is a good approximation, and a superset, of the set of instructions available in the architecture. Some of these combinations are illegal and cannot be executed in hardware. These illegal combinations are represented as constraints of the target processor and are explicitly described in ISDL. The constraints declare a subset of all possible combinations as legal. This subset is the set of instructions in the architecture. The constraints effectively form a filter which when applied to the set of possible combinations of operations results in the set of valid instructions. Figure 3-2 illustrates this mapping function.

## 3.3 The Structure of an ISDL Description

ISDL is able to describe a wide variety of processor architectures including VLIW architectures, standard microcontrollers, and custom datapath DSP cores. Specifically, ISDL can describe architectures with multiple functional units, different inter-

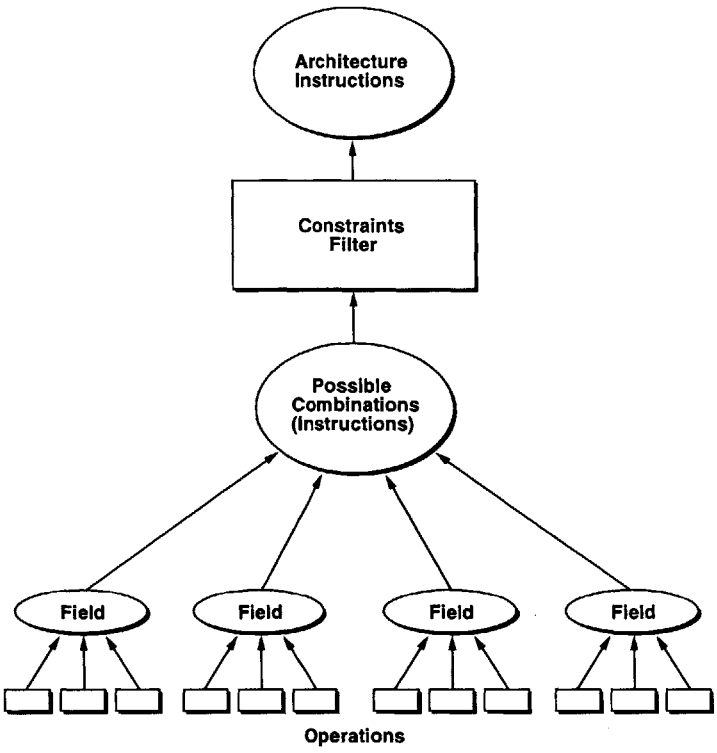



Figure 3-2: **ISDL model of instructions**

connect topologies, complex instructions, resource conflicts, pipelining idiosyncrasies, etc. ISDL can also describe automatically generated architectures. Such architectures cannot be guaranteed to have clean instruction sets (i.e., instruction sets where every operation combination is valid). In order to describe these instruction sets, ISDL supports explicit constraints that define the valid operation groupings. This allows operations in the instruction set to be treated as if they are completely orthogonal. The compiler can then avoid generating invalid instructions by ensuring that each instruction satisfies all of the constraints. Note that constraints are not an artifact of automatically generated architectures. In fact, many commercial architectures also include constraints in their instruction sets (e.g., the Motorola 56000 cannot perform a `Move` to the top of the hardware stack within the last three instructions of a `DO` loop).

An ISDL description consists of the following six sections:

1. Instruction Word Format

2. Global Definitions

3. Storage Resources

4. Instruction Set

5. Constraints

6. Optional Architectural Details

Each section is described below, and a detailed ISDL description of a sample processor is provided in Section 3.4. The ISDL user's manual [22] provides a complete description of ISDL including syntax specifications.

### 3.3.1 Instruction Word Format

The *Instruction Word Format* section defines the structure of the hardware instruction word. The hardware instruction word refers to the binary representation of the instruction. The instruction word is separated into multiple fields each containing one or more subfields. The field and subfield division of the instruction word, as well as the bitwidth of each subfield are provided in this section. The instruction word is assembled by concatenating all subfields in the specified order beginning with the most significant bit.

Note that the division into subfields is a convenience provided to the designer. The subfield division may be arbitrary; however, careful subfield division can make subsequent parts of the machine description easier to write.

### 3.3.2 Global Definitions

The second section of an ISDL description contains a list of definitions used in subsequent sections. These definitions consist of three types: *Tokens*, *Non-terminals*, and *Split functions.*

Tokens and non-terminals are abstractions used to simplify the assembly syntax definitions of the ISDL operations. Split functions describe how long constants, that

need to be included in the instruction word, can be divided across multiple subfields of the instruction word.

### Tokens

*Tokens* are a symbolic representation of the assembly syntax of the target processor. Tokens are used to represent entities such as register names, memory bank names, and immediate constants. Tokens can also be used to group syntactically related entities. In order to differentiate among the elements in a group, tokens return a value identifying the particular element being represented (e.g., register names such as R0 through R15 can be abbreviated as one token whose value corresponds to the register number). A token definition contains a name for the token, a definition of the assembly syntax for the syntactic entities it represents, and a return value (if there is one). The token name can be used to refer to the token in non-terminal and operation definitions.

The assembly syntax of each ISDL operation can only be described using tokens, non-terminals, and punctuation characters. In order to avoid having to explicitly define tokens that are likely to appear in the assembly syntax, there are some tokens that are automatically defined. These include operation names as well as tokens that represent integers, hexadecimals, floating point numbers, single characters, and labels (symbolic names that represent instruction memory locations).

### Non-Terminals

*Non-terminals* are the main source of abstraction in ISDL. They are typically used for two purposes:

1. to group multiple entities such as tokens or other non-terminals into a single syntactic entity, and

2. to factor out common patterns representing partial operation definitions (e.g., addressing modes) from operation and other non-terminal definitions.

For example, consider a `Move` operation that moves data across a bus that has seven units attached to it:

```
Move SRC DEST
```

If `SRC` and `DEST` each represent seven different options, then there are 49 valid move operations. Factoring out `SRC` and `DEST` into non-terminals allows all valid move operations to be specified concisely using just one move operation definition and two non-terminal definitions (one for `SRC` and another for `DEST`).

Non-terminal definitions consist of a name followed by a list of options that the non-terminal can represent. The non-terminal name can appear in the assembly syntax definition of any operation or any other non-terminal. The inclusion of a non-terminal name in an assembly syntax definition implies that the name can be replaced by any option that the non-terminal represents. A non-terminal option contains the following six components:

- **Assembly Syntax** - The assembly syntax describes the assembly representation of the non-terminal option. It can refer to tokens or other non-terminals.

- **Return Value** - The return value identifies the non-terminal option. This value can be a function of the return values of tokens and non-terminals referred to in the assembly syntax for this option. The return value can be used in operation bitfield assignments (Section 3.3.4).

- **RTL Action Clause** - An RTL action clause describes the RTL (register transfer level)[1] equivalent of the non-terminal option. When the non-terminal option is referenced in the RTL portion of an operation definition (Section 3.3.4), the reference is replaced by the non-terminal's RTL action clause. The RTL action clause can refer to the return value of tokens or the RTL action clause of non-terminals referred to in the assembly syntax for this option.

[1] An RTL specification describes the effect of an operation on the storage elements of the target processor. It describes the effect of the operation at a low level which directly refers to the registers and memories of the processor.

- **RTL Side-Effects Clause** - The RTL side-effects clause follows the same structure as the RTL action clause but it describes side-effects.

- **Cost Modifier Clause** - The cost modifier clause contains a set of expressions describing the effect of the non-terminal option on the operation costs. The cost expression can refer to the return value of the tokens or the cost modifier clause of the non-terminals referred to in the assembly syntax for this option.

- **Timing Modifier Clause** - The timing modifier clause follows the same structure as the costs modifier clause but it describes the timing parameters.

#### Split Functions

As described above, the binary instruction word consists of a set of fields each of which consists of several subfields. Each subfield represents a subset of bits, a *bitfield*, in the instruction word. The operation definitions assign values to these bitfields in order to identify the operations that are being represented by the instruction word. However, there are occasions when the binary data that must be embedded in the instruction word is longer than any of the specified bitfields. This can occur when a memory address or immediate data must be included in the instruction. In such cases, multiple bitfields are used to represent a single value. The division of a long bitfield into a set of the predefined instruction word subfields is described using *split function* definitions.

Split function definitions are used to automatically create functions that can take a long bitfield and split it into existing subfields of the instruction word. These functions can then be used in non-terminal return value expressions and bitfield assignment commands (Section 3.3.4) in order to assign the appropriate values to the instruction word subfields.

### 3.3.3 Storage Resources

The *Storage* section lists all storage resources visible to the programmer. It lists the names and sizes of the memories, register files, and special registers.

A storage definition consists of the type of storage, a name for the storage unit, and the size of the unit (width in bits for single registers, depth in locations and width in bits for addressed units). The instruction memory and program counter must be identified explicitly.

ISDL recognizes the following types of storage units:

- `Instruction Memory` - The instruction memory stores the instructions to be executed.
- `Memory` - A memory stores the data used by the instructions.
- `RegFile` - Register files store temporary data.
- `Register` - Single registers store temporary data.
- `CRegister` - This storage unit type represents control and status registers. Control and status registers have side effects when a value is written to them (e.g., may cause a change in processor mode), and they do not necessarily return the last value written to them when read (e.g., status of peripherals).
- `Stack(SP)` - This storage unit type represents hardware stacks. Hardware stacks are indexed by the Stack Pointer (`SP`). The stack pointer must be declared as a single register within the storage section.
- `MMIO` - Memory mapped I/O ports may have side effects when written and do not necessarily return the last value written to them when read.
- `ProgramCounter` - The program counter must be explicitly declared.

### 3.3.4 Instruction Set

The *Instruction Set* section is divided into *fields* corresponding to the multiple operations that can be performed in parallel within a single instruction. This division supports the description of VLIW architectures. Each field definition consists of a number of *operation* definitions that the corresponding functional unit can perform.

Each operation definition consists of the following elements:

- **Assembly Syntax** - This declares the assembly syntax of the operation. It consists of an operation name followed by a list of parameters. The parameters may be punctuation characters and/or the names of tokens or non-terminals.

- **Bitfield Assignments** - The bitfield assignments are a set of statements that assign the appropriate values to the subfields defined in the instruction word format section. These statements may make use of the return values of the tokens and non-terminals in the operation's parameter list. If the operation being described requires *additional* binary words to represent long constants (such as the target address of a jump operation), then the bitfield assignments can include assignments for the subfields of the additional binary words as well.

- **RTL Action** - This describes the effect of the operation on the storage resources in an RTL type language. It may make use of the return values of tokens and the RTL action clause of non-terminals appearing in the operation's parameter list.

- **RTL Side-Effects** - This describes any side-effects of the operation using the RTL language. It may make use of the return value of tokens and the RTL side-effects clause of non-terminals in the parameter list. The difference between RTL side-effects and the RTL action is that the side-effects happen after the RTL action is executed.

- **Costs** - Multiple costs are permitted. These include operation execution time, code size, and costs due to resource conflicts. ISDL requires three predefined costs: `Cycle`, `Size`, and `Stall`. The cycle cost declares the number of cycles the operation requires to execute on the hardware. The size cost declares the number of instruction words needed by the operation. The stall cost declares the number of stall cycles that will be inserted if the next instruction attempts to use the result of the operation.

- **Timing** - This information describes when the various effects of the operation take place (e.g., the effects of pipelining). ISDL requires two predefined timing

parameters: `Latency` and `Usage`. The latency parameter specifies the number of instructions (including the current one) that must be fetched before the result of the current operation becomes available. The usage parameter specifies the number of instructions (including the current one) that must be fetched before another instruction can assign an operation to the functional unit associated with the described operation.

### 3.3.5 Constraints

The Instruction Set section describes a number of fields whose operations can generally be executed in parallel. However, there may be certain combinations of operations that cannot be executed by the hardware. The *Constraints* section is used to make these combinations visible to the compiler so that the compiler can avoid generating invalid operation combinations.

Constraints are described as a set of Boolean rules, all of which must be satisfied for an instruction to be valid. Constraints may be time-shifted to indicate conflicts in instructions issued at different times. Wild cards may be used to simplify the constraint descriptions. Variables may be used to enforce any restriction requiring different parts of a single constraint to match.

There exist three types of constraints. They are:

- **Resource Conflicts** - Two parallel operations attempt to use the same resources (e.g., competition for the bus).
- **Bitfield Conflicts** - Two parallel operations try to set the same bitfield in the instruction word.
- **Syntactic Constraints** - Constraints that do not correspond to hardware conflicts, but are artifacts of the assembler syntax. For example, it is possible to have an architecture that allows two different operations to assign a value to the same bitfield. However, in such a scenario, the assigned value must be identical in both cases.

All three forms of constraints are included in the constraints section of ISDL.

### 3.3.6 Optional Architectural Details

The ISDL description can provide additional information about the hardware architecture in order to generate better tools. This information is not necessary to generate good code or to produce an instruction-level simulator, but it may result in better code and more accurate simulation (in terms of cycles taken). For example, correct code can be generated without any knowledge of the presence and structure of caches; however, further optimizations can be performed if information about the caches is available.

## 3.4 An ISDL Example

In order to better illustrate the features of ISDL, this section provides an extended example based on the simple architecture of Figure 3-3. It is a VLIW architecture with three functional units `U1`, `U2`, and `U3`. Each functional unit has its own register file consisting of four 8-bit registers. The architecture also includes a data memory of 32 8-bit locations and an instruction memory capable of storing 256 44-bit instructions. The register files and the two memories are connected through two buses: `DB1` and `DB2`. This architecture can perform three data operations and two data transfers in parallel. In terms of data manipulation operations, `U1` can perform addition and subtraction, `U2` can perform addition, subtraction and multiplication, and `U3` can perform addition and multiplication. The complete ISDL description for this architecture is presented in Appendix A.

The instruction word for the example architecture is shown in Figure 3-4. Each functional unit has its own field in the instruction word. Each field consists of an opcode, two source register identifiers, and one destination register identifier. Each of the buses also has its own field in the instruction word consisting of the databus source and destination identifiers.

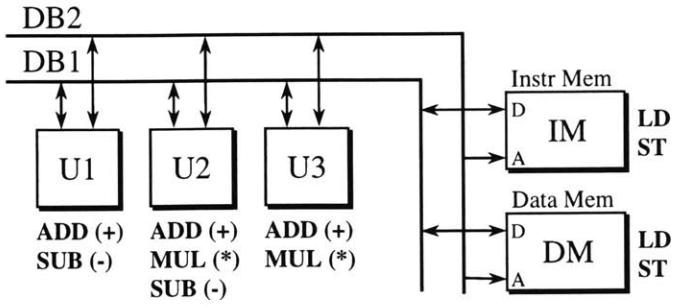



Figure 3-3: **The example VLIW architecture**

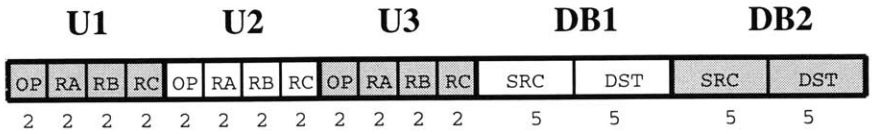



Figure 3-4: **The instruction word of the example VLIW architecture**

### Instruction Word Format

The Format section for this example architecture is shown below. It describes the components of the instruction word.

```
Section Format

U1   = OP[2], RA[2], RB[2], RC[2];
U2   = OP[2], RA[2], RB[2], RC[2];
U3   = OP[2], RA[2], RB[2], RC[2];
DB1  = SRC[5], DEST[5];
DB2  = SRC[5], DEST[5];
```

This description specifies that the instruction word is divided into five fields `U1`, `U2`, `U3`, `DB1`, and `DB2`. Each field is subdivided into subfields and each subfield is annotated with its length in bits. The concatenation of all subfields in MSB (most significant bit) to LSB (least significant bit) order results in the instruction word shown in Figure 3-4.

### Global Definitions

The token definitions for the example architecture are presented below:

```
Section Global_Definitions

//      assembly        token   value
Token   "U1.R"[0..3]    U1_R    { [0..3]; };
Token   "U2.R"[0..3]    U2_R    { [0..3]; };
Token   "U3.R"[0..3]    U3_R    { [0..3]; };
```

The first line that begins with the keyword `Token` defines a token that groups together the syntactic entities `U1.R0`, `U1.R1`, `U1.R2`, and `U1.R3` as denoted by the assembly syntax declaration `"U1.R"[0..3]`. These are the names of the registers in the register file of unit `U1`. The token is named `U1_R` and can be referred to in non-terminal and operation definitions using that name. The return values are zero, one, two, and three respectively (i.e., they are the index of the corresponding register) as denoted by the return value entry `{ [0..3]; }`. The second and third token definitions group the registers of unit `U2` and `U3` respectively.

The following set of non-terminal definitions are also part of the Global Definitions section:

```
Non_Terminal U1_RA: U1_R { $$ = U1_R; } { U1[U1_R] } {} {} {} ;
Non_Terminal U1_RB: U1_R { $$ = U1_R; } { U1[U1_R] } {} {} {} ;
Non_Terminal U1_RC: U1_R { $$ = U1_R; } { U1[U1_R] } {} {} {} ;
...

Non_Terminal SRC:
        U1_R { $$ = 0x00 | U1_R; } { U1[U1_R] } {} {} {} |
        U2_R { $$ = 0x04 | U2_R; } { U2[U2_R] } {} {} {} |
        U3_R { $$ = 0x08 | U3_R; } { U3[U3_R] } {} {} {} ;
```

```
Non_Terminal DEST:
        U1_R { $$ = 0x00 | U1_R; } { U1[U1_R] } {} {} {} |
        U2_R { $$ = 0x04 | U2_R; } { U2[U2_R] } {} {} {} |
        U3_R { $$ = 0x08 | U3_R; } { U3[U3_R] } {} {} {} ;
```

The first line defines a non-terminal named `U1_RA`. This name can be used to refer to it in operation definitions as well as other non-terminal definitions. This non-terminal consists of a single option, or syntactic entity, namely the token `U1_R`. The return value, which can be used in the bitfield assignments of operations, is the same as the return value of the token as denoted by the return value statement `{ $$ = U1_R; }`. The RTL action corresponding to this non-terminal is simply a reference to the appropriate storage location as denoted by the RTL action statement `{ U1[U1_R] }`. It specifies that the non-terminal refers to a register in register file `U1` indexed by the value of the `U1_R` token. The next set of braces normally contains the RTL side-effect of the non-terminal. In the example shown, it is blank because there is no side-effect. The next two sets of braces normally contain the costs and timing modifiers of the non-terminal option. These, too, are empty because in this example all of the values are zero.

The following two lines define non-terminals identical to `U1_RA` except that they have different names. The reason for defining identical non-terminals with different names is to distinguish among them when they are used in the same operation definition.

The complete ISDL description defines similar non-terminals for fields `U2` and `U3`. For brevity, these are omitted in the description above.

The next non-terminal definition is named `SRC` and has three options: `U1_R`, `U2_R`, and `U3_R`. This non-terminal groups the register names of all three register files into a single syntactic entity. The return values of this non-terminal are as follows: for registers in the `U1` register file, the return value is simply the register index; for registers in the `U2` register file, the return value is the register index plus four[2]; and

[2]The value of `U2_R` is between 0 and 3. This value is represented using two bits. Thus, `0x04 | U2_R = 4 + U2_R`.

for registers in the U3 register file, the return value is the index of the register plus eight. Each non-terminal option is assigned a unique value in order to distinguish between the various options. The RTL action for each option is the reference to the appropriate storage location. This non-terminal does not result in any side-effects or costs and timing modifications. An identical non-terminal named DEST is also defined. It is to be used in conjunction with the SRC non-terminal in describing databus move operations.

### Storage Resources

The following is the complete Storage section description for the example architecture.

```
Section Storage

Instruction Memory INST  =  0x100  ,  0x2C
Memory DM                =   0x20  ,   0x8
RegFile U1               =    0x4  ,   0x8
RegFile U2               =    0x4  ,   0x8
RegFile U3               =    0x4  ,   0x8
ProgramCounter PC        =              0x8
```

Each of the storage units (memories and register files) is listed along with the number of entries it contains and the width of each entry. For individual registers, such as the Program Counter, the size describes the width of the register in bits. Note that the instruction memory is identified explicitly, and that the program counter must be included even though it is implied by the instruction set.

### Instruction Set

A portion of the Instruction Set section for the example architecture is shown below.

```
Section Instruction_Set

Field U1f:
  U1_add U1_RA, U1_RB, U1_RC
    { U1.OP = 0x0; U1.RA = U1_RA; U1.RB = U1_RB;
      U1.RC = U1_RC; }
    { U1_RC <- ADD(U1_RA,U1_RB); }
    {}
    { Cycle = 1; Size = 1; Stall = 0; }
    { Latency = 1; Usage = 1; }
  ...
Field U2f:
  ...
Field U3f:
  ...
...
```

The operations of the three functional units, the memories, and the databuses are defined in the instruction set section. The functional unit descriptions consist of three field definitions, one for each functional unit. Each field lists all of the operations that the corresponding functional unit supports. For brevity, a single operation, namely, an add on unit U1 is presented. The syntax of the operation is shown on the first line of the operation definition. It consists of the operation name U1_add, followed by a list of parameters. The operation's parameters are three register names denoted by the non-terminals U1_RA, U1_RB, and U1_RC. The following is an example of an assembly operation of this type:

```
U1_add U1.R0, U1.R2, U1.R2
```

The first set of braces in the operation definition contain the bitfield assignments (i.e., the bits assigned to the various subfields in the instruction word to denote this

operation). In this case, the subfields of the U1 field are assigned the following values: the OP subfield is assigned the value 0 which is the opcode for the add operation, and the RA, RB, and RC subfields are set to the return values of the corresponding non-terminals. These are actually the indices of the corresponding registers in the register file.

The next set of braces contain the action of the operation in RTL. For this operation, the value of the register corresponding to the first parameter is added to the value of the register corresponding to the second parameter. The result is stored in the register corresponding to the third parameter of the operation.[3]

The third set of braces describe the side-effects of the operation. In this example, there are no side-effects specified. An empty side-effects specification in ISDL implies that the only side-effect is that the program counter, PC, is incremented in order to fetch the next instruction. Note that control flow in ISDL is expressed by explicit manipulation of the PC.

The last two sets of braces provide the cost and timing parameters of the operation. This add operation takes one cycle to execute, and requires at most one instruction word. Such an operation will not introduce any stall cycles if a subsequent instruction attempts to access the result of the current operation. A one unit latency specifies that the next instruction can use the result of the current operation. The usage entry specifies that the next instruction can assign an operation to functional unit U1 which is the functional unit of the current operation.

### Constraints

A portion of the Constraints section of the example VLIW architecture description is presented below:

[3]The reference of the non-terminals U1_RA, U1_RB, and U1_RC inside the RTL action of the operation description refers to the RTL value of the corresponding non-terminal.

```
Section Constraints

// SRC and DEST cannot be the same on either bus
~( DB*_move U@[1].R*, U@[1].R* )
```

The third line declares a constraint that is not satisfied if (1) the instruction contains a databus move operation on either bus (represented by `DB*`), and (2) the source (first parameter) and destination (second parameter) of the operation come from the same register file (represented by '`U@[1].R*, U@[1].R*`' where `@[1]` is a variable that must match in both its instances). This type of operation cannot be executed by the hardware because each register file in this architecture only has one port attached to each databus. Therefore, a constraint is required to disallow this type of operation.

A more detailed description of the ISDL constraints is provided in Section 7.1. The next section explains how constraints can be used to simplify the description of the target processor.

## 3.5 Using Constraints to Simplify the Machine Description

Of the previously existing machine description languages, described in Section 2.3, the nML machine description language is the one that is the most similar to ISDL. The nML language uses an attributed grammar to describe the target processor. However, nML does not support explicit constraints. A language that does not support explicit constraints can only describe valid target processor instructions. This can often be significantly more complex than describing a set of generally orthogonal operations, and then eliminating illegal combinations through the use of constraints. As a result, machine descriptions written in ISDL will generally be significantly simpler and easier to use than those written in a language that does not support the description of constraints.

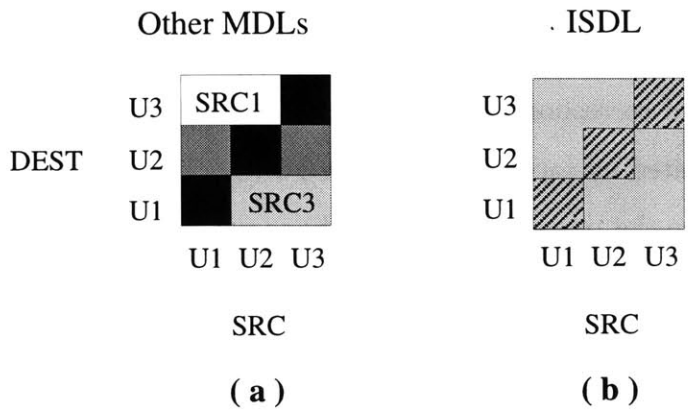



Figure 3-5: **Constraints help simplify the machine description (a)** Instruction space for languages without constraints **(b)** Instruction space using ISDL

Figure 3-5 illustrates how constraints simplify the description of a target processor. In the example architecture shown in Figure 3-3, the databuses cannot be used to transfer data from one register to another within the same register file. Rather, they can only be used to transfer data across register files. Thus, in describing a databus `Move` operation from `SRC` to `DEST`, it is necessary to specify the legal combinations of `SRC` and `DEST`. In a language that is purely an attributed grammar, without support for explicit constraints, all possible legal combinations of `SRC` and `DEST` must be listed as separate `Move` operations. To describe the databus `Move` operation of the example architecture using such a language, the following rules are required:

```
SRC1:   U1 | U2 ;
SRC2:   U1 | U3 ;
SRC3:   U2 | U3 ;

Move1:  Move SRC1, U3 ;
Move2:  Move SRC2, U2 ;
Move3:  Move SRC3, U1 ;
```

The first rule defines a non-terminal, `SRC1`, as all possible sources except for the `U3` register file. Its corresponding operation is `Move1` which specifies that a

'Move SRC1,U3' operation is supported by the target processor. This operation is legal because the SRC1 non-terminal can never represent the U3 register file. Similar non-terminals and corresponding operations must be defined for each possible destination, as illustrated by each row of Figure 3-5 (a). The shaded (non-black) portions of each row define the possible sources of the move operation for the destination corresponding to that row. The black boxes represent the combinations of source and destination that are illegal on the target processor. For this particular example, the description of the valid move operations consists of three non-terminal definitions and three corresponding move operations.

On the other hand, with ISDL, which supports explicit constraints, a single move operation definition, 'Move SRC,DEST', can be used. In this operation definition SRC and DEST are non-terminals corresponding to all possible sources and destinations of the operation.

```
SRC:    U1 | U2 | U3 ;
DEST:   U1 | U2 | U3 ;

Move1:  Move SRC, DEST ;
```

In other words, the move operation includes the entire space of sources and destinations, as shown in the shaded region of Figure 3-5 (b). In addition to the one move operation, a constraint which prohibits the SRC and DEST of the Move operation from coming from the same register file must also be described.

```
~( DB*_move U@[1].R*, U@[1].R* )
```

The constraint corresponds to removing the patterned shaded blocks from the instruction space. The combination of one operation, two general purpose non-terminals, and one constraint describes the same information in ISDL, that the three operation specific non-terminals and three operations described in nML (or any other language consisting of an attributed grammar without support for constraints). As the instruction space expands, or additional constraint dimensions are introduced, the

number of rules required to specify all valid instructions increases rapidly for machine description languages that do not support explicit constraints. In ISDL, on the other hand, adding constraint dimensions simply involves the addition of a constant number of constraints to the description. This demonstrates that a machine description language that supports explicit constraints will result in significantly simpler target processor descriptions.

# Chapter 4

# The Split-Node DAG

Chapter 1 presented an overview of the AVIV code generation framework. This framework is reillustrated in Figure 4-1. This chapter presents the Split-Node DAG definition and describes how it is created from the application code and the ISDL machine description of the target processor. The chapter begins with an overview of the steps involved in creating the Split-Node DAG. A detailed description of each step follows the high-level overview. The following chapters elaborate on the various tasks involved in generating code from the Split-Node DAG.

The Split-Node DAG is a novel machine-specific graphical representation of the application code. It explicitly represents all possible ways of implementing the operations of the input application on the target processor. It also models all of the required data routing operations. This allows the cost of the data routing operations to be considered when selecting target processor instructions to cover the nodes of the Split-Node DAG. The Split-Node DAG allows for the exploration of the parallelism available on the target processor. In addition, it contains all of the information necessary to consider the trade-offs in the instruction selection, resource allocation, and scheduling tasks of code generation.

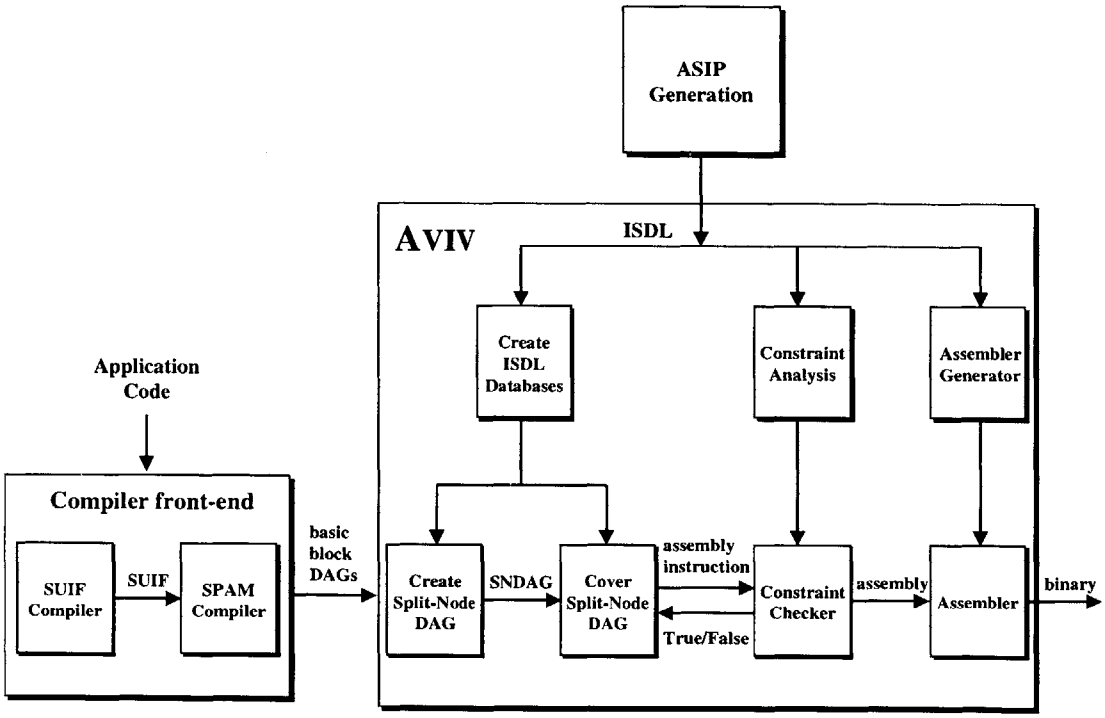



Figure 4-1: **Retargetable code generation framework using AVIV**

## 4.1 Roadmap for Transforming High-Level Code into a Split-Node DAG

In order to create the Split-Node DAG, the application code is first transformed into an intermediate form using the SUIF compiler. The SPAM compiler then converts the SUIF intermediate form into a set of machine-independent basic block DAGs (Directed Acyclic Graphs) that represent the application code. A DAG is a graphical representation of the intermediate form. Each node of the DAG represents a SUIF operation, and each edge represents a precedence relation. To convert a basic block DAG into a Split-Node DAG, each operation node of the DAG is split into several nodes representing the various functional units that can execute that operation on the target processor. The functional unit information can be used to determine whether or not a data transfer is required between a pair of operation nodes. A data transfer is required if the node that produces a value and the node that consumes the value do not use the same storage locations. In addition to representing the multiple ways of executing each operation on the target processor, the Split-Node DAG also includes all required data transfers. Including the data transfers in the Split-Node DAG ensures that when covering the nodes of the Split-Node DAG with target processor instructions, the optimization functions can account for the extra operations required to cover the data transfer nodes.

The ISDL description of the target processor is used to extract all of the machine-specific information necessary for creating the Split-Node DAG. In order to simplify this process, the ISDL description is parsed and analyzed, and the results of the analysis are stored in various databases that are used in subsequent steps of the code generation process. Two databases are required for the creation of the Split-Node DAG. The first provides a correlation between the SUIF machine-independent operations and the ISDL machine-specific operations. The second database stores all the register-to-register data transfer operations available on the target processor (i.e., it does not include load or store operations).

## 4.2 The Compiler Front-End

As illustrated in Figure 4-1 the code generation process in AVIV begins with the SUIF compiler front-end. The SUIF compiler receives the application code, written in a high-level language, as input. It transforms the application code into an unoptimized intermediate representation named SUIF. Machine-independent optimizations are then performed on the unoptimized SUIF representation. The resulting optimized SUIF representation is then passed to the SPAM compiler in order to convert the intermediate form into basic block DAGs.

### 4.2.1 Overview of the SUIF Compiler

The SUIF compiler is used as the front-end of the SPAM compiler for two reasons [51]. First, the object-oriented design style of SUIF provides well defined interfaces to all of the front-end data structures. These interfaces facilitate the implementation of new machine-independent optimizations. Second, the SUIF intermediate representation supports both high-level and low-level data structures to represent the application code. The high-level data structures include `for` loops, `if` statements, and array accesses. High-level data structures are useful for compiler passes that perform dependence analysis and loop transformations. Low-level data structures (i.e., sequential lists of instructions) are better suited for scalar optimizations and code generation. Supporting both forms of data structures can facilitate subsequent compilation tasks by making the information available in the most desirable form.

### 4.2.2 Machine-Independent Optimizations

In order to improve the quality of the code, machine-independent optimizations are performed on the intermediate representation of the application. There are several forms of machine independent optimizations. They include structure-preserving transformations, algebraic transformations, and transformations that can increase the machine independent parallelism. Examples of these optimizations are described

below[1].

### Structure-Preserving Transformations

Examples of structure-preserving transformations are dead-code elimination and common subexpression elimination.

Dead-code elimination is the process of removing code that produces a result that is never subsequently accessed. Dead-code may be removed because it does not affect the functionality of the program. For example, in the following lines of code,

```
1:  x = y + 1;
2:  x = y * z;
3:  z = x - 3;
```

the first assignment to variable `x` on line 1 is never subsequently used; therefore, line 1 of the code is dead-code and can be removed from the program.

Common subexpression elimination is the process of finding multiple expressions that produce the same result and merging them into a single expression whose result is used multiple times. For example, in the following lines of code,

```
1:  a = b + c;
2:  b = a - d;
3:  c = b + c;
4:  d = a - d;
```

the second and fourth statements produce the same value, namely `b + c - d`. The reevaluation of the expression `a - d` in the fourth statement is redundant. Instead, the expression should be evaluated once, and the result should be reused as shown below:

[1]The examples used to describe the structure-preserving and algebraic transformations were found in [3].

```
1:  a = b + c;
2:  b = a - d;
3:  c = b + c;
4:  d = b;
```

Note that although the first and third statements appear to have the same expression on their right hand side, they are not equivalent because the second statement redefines variable b.

### Algebraic Transformations

Algebraic transformations can either simplify expressions or replace *expensive* operations with *cheaper* operations that have the same functionality. For example the statements,

```
x = x + 0;
```
*and*
```
x = x * 1;
```

can be eliminated because they do not affect the value of variable **x**. Furthermore, the statement

```
y = y * 2;
```

which includes a multiplication operation can be replaced by the statement

```
y = y + y;
```

which includes an addition operation. The two statements are functionally equivalent; however, an addition operation is often *cheaper* to execute than a multiplication operation.

### Other Transformations

The transformations described above simplify the input code without altering its structure. There are additional transformations that can produce more efficient code but do alter the program's structure. These transformations include loop optimizations such as loop-invariant code motion. Loop-invariant code motion moves code that is independent of the loop index outside of the loop. Such code only needs to be executed once rather than on each iteration of the loop.

Loop unrolling is another example of a loop optimization. Loop unrolling increases the amount of parallelism available within a segment of code. This is achieved by exposing independent operations that occur in different iterations of the loop and allowing them to be executed in parallel. For example, in the following code segment that initializes the 10 element array `A`, each iteration of the loop is independent of all other iterations.

```
for ( i = 0; i < 10; i++ ) {
      A[i] = i;
}
```

Unrolling the loop once produces the following code segment:

```
for ( i = 0; i < 10; i = i + 2; ) {
      A[i] = i;
      A[i+1] = i+1;
}
```

In the unrolled version of the loop, two-way parallelism exists because two independent assignments are being made within the loop. An architecture that is capable of executing both operations in parallel can increase its performance significantly by unrolling the loop and executing the assignments in half as many iterations.

All of the transformations described above are machine independent. In other words, no knowledge about the target processor is required in order to perform these optimizations. However, the effect of each optimization can depend on the target

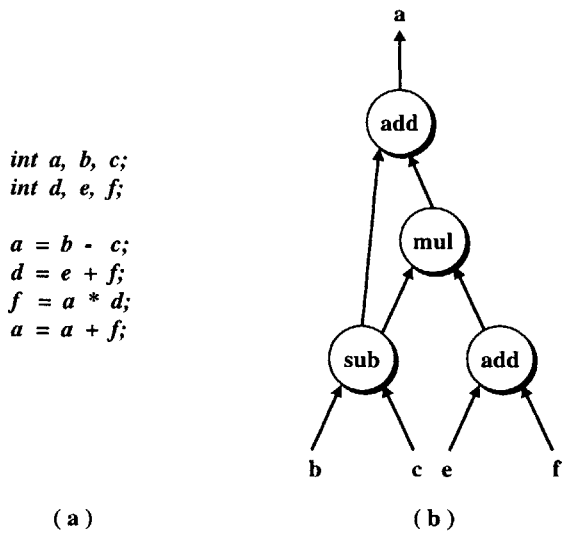



Figure 4-2: **Converting a basic block of code into a basic block DAG** (a) Input code (b) Resulting basic block DAG

architecture. For example, if a target architecture only supports two-way parallelism, then unrolling a loop eight times may not produce better results than unrolling the loop twice since the increased parallelism cannot be utilized by that target processor.

### 4.2.3 Creating the Basic Block DAGs

After the machine-independent optimizations are performed, the optimized SUIF intermediate form is converted into basic block DAGs. A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without the possibility of branching except at the end. A *basic block DAG* is a directed acyclic graph representation of the basic block.

Figure 4-2 illustrates the process of converting a basic block of code (Figure 4-2 (a)) into a basic block DAG (Figure 4-2 (b)). Each node in the basic block DAG represents a SUIF operation, and each edge represents a precedence relation. This example illustrates that nodes of a DAG can have more than one parent (i.e., the `sub` node has two parent nodes because it is a common subexpression of the `mul` and `add` operations). Note that transforming the input code into a basic block DAG is also a

machine-independent transformation; thus, it does not require any information from the ISDL description of the target processor.

## 4.3 Matching SUIF Operations to ISDL Operations

Each node in the basic block DAG corresponds to one SUIF operation. These SUIF operations must be translated into operations available on the target processor in order to allow for instruction selection. This translation involves mapping each SUIF operation, as well as each operation in the ISDL description, into an expression tree representation of the operations. An expression tree matcher is then used to determine which SUIF and ISDL operations are equivalent.

The operation expression trees represent the functionality of the operation. The functionality of each SUIF operation is set and thus does not need to be interpreted. However, the functionality of each ISDL operation is not known in advance. Therefore, the first step in creating the operation expression trees is to analyze the RTL portion of each ISDL operation.

An RTL statement in ISDL represents the functionality of the operation. The RTL statements can represent various types of expressions including assignment and control flow expressions. An assignment expression assigns the value of a specified ISDL operation to a specified storage entity. Control flow expressions include if statements, for loops, while loops, and switch statements.

The expression trees created to match the ISDL and SUIF operations follow the same format as the ISDL RTL statements. Thus, an *if statement* is represented as an *if statement* expression tree. The following examples should clarify the structure of the operation expression trees.

A simple example of an operation expression tree is shown in Figure 4-3. In this example the SUIF operation is an `ADD` operation on two source operands, and the ISDL operation is also an `ADD` on two source operands. Such matches are called

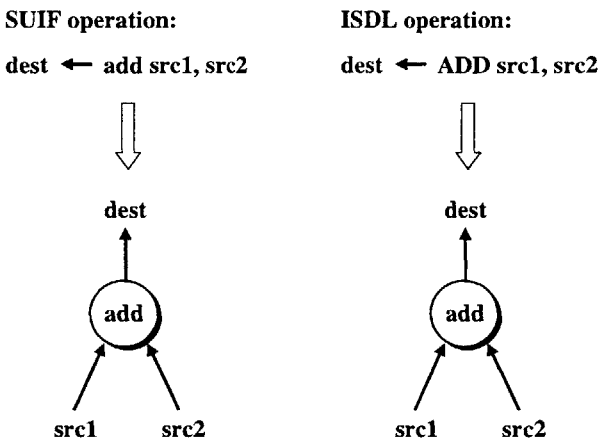



Figure 4-3: **Expression trees for add operation**

*one-to-one* matches because one ISDL operation is equivalent to one SUIF operation.

More complex expression trees can also result in one-to-one matches. For example, a *branch on false* operation results in the complex expression tree illustrated in Figure 4-4. Although the SUIF and ISDL representations of this operation are quite different, the resulting expression tree mappings are identical. This means that the two operations perform the same function.

One-to-one matches also include matches that require a reversal of the operands in order to find a match. For example, a *signed less than* operation between operands `A` and `B` is equivalent to a *signed greater than* operation between operands `B` and `A` (i.e., $A < B \equiv B > A$). These two operations would result in a one-to-one match.

In addition to finding one-to-one matches, many-to-one and one-to-many matches must be found as well. Many-to-one and one-to-many matches are defined below:

- *Many-to-One match* - Several SUIF operations can be merged into a single ISDL operation. This occurs frequently in architectures containing complex functions such as multiply-accumulate. A *multiply-accumulate* complex function can replace a *multiply* operation followed by an *add* operation in SUIF.

- *One-to-Many match* - In cases where no ISDL operation match was found for

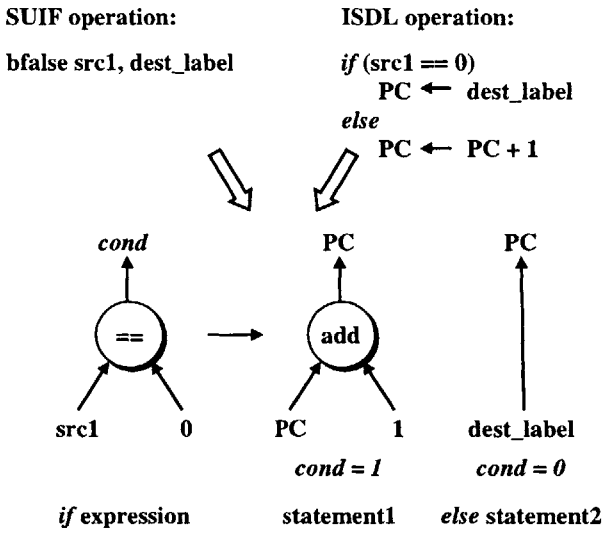



Figure 4-4: **Expression tree for branch on false operation**

a particular SUIF operation, combinations of multiple ISDL operations should be searched for equivalence to the single SUIF operation. For example, if the basic block DAG contains a *signed equal* comparison operation, but the target processor only contains a *signed not equal* comparison operation, then a *not* operation can be merged with the *signed not equal* operation to form a one-to-many match of the SUIF operation (*signed equal* is equivalent to a *signed not equal* followed by a *not* operation).

When parsing the ISDL description, AVIV searches for one-to-one, many-to-one, and one-to-many matches. The parsing process occurs once, and the resulting matches are stored in the *SUIF-ISDL Correlation Database.* This database stores correlations between the SUIF basic operations and the target processor operations.

## 4.4 Additional Databases Created by AVIV

The *Register-to-Register Data Transfer Database* stores all of the register-to-register data transfer operations available on the target processor. These data transfer operations are defined as data move operations from one register file (or register) to

another. In order to find all of the data transfer operations available on the target processor, the RTL statement of each operation in the ISDL description is checked to determine whether or not it corresponds to an *assignment* statement whose source and destination operands are registers or register files. If so, the operation is considered a data transfer and is added to the database. If the source or destination operands are non-terminals representing multiple sources or destinations, then the operation is expanded into all possible data transfers. Each possible data transfer is added to the database individually. This is necessary because when a data transfer operation is inserted into the Split-Node DAG, it must represent exactly one possible move operation. A data transfer operation is required if the operation that produces a value stores its result in a storage location that the operation that consumes the value cannot access. In such a situation, a data transfer operation is inserted between the producing and consuming operations. Note that *loads* and *stores* are also considered to be data transfers but are not included in this database. The reason is that register-to-register data transfers will not match any SUIF operations. Therefore, they need to be treated differently from all other operations.

Additional databases are created to store all of the ISDL description information in a manner that is more easily accessible by the code generator. The ISDL description is parsed once, and the parsed data is stored in the databases. Subsequently, when the code generator requires information about the target processor, it looks up the required information in the databases. Below is a list of all databases used by AVIV:

- **Token and Non-terminal Database** - Stores all of the tokens and non-terminals and their associated information.
- **Storage Database** - Stores a list of all of the storage capabilities of the target processor including the type and size of each storage element.
- **Fields Database** - Stores a list of all fields present in the ISDL description along with indices of `NULL` and `NOP` operations for that field, if they exist.
- **Operation Database** - Stores a complete list of operations available on the target processor and their associated data.

- **SUIF-ISDL Correlation Database** - Stores a list of correlations between the SUIF basic operations and the ISDL target processor operations.

- **Register-to-Register Data Transfer Database** - Stores all of the register-to-register data transfer operations available on the target processor.

These databases are used repeatedly in the generation of the Split-Node DAG and in the routines that cover the Split-Node DAG nodes with target processor instructions.

## 4.5 Converting the Basic Block DAGs into Split-Node DAGs

Syntactically, a Split-Node DAG is similar to a DAG representing the operations performed in a block of code. A Split-Node DAG has two additional types of nodes: *split nodes* and *data transfer nodes*. A *split node* corresponds to an operation node in the original DAG. The immediate (non data transfer node) descendants of a split node correspond to all possible ways the operation may be performed on the target processor. The *data transfer nodes* are inserted on the path between a split node and its immediate operation descendants and correspond to data transfers required along that path.

Given a basic block DAG representation of the application code and the databases generated by AVIV, the Split-Node DAG can be created. In particular, the SUIF-ISDL Correlation database and the Register-to-Register Data Transfer database are required for this task.

In order to clarify the derivation of the Split-Node DAG, an illustrative example is provided. For this example, the basic block DAG shown in Figure 4-5 is converted into a Split-Node DAG. The remainder of this chapter describes this conversion process in detail.

In the basic block DAG of Figure 4-5, the *leaves* of the DAG, represented as squares, correspond to the *primary inputs* of the DAG. The *roots* of the DAG, repre-

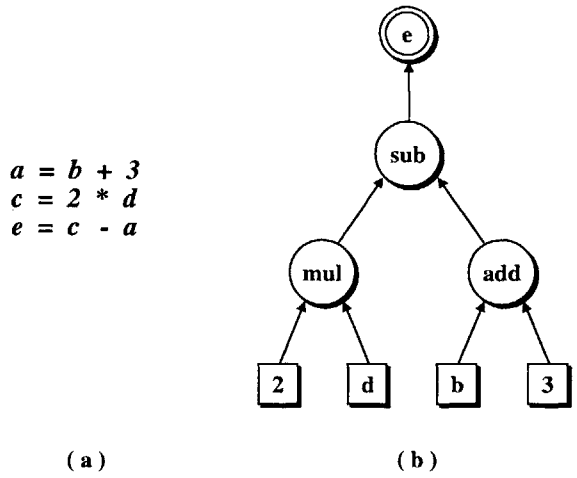



Figure 4-5: **Example basic block DAG (a)** C code for basic block **(b)** The corresponding basic block DAG assuming variables $a$, and $c$ are not live on exit from the basic block.

sented by double circles, correspond to the *primary outputs.* AVIV assumes that primary inputs and outputs reside in memory. The target processor used in this example is the VLIW processor presented in Figure 4-6. This processor is nearly identical to the one presented in Figure 3-3 except that, for the sake of simplicity, it only contains one databus (`DB1`). In the processor shown here, unit `U1` can perform addition (`ADD`), and subtraction (`SUB`). Unit `U2` can perform addition (`ADD`), subtraction (`SUB`), and multiplication (`MUL`). Unit `U3` can perform addition (`ADD`), and multiplication (`MUL`). Each unit contains its own register file and can perform only one operation at a time. The architecture also includes an instruction memory (`IM`), and a data memory (`DM`). Load and store operations can be performed on the two memories. A single databus connects all units and memories. The databus allows data to be transferred between the different functional units. In addition, the databus can be used to perform a *load constant* (move immediate) operation.

The first step in converting the basic block DAG into a Split-Node DAG is to insert *split-nodes* between each pair of operation nodes. The second step is to replace each operation node with $N$ operation nodes, where $N$ is the number of target processor operations that match the DAG operation node. Figure 4-7 illustrates these first two

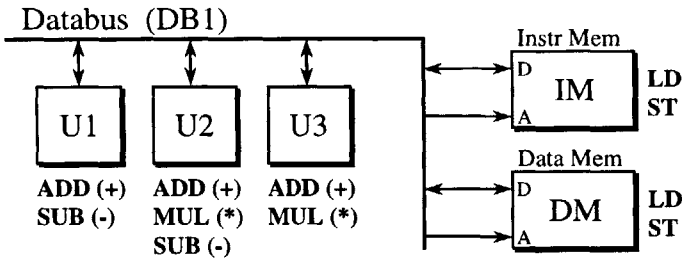



Figure 4-6: **Example target architecture**

steps. The *SUIF-ISDL Correlation Database* is used to perform the second step. For this example architecture, the `SUB` node is replaced by two `SUB` nodes, one on unit `U1` and the other on unit `U2`. The `MUL` node is replaced by two `MUL` nodes, one on unit `U2` and one on unit `U3`. Finally, the `ADD` operation node is replaced by three `ADD` nodes on units `U1`, `U2`, and `U3`.

The final step in creating the Split-Node DAG is to insert all of the required data transfer nodes. This includes data transfers from one register file to another, as well as load, load constant, and store nodes. In the example architecture, each functional unit has its own dedicated register file. Thus, if an operation is executed on one functional unit (e.g., unit `U1`) but its operand was produced on another functional unit (e.g., unit `U2`), then a data transfer is required to move the operand from its register file (e.g., unit `U2`'s register file) to the operation's register file (e.g., unit `U1`'s register file). If an operation and its child are both executed on the same functional unit, then no data transfer is required. In order to determine which target processor operation can perform each required data transfer, the *Register-to-Register Data Transfer Database* is queried. This query returns all data transfer operations that move a value from the source register file (e.g., unit `U2`'s register file) to the destination register file (e.g., unit `U1`'s register file). All data transfer operations returned from the query are included in the Split-Node DAG. This implies that more than one path can connect two operation nodes if multiple data transfer paths exist.

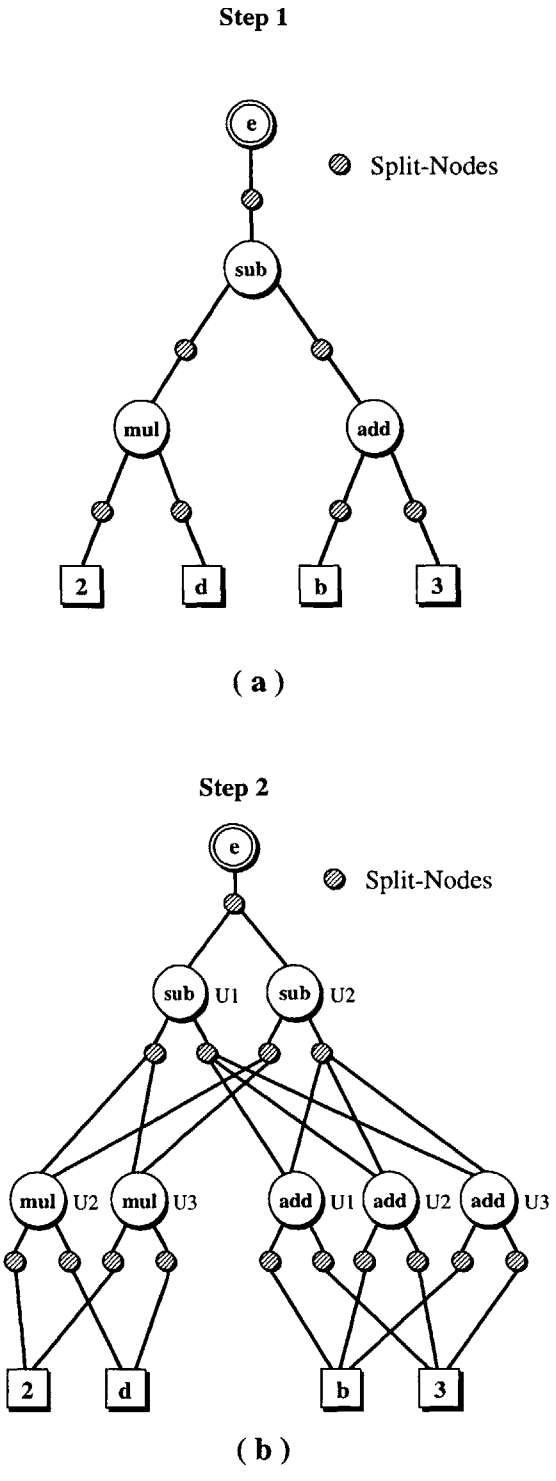



Figure 4-7: **Splitting the basic block DAG nodes into multiple nodes** **(a)** Inserting split nodes into the DAG **(b)** Splitting the DAG operation nodes into multiple target processor operation nodes.

Finally, any edge whose source is a constant value (e.g., 2), has an ISDL operation equivalent to a *load constant* operation inserted between the constant node and the operation node. These nodes are considered as data transfer nodes. In the example architecture, the databus is capable of performing a load constant (move immediate) operation. In addition, any edge that joins an input variable and an operation node has a *load* data transfer node inserted along it, and any edge that joins an operation node and an output variable has a *store* data transfer node inserted along it. This is required because AVIV assumes that all inputs and outputs reside in memory.

Figure 4-8 illustrates the resulting Split-Node DAG. Each *split-node* has children corresponding to the multiple units that can perform the `ADD`, `MUL`, or `SUB` operations on the target architecture. Note that paths from several split-nodes can reconverge to one set of operation nodes (i.e., the operation nodes are not duplicated for each split-node). Any edge that connects two operations, A and B, in the original basic block DAG is now split into multiple edges representing all possible paths from operation A to operation B. If these edges result in a transfer from one unit to another, then a data transfer node is added along that edge. This includes multi-level paths if a direct transfer path is not available on the target architecture. If multiple data transfer paths exist, then all of them are included in the Split-Node DAG.

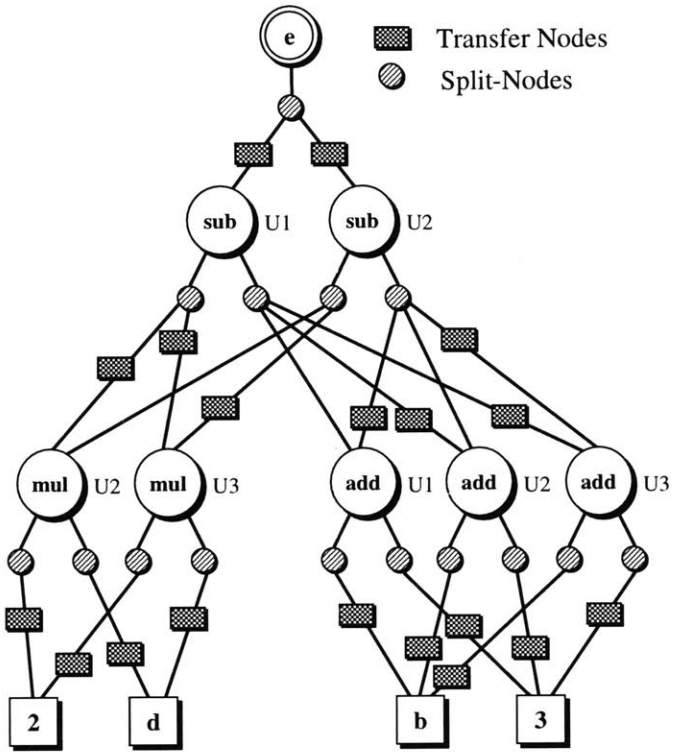



Figure 4-8: **The Split-Node DAG**

# Chapter 5

# Covering the Split-Node DAG

This chapter begins with an introduction to the various tasks involved in code generation. It then describes the methodology used by AVIV to perform code generation from a Split-Node DAG. AVIV's goal is to cover the Split-Node DAG with a minimal-cost set of target processor instructions. In order to achieve results that are close to optimal, AVIV addresses the various tasks of code generation concurrently because these tasks are tightly coupled.

## 5.1 Code Generation Tasks

Code generation consists of three main tasks:

1. Instruction Selection,
2. Resource Allocation, and
3. Scheduling.

Instruction selection is the process of mapping the operations available on the target processor onto the input DAG and selecting the best set of target processor operations to *cover* all nodes of the DAG. A target processor operation may cover more than one node in the input DAG; however, each node in the DAG can only be covered once (i.e., there can be no overlap between the selected operations).

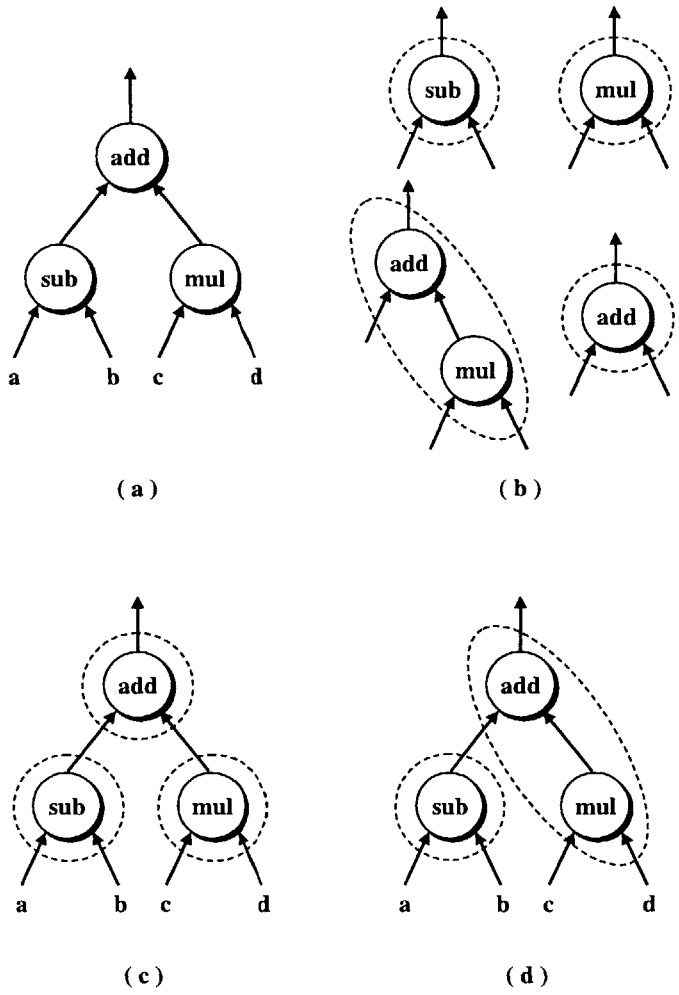



Figure 5-1: **Instruction selection (a)** The input DAG **(b)** The target processor operations **(c-d)** Two possible covers of the input DAG.

Figure 5-1 illustrates the process of instruction selection. The input DAG is shown in Figure 5-1 (a). The available target processor operations are shown in Figure 5-1 (b). Two possible ways to cover the input DAG are illustrated in Figure 5-1 (c) and (d). In (c), each operation is covered individually with a total cost of three target processor instructions. In (d), a complex *multiply-accumulate* operation is used to simultaneously cover two nodes of the input DAG incurring a total cost of two instructions.

In addition to selecting between basic and complex operations, instruction selection may also involve selecting which functional unit should implement a particular operation when more than one functional unit is capable of executing that operation. Thus, each node in the input DAG may have multiple potential covers. The instruction selection task must select the best combination of operation covers that will cover the entire DAG. In other words, *covering* the nodes of the DAG implies selecting a non-overlapping set of target processor instructions that will implement the functionality of the DAG.

Section 4.3 presented the first half of the task of instruction selection which is the matching of target processor operations to operations in the input DAG in order to generate the Split-Node DAG. The remaining portion of instruction selection is to select the best set of target processor instructions that cover the entire input DAG.

The second task of code generation is resource allocation. In the context of VLIW architectures, resource allocation assigns a functional unit to execute each operation and performs register allocation to determine which register will store each intermediate value. Register allocation is the process of assigning operands and results to the available registers on the target processor. A major goal of register allocation is to avoid spills to memory. Spill operations usually require additional instructions and consequently result in reduced performance. For unifunctional architectures, resource allocation consists solely of register allocation.

The final task of code generation is scheduling. Scheduling is the task of taking all of the selected target processor instructions and determining a complete ordering for them. The scheduling phase must ensure that all precedence constraints are satisfied. Additionally, it attempts to minimize the total cycle cost of the resulting schedule. For example, if instruction B depends on the result of instruction A, but instruction A requires two cycles to execute, then the scheduler may choose to insert another independent instruction, C, between instructions A and B. The insertion of instruction C between instructions A and B can only be performed if the new ordering of instructions still satisfies all of the precedence constraints. Reordering the instructions could lead to a more efficient total schedule as illustrated in Figure 5-

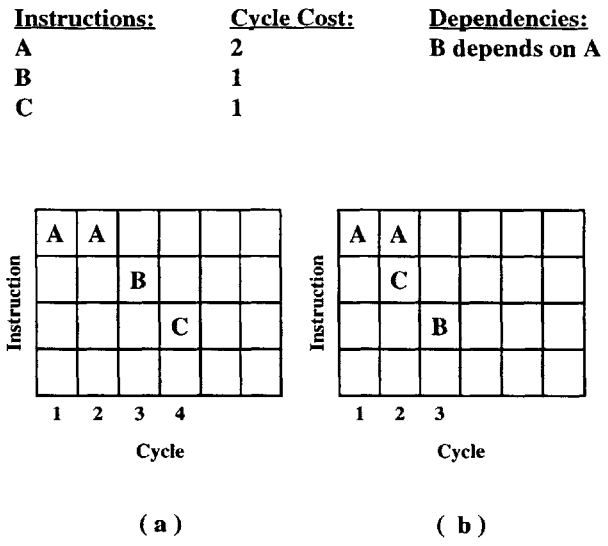



Figure 5-2: **Optimizing the schedule by reordering the instructions**

2. In Figure 5-2 (a), the order of the instructions is A, B, C which incurs a total cost of four cycles. Reordering the instructions to A, C, B allows the extra cycle of instruction A to be overlapped with the execution of instruction C. Figure 5-2 (b) illustrates that this results in a total cycle cost of three.

The instruction selection, resource allocation, and scheduling tasks described above are very complex problems for which no polynomial-time algorithm exists. In fact, it has been shown that optimal code generation for DAGs on a one-register machine is NP-complete[1] [8]. This result was extended to show that even with an unlimited number of registers, the problem remains NP-complete [2]. The difficulty is in determining the optimal order in which the DAG should be evaluated in order to arrive at a minimum cost solution.

In order to alleviate the complexity of generating code for DAGs, the traditional approach has been to convert the DAGs into multiple trees by splitting the DAG at nodes that have multiple fanouts (i.e., nodes that have more than one parent node), as

[1]An NP-complete problem is generally thought of as an intractable problem because no polynomial-time algorithm is known for it. Furthermore, unless P = NP, no polynomial time algorithm exists for any NP-complete problem.

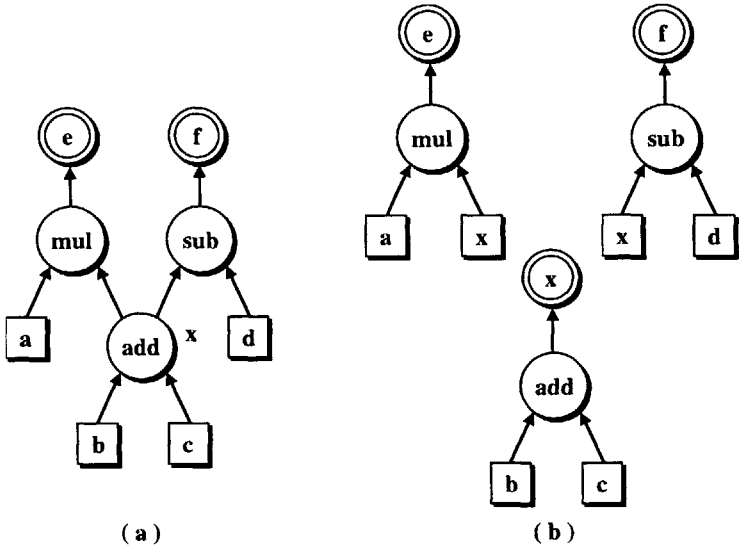



Figure 5-3: **Splitting a DAG into multiple trees (a)** Original DAG **(b)** Multiple tree representation of DAG

illustrated in Figure 5-3. Each point in the DAG that is split has primary inputs and outputs inserted in order to maintain connectivity between the multiple trees. The code generation task is then performed on the individual trees using optimal linear-time algorithms for covering the trees. Examples of linear-time algorithms that find the optimal order of evaluation for trees can be found in [47, 1].

Although optimal linear-time algorithms exist for covering trees, they do not necessarily imply an optimal covering of the DAG. Splitting DAGs into trees precludes the use of complex instructions in cases where internal vertices are shared. Therefore, covering trees rather than DAGs may lead to suboptimal results. Furthermore, performing resource allocation and scheduling on each tree independently could adversely affect the final results of covering the entire DAG. This is because an algorithm that covers each tree independently may not be able to determine the best use of the parallelism available on the target processor. AVIV addresses these issues by setting up a heuristic algorithm to cover the input DAGs directly rather than splitting the DAGs into trees and covering each tree independently.

An additional complication of code generation is that the various tasks of code generation are interdependent. Optimizing one task without taking the others into account can lead to suboptimal results. This problem is typically referred to as the *phase coupling* problem [20]. For instance, suppose that during the instruction selection phase, every time a multiply-accumulate pattern is encountered, the instruction selection routine selects the complex operation that can cover both the multiply and add nodes simultaneously. However, the complex multiply-accumulate operation may only be executable on one functional unit with a limited number of registers. In such an architecture, it may be inadvisable to always select the multiply-accumulate complex function rather than the individual multiply and add functions because register resource constraints may end up causing spills to memory. The spills may incur a higher penalty than using two separate operations to cover the multiply-accumulate sequence of operations. If the instruction selection phase accounted for the register resources, however, then this situation could potentially be avoided.

## 5.2 Code Generation Using the Split-Node DAG

The goal of the AVIV code generator is to cover the Split-Node DAG with a minimal-cost set of target processor instructions. AVIV addresses the instruction selection, resource allocation, and scheduling phases of code generation concurrently because, as mentioned above, these phases are mutually dependent. Therefore, performing them sequentially generally yields suboptimal code. In covering the Split-Node DAG, the following tasks are performed simultaneously:

- functional unit assignment,
- merging operations and data transfers into instructions,
- register bank allocation, and
- scheduling.

Detailed register allocation and peephole optimizations are performed as a second step.

Explore possible split-node functional unit assignments (Section 5.2.1):
- Estimate cost of assignment.
- Select several lowest cost assignments to explore in further detail.

For each selected assignment:
- Insert required data transfers (Section 5.2.2).
- Generate all maximal groupings of nodes that could be executed in parallel (Section 5.2.3).
- Select a minimal-cost set of maximal groupings that covers all nodes (Section 5.2.4).

The final solution is the lowest-cost solution found above.

Figure 5-4: **Algorithm for covering the Split-Node DAG**

Figure 5-4 presents the algorithm used to cover the Split-Node DAG with a minimal-cost set of target processor instructions. Multiple heuristics are used to reduce the runtime of the algorithm without sacrificing the quality of the results. The algorithm begins with a high-level analysis that explores the possible split-node functional unit assignments. It then selects several of the lowest cost assignments to explore in further detail. This selection is based on a heuristic cost function that is described in Section 5.2.1. Each of the selected functional unit assignments is then explored in detail.

The detailed analysis begins by adding the required data transfers to the functional unit assignment (Section 5.2.2). Next, the nodes in the current assignment, including the transfer nodes, are merged into *maximal groupings*. All nodes within a maximal grouping can be executed in parallel and correspond to a single VLIW instruction (Section 5.2.3). A heuristic covering algorithm then covers the nodes in the current assignment with a minimal-cost set of maximal groupings (Section 5.2.4). After the selected functional unit assignments have been explored, the assignment yielding the lowest cost solution is selected, and code is generated using that assignment.

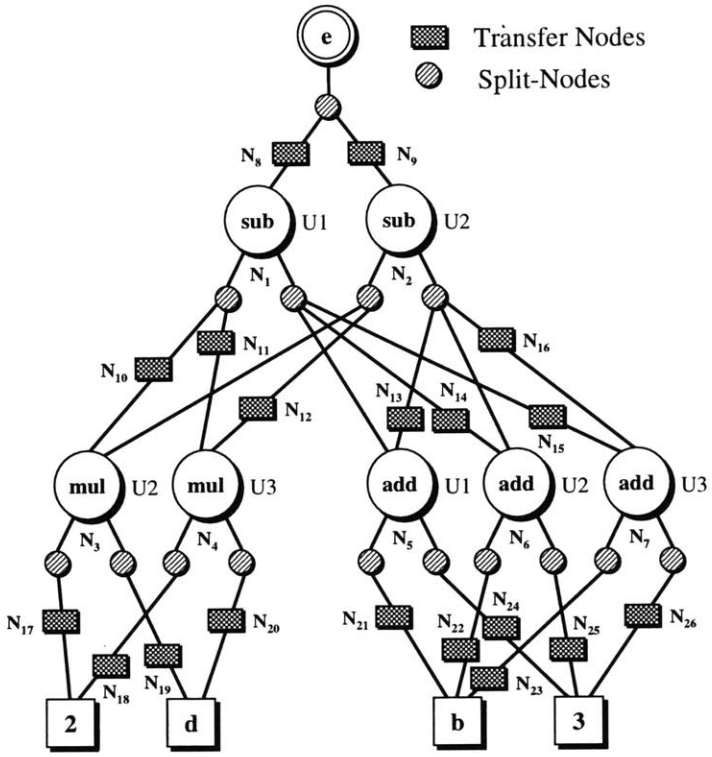



Figure 5-5: **The Split-Node DAG**

### 5.2.1 Exploring the Split-Node Functional Unit Assignments

Instruction selection on the Split-Node DAG is the task of selecting a target processor operation (i.e., functional unit) to cover each split-node. The Split-Node DAG of Figure 4-8 is reillustrated in Figure 5-5 with a label attached to each node. For this Split-Node DAG, node $N_1$ or $N_2$ could be selected to cover the subtract split-node, node $N_3$ or $N_4$ could be selected to cover the multiply split-node, and so on. The number of possible split-node covering assignments can be calculated by multiplying the number of possible target processor operations covering each split-node (i.e., for this example $2 \times 2 \times 3$). This is a very small basic block that results in few possible assignments. However, for typical basic blocks, the multiplicative growth in the number of possible split-node functional unit assignments quickly makes it prohibitively CPU-intensive to explore all possible cases.

Thus, the first step of AVIV's code generation algorithm prunes the search space

by selecting only a few of the split-node functional unit assignments to explore in depth. This selection is made based on a cost function that accounts for the two main factors contributing to the higher cost (measured in number of instructions) of covering the Split-Node DAG. These two factors are the amount of parallelism that is foregone due to the split-node covering (i.e., functional unit) assignments, and the number of data transfers required for the given assignment. To calculate the total cost, the total foregone parallelism (FP), which is the sum of the foregone parallelism between pairs of operation nodes ($FP_{ij}$), and the total number of transfer nodes (TN) must first be calculated. The following equations are used for these calculations:

$$FP_{ij} = \begin{cases} 1 & \text{if } node_i \text{ and } node_j \text{ share a common resource, and} \\ & \text{they could have otherwise been executed in parallel.} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

$$\text{Total FP} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} FP_{ij} \text{ (n = number of split-nodes)} \tag{5.2}$$

$$\text{Total TN} = \text{total number of transfers required for assignment} \tag{5.3}$$

The total cost is a weighted sum of the foregone parallelism and the number of required data transfers, as shown below:

$$\text{Total Cost} = (w_1 \times \text{Total FP}) + (w_2 \times \text{Total TN}) \tag{5.4}$$

It is important to include both factors in the cost function because they allow for the optimization of the instruction selection and scheduling phases of code generation concurrently. The relative magnitude of the two weights, $w_1$ and $w_2$, can be modified to indicate the relative importance of required transfers to foregone parallelism in the target architecture. A good heuristic is to make the two weights equal.

Calculating this cost function for all possible functional unit assignments can in itself be prohibitively expensive in terms of runtime. Thus, the search space of

For each split-node traversed in order of increasing level from the top of the Split-Node DAG:

- Calculate the incremental cost, along the current path, of each operation node corresponding to the current split-node using equation 5.7.
- Prune all paths that do not result in minimum incremental cost.

Each remaining unpruned path represents a split-node functional unit assignment that should be explored in detail.

Figure 5-6: **Algorithm for selecting split-node functional unit assignments**

possible assignments is actually pruned by traversing the Split-Node DAG in a top-down fashion, calculating an incremental cost for each split-node encountered, and pruning the search space at all points that do not result in minimum incremental cost. A high-level description of the algorithm used to select a set of split-node functional unit assignments is presented in Figure 5-6. The incremental cost used in this algorithm is calculated using the following functions:

$$FP_i = \sum_{j=1}^{i-1} FP_{ij} \tag{5.5}$$

$$TN_i = \begin{array}{l}\text{additional number of transfer nodes required as a result} \\ \text{of adding } node_i \text{ to the previously encountered nodes}\end{array} \tag{5.6}$$

$$Incremental\ Cost_i = (w_1 \times FP_i) + (w_2 \times TN_i) \tag{5.7}$$

$FP_i$ represents the amount of foregone parallelism between $node_i$ and the previously encountered nodes in the incremental search space. Thus, the $incremental\ cost_i$ only accounts for costs resulting from adding $node_i$ to the previously encountered nodes. For the purpose of illustrating this cost function, assume that equal weights of value 1 are assigned to $w_1$ and $w_2$. Figure 5-7 shows the incremental cost at each node as well as the locations where the search space was pruned (indicated by an X) for the Split-Node DAG of Figure 5-5.

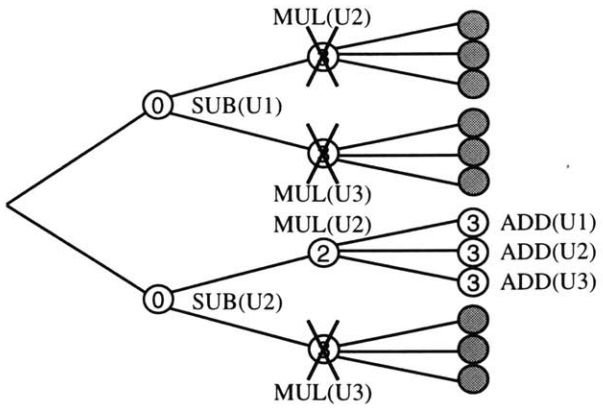



Figure 5-7: **Pruning the search space of split-node assignments**

The incremental cost of the two SUB nodes is zero because they are the first nodes encountered when traversing the Split-Node DAG top-down. Since the two costs are equal, the search is continued along both branches. The next set of nodes tested could either be the MUL or the ADD nodes because their *level from the top* is the same. The algorithm arbitrarily picks the MUL nodes first. Following the SUB node on the unit U1 branch, both the MUL on unit U2 and the MUL on unit U3 have an incremental cost of three. One to transfer the result of the MUL executed on unit U2 or U3 to the SUB operation executed on unit U1. Another two to load the two operands of the operations.

Along the other branch of the SUB operation, that is the SUB node on unit U2, the cost incurred by the MUL operations differ. If the MUL is to be executed on unit U2, then the incremental cost is just two to load the two operands. No additional cost is incurred because no data transfer is required. A data transfer is not required because both the MUL and the SUB operations along that path are executed on unit U2. In addition, no possible parallel execution is prevented because there is a data dependence between the MUL and SUB operations. On the other hand, the MUL on unit U3 has an incremental cost of three because in addition to loading its two operands,

it requires a data transfer to the SUB operation.

The search space is pruned at all the MUL nodes with a cost of three, and then the search continues along paths of minimum incremental cost. In this example, the search is continued along the SUB on unit U2, and MUL on unit U2 path. Along this path, the three possible ADD nodes are encountered. The ADD on unit U1 has an incremental cost of three (two to load its operands, and an additional one to transfer its result to the SUB operation). The ADD on unit U3 has an incremental cost of three, as well, for the same reasons. The ADD on unit U2, however, also introduces an incremental cost of three except in this case its third unit of cost is not due to a data transfer. Instead, its cost is incremented because if both the MUL and ADD operations are to be executed on unit U2, then the possibility of executing the two operations in parallel is precluded due to the introduced resource conflict. Note that executing the MUL or ADD operations on the same unit as the SUB operation does not preclude any possible parallelism because there is a data dependency from the MUL and ADD operations to the SUB operation.

Three unpruned paths result from pruning the search space of this example. These paths represent the split-node functional unit assignments that AVIV should explore in depth. The three assignments are:

1: SUB on unit U2 ($N_2$), MUL on unit U2 ($N_3$), ADD on unit U1 ($N_5$)

2: SUB on unit U2 ($N_2$), MUL on unit U2 ($N_3$), ADD on unit U2 ($N_6$)

3: SUB on unit U2 ($N_2$), MUL on unit U2 ($N_3$), ADD on unit U3 ($N_7$)

The node labels (appearing in parentheses) correspond to the node labels in the Split-Node DAG of Figure 5-5.

Once a split-node functional unit assignment is made, any operation nodes that are not included in the assignment are irrelevant and can be eliminated from the Split-Node DAG. All edges connected to the removed nodes and any data transfer nodes along those edges may also be removed. The remaining relevant portions of the Split-Node DAG for the first functional unit assignment are illustrated in Figure 5-8. Each transfer node has been labeled with its corresponding unit. The register-to-register

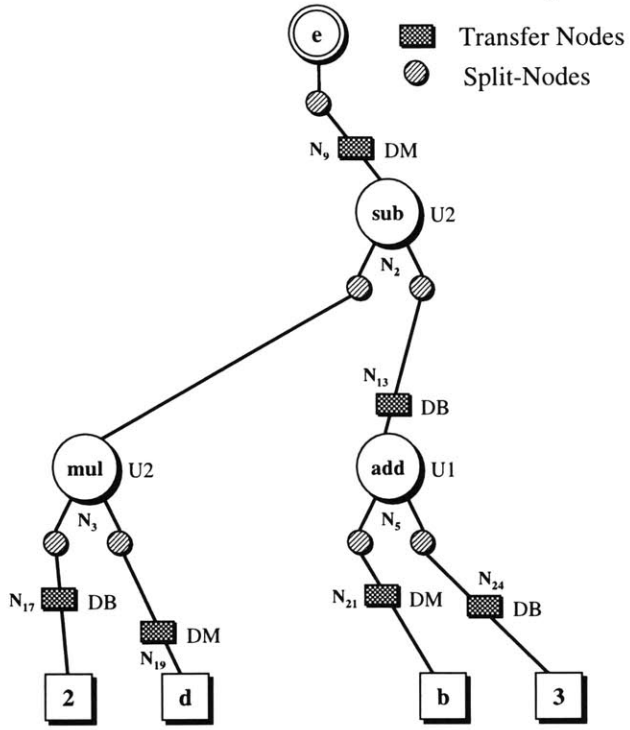



Figure 5-8: **The Split-Node DAG after functional unit assignment**

data transfer ($N_{13}$) and the two load constant (move immediate) data transfers ($N_{17}$ and $N_{24}$) correspond to databus (DB) move operations. The memory load and store operations ($N_{19}$, $N_{21}$, and $N_9$) are data memory (DM) operations.

### 5.2.2 Adding the Required Transfers

For each split-node functional unit assignment selected, the required data transfer nodes are added. In the case of a single data transfer path, this step is straightforward because for each pair of nodes that requires a data transfer between them, there is exactly one possible data transfer node that can be selected.

This is clearly illustrated by the reduced Split-Node DAG of Figure 5-8. This reduced Split-Node DAG corresponds to the functional unit assignment consisting of the operation nodes $N_2$, $N_3$, and $N_5$. The target processor of this Split-Node DAG contains a single data transfer path. Therefore, the required transfers are simply the

transfers along the edges of the reduced Split-Node DAG. Specifically, the required transfers are:

- Node $N_{13}$ which transfers the result of node $N_5$ to node $N_2$.
- Nodes $N_{17}$ and $N_{19}$ which load the operands of node $N_3$.
- Nodes $N_{21}$ and $N_{24}$ which load the operands of node $N_5$.
- Node $N_9$ which stores the result of node $N_2$ in variable e.

These transfer nodes are added to the functional unit assignment to form a complete assignment of target processor nodes for the input application.

The data transfers for the second and third functional unit assignments are determined in the same manner. The resulting complete assignments, including both the functional unit assignments and their corresponding data transfer assignments, are listed below:

1: $N_2$, $N_3$, $N_5$, $N_9$, $N_{13}$, $N_{17}$, $N_{19}$, $N_{21}$, $N_{24}$
2: $N_2$, $N_3$, $N_6$, $N_9$, $N_{17}$, $N_{19}$, $N_{22}$, $N_{25}$
3: $N_2$, $N_3$, $N_7$, $N_9$, $N_{16}$, $N_{17}$, $N_{19}$, $N_{23}$, $N_{26}$

Determining the required data transfers is not as simple for architectures containing multiple transfer paths. In such architectures, there may be more than one possibility for any required data transfer. The problem of selecting the best transfer paths resembles the problem of selecting the best split-node assignments. In both of these cases the number of options grows multiplicatively. In order to reduce the number of options that must be explored in depth, here too, a heuristic cost function is applied. The heuristic calculates the amount of foregone parallelism in each possible data transfer node combination. It selects the combination that minimizes the foregone parallelism. One or more selections are permitted. However, the greater the number of combinations selected, the greater the number of solutions that need to be explored in detail.

Create all possible combinations of transfer nodes such that exactly one transfer node is included for each path that requires a data transfer.

For each combination:
- Calculate the total amount of foregone parallelism using equation 5.8.

Select the combination that allows for the most parallelism by selecting the lowest cost combination.

Figure 5-9: **Algorithm for selecting the best transfer nodes for functional unit assignment**

A description of the algorithm used to select the best transfer node combination is provided in Figure 5-9. The cost function used to differentiate between each of the possible data transfer node combinations is given below:

$$\text{Total Cost} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} FP_{ij} \text{ (n = number of transfer nodes)} \qquad (5.8)$$

### 5.2.3 Maximal Groupings

For the remainder of this chapter, the term *assignment* will be used to refer to the combination of the nodes in the functional unit assignment and the nodes in the data transfer assignment. Note that each assignment corresponds to a single implementation of the Split-Node DAG. It can be thought of as a machine-specific DAG in which each node is associated with a single target processor operation.

Once the best set of assignments have been identified, each assignment in the set should be explored in depth. The goal of the detailed exploration is to determine which assignment results in the minimum-cost set of target processor instructions required to cover all of its nodes. This assignment is selected as the final cover of the Split-Node DAG.

The detailed assignment exploration begins by examining the nodes in a given assignment and merging them into groups of nodes that can be executed in parallel

on the target processor. Each grouping corresponds to a VLIW instruction. The goal is to identify a minimal-cost set of instructions that can cover all of the nodes in the assignment. The minimal-cost set of instructions is selected as the solution for the given assignment.

In order to reduce the total cost of the instructions required to cover the Split-Node DAG, it is preferable to select instructions that exploit the parallelism provided in the target processor. Therefore, the groupings of nodes that are examined are all the *maximal groupings* of parallel nodes. In other words, subsets of a larger grouping are not considered as a possible grouping. The maximal groupings combine operation nodes with data transfer nodes. Note that every node in the assignment being explored is covered by at least one grouping. Also, it is possible for a grouping to contain only one node.

### Generating the Maximal Groupings

In order to generate the maximal groupings, an $N \times N$ matrix representing pairwise parallelism is created, where $N$ is the number of nodes in the current assignment of the Split-Node DAG. This matrix contains a 0 for element $[i, j]$ if $N_i$ can be executed in parallel with $N_j$, and 1's elsewhere. Two operations can be performed in parallel if they use different functional units and have no directed path between them. Figure 5-10 (a) shows the matrix for one of the selected assignments for the Split-Node DAG of Figure 5-5. The assignment consists of nodes $N_2$, $N_3$, $N_5$, $N_9$, $N_{13}$, $N_{17}$, $N_{19}$, $N_{21}$, and $N_{24}$. Row $N_5$, for example, specifies that an `ADD` on unit `U1` ($N_5$) can be performed in parallel with a `MUL` on unit `U2` ($N_3$) or with either of the `MUL`'s load operations ($N_{17}$ or $N_{19}$).

The matrix could also be represented as a graph whose vertices are the nodes in the assignment. An edge exists between any pair of nodes whose matrix entry is zero. In such a representation, finding the maximal groupings corresponds to finding the *maximal cliques* of the graph.

**Definition 1** [11] *"A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in $E$. In other words, a clique*

| | $N_2$ | $N_3$ | $N_5$ | $N_9$ | $N_{13}$ | $N_{17}$ | $N_{19}$ | $N_{21}$ | $N_{24}$ |
|---|---|---|---|---|---|---|---|---|---|
| $N_2$ | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $N_3$ | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| $N_5$ | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $N_9$ | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| $N_{13}$ | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| $N_{17}$ | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| $N_{19}$ | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| $N_{21}$ | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| $N_{24}$ | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

( a )

$(C_1 : N_2)$
$(C_2 : N_9)$
$(C_3 : N_3, N_{13})$
$(C_4 : N_3, N_5)$
$(C_5 : N_3, N_{21}, N_{24})$
$(C_6 : N_5, N_{17}, N_{19})$
$(C_7 : N_{13}, N_{19})$
$(C_8 : N_{17}, N_{21})$
$(C_9 : N_{19}, N_{24})$

( b )

Figure 5-10: **Generating the maximal cliques (a)** Pairwise parallelism matrix **(b)** Resulting maximal cliques

*is a complete subgraph of G".*

A *clique* refers to a grouping of nodes that can all be executed in parallel. This implies that the matrix entry of every pair of nodes in the clique is zero. A *maximal clique*, or *maximal grouping*, refers to the largest possible grouping of nodes that can be executed in parallel. Thus, a maximal clique of the graph described above, is a clique that is not a subset of any other clique. Visually, this can be pictured by reordering the rows and columns of the pairwise-parallelism matrix in order to find the largest squares of zeroes. The reordering of the rows must be identical to the reordering of the columns. Squares of zeroes that are not a subset of any larger square correspond to a maximal clique. Finding all of the maximal cliques is an NP-complete problem [18].

The maximal cliques ($C_i$s) generated for this example are shown in Figure 5-10 (b). Note that due to data dependencies, nodes $N_2$ and $N_9$ cannot be executed in parallel with any of the other nodes in the assignment. Thus, their corresponding cliques,

$C_1$ and $C_2$, contain a single node. Also note that none of the cliques contain more than three nodes because the given example has a maximum parallelism of three if the functional unit assignments are taken into consideration. Although, as shown in Figure 5-8, there exist four independent load operations (two load constant operations that are assigned to the databus, and two memory load operations that are assigned to the data memory) at the bottom of the Split-Node DAG, these operations are not merged into a four node maximal clique. The reason is that operations that are assigned to the same unit cannot be executed in parallel. By setting the matrix entry to one for any pair of nodes assigned to the same unit, it is ensured that such nodes cannot be merged into the same clique.

The pseudo-code shown in Figure 5-11 describes the algorithm for generating all of the maximal cliques using the pairwise parallelism matrix. The algorithm consists of setting up initial cliques containing a single element, and then building these cliques up into maximal cliques. Setting up the initial cliques is performed by the `generate_max_cliques` function. It iterates through all nodes in the assignment as the starting clique node. It calls the main routine `gen_max_clique` with each of the initial cliques together with the index of the starting node.

The main routine greedily expands the input clique by adding all nodes that do not preclude the addition of any other node to the current clique. When no more nodes can be added that satisfy this criterion, the `gen_max_clique` function loops through all nodes in the assignment. For each node that can be executed in parallel with all of the nodes already in the clique, it adds the node and recursively calls itself in order to expand the clique further. When no new nodes can be added, a maximal clique has been found.

In order to avoid generating many duplicate maximal cliques, the `gen_max_clique` routine has a pruning condition that identifies cliques that have already been encountered. The pruning condition is ($i < index$). The pruning condition follows from the fact that if i < index, then the `gen_max_clique` routine has already generated all the cliques that would be generated by the current (recursive) call. Therefore, the current branch can be terminated. For example, suppose that the current clique is

```
generate_max_cliques {
   for (i = 0; i < number_of_nodes; i++) {
      first node of clique = node i;
      gen_max_clique( clique, i );
   }
}

gen_max_clique( clique, index ) {
   for (i = 0; i < number_of_nodes; i++) {
      if (i can be executed in parallel with all of the nodes in
         the current clique)
         if (adding i will not preclude adding any other node)
            if (i < index)        // Pruning condition
               return;
            else
               add i to clique;
   }

   for (i = 0; i < number_of_nodes; i++) {
      if (i can be executed in parallel with all of the nodes in
         the current clique)
         gen_max_clique(clique with i added, MAX(i, index));
   }
}
```

Figure 5-11: **Pseudo-code for generating maximal cliques**

clique $c = \{j\}$ and that node $i$, which does not preclude the selection of any other node, is being added to the clique. If $i < j$, then the routine has already generated maximal cliques from clique $\{i, j\}$, when it started with clique $\{i\}$ as the seed.

### Reducing the Number of Maximal Cliques Generated

The generation of the maximal cliques is an NP-complete problem that is being solved exactly. As a result, it is the most time consuming portion of AVIV's code generation algorithm. In order to improve the runtime of the algorithm, a heuristic that limits the nodes that may be added to a clique is used. This heuristic is based on the observation that in the optimum solution, nodes in widely different levels of the Split-Node DAG (measured from both the source and sink nodes of the Split-Node DAG) tend not to be scheduled in the same instruction. Therefore, restricting the allowable variance in the levels of nodes that can be merged into a single maximal clique, should in most cases not affect the optimality of the solution. This heuristic decreases the runtime because it results in the generation of fewer and smaller maximal cliques.

### Eliminating Illegal Instructions

The maximal clique generation phase merges all nodes that have no data dependence between them and are executed on different functional units into single instructions (i.e., cliques). Merging based solely on this criterion is insufficient to guarantee the *validity* of the instructions on the target processor. Illegal groupings are possible, and they are described in the Constraints section of the ISDL description. To avoid illegal groupings, each proposed instruction must be compared with the constraints of the target processor. If the constraints are not satisfied, then the illegal instruction, or maximal clique, is split into instructions with smaller cliques. This is repeated until all of the constraints are satisfied. The process of determining whether or not an instruction satisfies all of the constraints is described in detail in Chapter 7.

In the example target architecture shown in Figure 4-6, there exists a constraint that specifies that any databus operation cannot be executed in parallel with any data memory operation. The reason is that both the databus and the data memory

operations require the use of the databus which is a shared resource. As a result, the parallel execution of such operations must be explicitly prohibited through a constraint.

The maximal cliques that were generated for the Split-Node DAG functional unit assignment of Figure 5-8 are listed again below. Here they are shown with the results of comparing each clique to the target processor constraints.

| Clique | Result |
|---|---|
| $(C_1 : N_2)$ | Valid |
| $(C_2 : N_9)$ | Valid |
| $(C_3 : N_3, N_{13})$ | Valid |
| $(C_4 : N_3, N_5)$ | Valid |
| $(C_5 : N_3, N_{21}, N_{24})$ | Invalid |
| $(C_6 : N_5, N_{17}, N_{19})$ | Invalid |
| $(C_7 : N_{13}, N_{19})$ | Invalid |
| $(C_8 : N_{17}, N_{21})$ | Invalid |
| $(C_9 : N_{19}, N_{24})$ | Invalid |

In order to satisfy the constraints, all cliques containing both databus and data memory operations must be split into smaller cliques that do not merge the two types of operations. A *valid* maximal clique is defined as a maximal clique that satisfies all of the constraints. These include cliques that were subdivided in order to satisfy the constraints. The resulting *valid* maximal cliques for this example are shown below.

$(C_1 : N_2)$ $(C_5 : N_3, N_{21})$

$(C_2 : N_9)$ $(C_6 : N_3, N_{24})$

$(C_3 : N_3, N_{13})$ $(C_7 : N_5, N_{17})$

$(C_4 : N_3, N_5)$ $(C_8 : N_5, N_{19})$

Note that in the case of pairwise constraints (i.e., constraints between two operations), it may be preferable to check the constraints prior to generating the maximal

cliques rather than afterwards. This can be achieved by modifying the pairwise parallelism matrix so that it only contains a zero entry for pairs of nodes that: (1) do not have a data dependency between them, (2) are assigned to different functional units, and (3) do not have a pairwise constraint between them. This would reduce the number of invalid maximal cliques generated. In order to modify the pairwise parallelism matrix in this manner, every pair of operations that satisfies the first two points would have to be compared to all of the pairwise constraints. The matrix entry corresponding to such a pair of operations would only be set to zero if the pair of operations satisfied all of the pairwise constraints (in addition to the first two points listed above). The maximal cliques would then be generated from the resulting matrix. This would be followed by another round of constraint checking for all constraints that depend on more than two operations interacting with each other.

### 5.2.4 Selecting a Minimum-Cost Set of Maximal Cliques

Once the *valid* maximal cliques have been identified, the next step is to find the minimum-cost set of valid cliques that cover all of the nodes in the assignment. The algorithm used to cover the nodes is presented in Figure 5-12. A description of the components of the algorithm is provided below.

AVIV's covering algorithm begins with an empty solution set. It then selects a maximal clique that covers the largest number of remaining uncovered nodes whose children have all been covered and whose register requirements do not exceed the available resources. Selecting nodes whose children have all been covered implies that the nodes at the bottom of the Split-Node DAG will be scheduled before nodes that depend on them. This selection criterion automatically creates a valid schedule as the cliques are selected. The available register resources are determined by performing a liveness analysis on the selected nodes and maintaining a running upper bound on the number of required registers for each register bank. After selecting a clique, the clique is reduced by placing the nodes that are not ready to be scheduled into their own separate clique. In addition, the nodes that are covered by the selected clique are eliminated from the remaining cliques. This selection process is repeated until all

```
For each assignment:
   while any nodes in assignment remain uncovered {
      for ( i = 0; i < number of max cliques; i++ ) {
         count = number of nodes that are ready for scheduling
                 and have available register resources
         if ( count > best count )
            best clique = current clique
         else if ( ( count == best count ) and ( count ! = 0 ) )
            if ( lookahead cost function < best result of lookahead function )
               best clique = current clique
            else if ( lookahead cost function == best result of lookahead function )
               if ( num nodes that share parent with already covered nodes >
                    best num nodes that share parent with already covered nodes )
                  best clique = current clique
               else if ( num loads that reload spill < best num loads )
                  best clique = current clique
               else if ( total level from top > best total level from top )
                  best clique = current clique
      } /* end of for loop */

      if a new best clique was found
         Add best clique to current solution (only include nodes that are ready
         for scheduling)
         Remove covered nodes from remaining cliques

         Increment total number of instructions by the number of hardware
         instruction words required to represent the nodes of best clique
      else
         Select best node to spill
         Regenerate maximal cliques with added load and spill nodes
   } /* end of while loop */

Select assignment that resulted in the minimum total number of instructions.
```

Figure 5-12: **Algorithm for covering the nodes in an assignment using a minimum-cost set of cliques**

of the nodes have been covered.

An accurate instruction count is maintained as the cliques are selected. If any of the selected cliques require *additional* binary words to represent long constants, such as target addresses, the instruction count is adjusted accordingly.

If several cliques cover an equal number of nodes, the algorithm differentiates among them by using a lookahead cost function. This function estimates the number of cliques required to cover the remaining nodes assuming that the clique under consideration was added to the solution set. If the lookahead cost function results in another tie, then the algorithm attempts to optimize the following criteria:

- Give precedence to cliques that contain the largest number of nodes that share a parent with an already covered node. The reason for this provision is that if all of the children of a node have been covered, then the parent node can be scheduled. Scheduling the parent will typically reduce the overall storage resources required provided that the child nodes are no longer live.

- Minimize the number of load operations that reload spilled values in the clique. This attempts to schedule the loads as late as possible so that the scarce resources remain available for other operations.

- Give precedence to cliques whose nodes' *level from the top* is greater than other clique nodes. This criterion attempts to cover the longest path of uncovered nodes first.

In the event that no further cliques can be selected because the required register resources are unavailable, spills to memory must be introduced. The algorithm determines which covered node's result should be spilled. The decision is based on the most needed resource and the number of parent nodes that would later require the spilled value to be reloaded from memory. Once the node that should be spilled is selected, the required load and spill nodes are added to the Split-Node DAG, and any transfer nodes that are no longer needed are removed. Figure 5-13 shows a simple example of adding loads and spills to a Split-Node DAG. In this example, the `add`

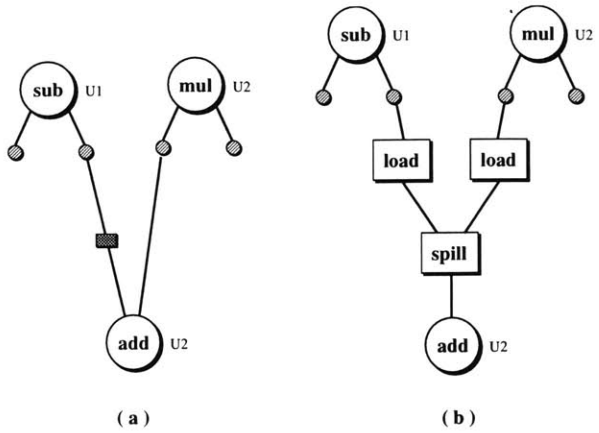



Figure 5-13: **Inserting loads and spills into the Split-Node DAG (a)** the original Split-Node DAG, **(b)** the Split-Node DAG augmented with load and spill nodes

node is selected as the node to spill, and the transfer node between the `add` and `sub` nodes is removed. New maximal cliques are then generated for all remaining uncovered nodes including the new load and spill nodes. The covering algorithm continues with the new maximal cliques and the remaining uncovered nodes.

### 5.2.5 The Covering Solution

The assignment that required the minimum-cost set of cliques to cover all its nodes is chosen as the final solution. The order in which the cliques were selected determines a schedule of the cliques (i.e., instructions). The final covering solution is thus a minimal-cost set of reduced maximal cliques that cover the Split-Node DAG. At this point, the following tasks have been completed: (1) functional unit assignment, (2) merging operation and data transfer nodes into VLIW instructions, (3) register bank allocation including the addition of load and spill nodes when necessary, and (4) scheduling. The remaining task is that of detailed register allocation.

### 5.2.6 Detailed Register Allocation

Detailed register allocation is performed using a liveness analysis of the nodes in the final solution. Any conventional graph coloring algorithm [9] could also be used to allocate the registers. It is guaranteed that a register, within the appropriate register bank, will be available for each node. This is the case because in the instruction selection and scheduling step the register resource requirements were analyzed, and loads and spills were inserted when necessary.

### 5.2.7 Peephole Optimization

If spills to memory were required, then in the initial pass of the algorithm the spill operations would have been scheduled as individual instructions. The reason is that the Split-Node DAG is covered from the bottom-up, and spill operations are only inserted when no further nodes can be covered. This implies that a spill operation will be scheduled on its own even though it could potentially be merged with earlier operations. In order to improve the final schedule, the algorithm attempts to reoptimize the final solution by determining whether the spill instruction could have been merged with earlier instructions. If it succeeds in merging the spill with an earlier instruction, then the possibility of further optimization exists. In this case, the algorithm attempts to move operations from the instructions that were scheduled after the spill into the empty slots in the instructions following the spill operation provided that all the dependency constraints are still satisfied. This additional optimization may, or may not, reduce the final number of required instructions.

Peephole optimization is also useful if more than one spill operation is required. In this case, it is possible that the algorithm will not find the best ordering of the instructions. This occurs because the algorithm inserts one spill operation at a time and then attempts to cover as many of the nodes as possible before inserting an additional spill operation. This order of events may result in the following partial schedule:

```
spill1
operationA
spill2
operationB
```

It is possible that if the second spill operation is moved earlier in the schedule, then operations A and B could be merged into a single instruction. This yields the following shorter schedule:

```
spill1
spill2
operationA and operationB
```

The difference between this optimization and the previous one is that here the second spill operation is not being merged with an earlier instruction. It is simply being moved to an earlier slot in the schedule provided that such a reordering still satisfies all of the dependency constraints.

## 5.3 Alternate Implementation

Section 5.2.4 described the method used by AVIV to select the best combination of cliques to cover each assignment of the Split-Node DAG. This method covers the Split-Node DAG from the bottom-up so that the selected cliques automatically produce a valid schedule.

An alternate implementation would be to:

1. Arbitrarily select a clique that covers as many uncovered nodes as possible from the Split-Node DAG.

2. Remove all covered nodes from the remaining cliques.

3. Repeat steps 1 and 2 until all nodes are covered.

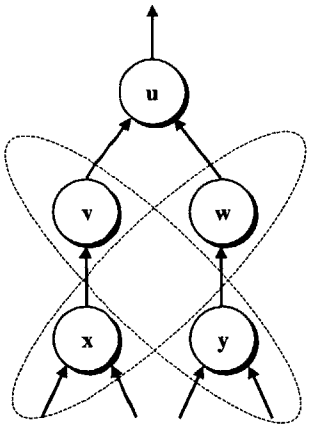



Figure 5-14: **Cycles in the clique selections**

The problem with such an implementation is that if the Split-Node DAG is not covered from the bottom-up, then the arbitrary selection of the largest remaining clique could lead to a cycle in the schedule which is not valid. For example, consider the DAG of Figure 5-14. If the first selected clique covers nodes `x` and `w`, and the second selected clique covers nodes `v` and `y`, then although both selected cliques were valid cliques the resulting schedule is not valid because it contains a cycle. In order to fix this problem, step 1 of the algorithm must be modified to ensure that each selected clique does not result in a cycle with any of the previously selected cliques. This modification slows the algorithm significantly and does not produce higher quality results than the algorithm used by AVIV. Furthermore, this implementation requires additional scheduling and register allocation phases after clique selection. This is because cycle-free clique selection does not establish an ordering for the cliques. It simply establishes that a valid ordering exists. In addition, register allocation cannot be performed until the schedule is determined.

## 5.4 Handling Complex Operations

The Split-Node DAG supports the representation of complex operations. The main difference between handling basic and complex operations is that in the case of com-

plex operations, not every split node needs to be covered. For example, consider the Split-Node DAG of Figure 5-15 which contains a multiply-accumulate (`mac`) complex operation in addition to the basic operations (`add`, `sub`, and `mul`). For the sake of simplicity, the data transfer nodes have been left out. Furthermore, it is assumed that each of the operations can only be executed on one functional unit. Notice that the `mac` node's split-nodes connect to the `add` node's parent and to the `mul` node's children. It bypasses the `mul` node's parent split-node. As expected, this representation implies that if the `mac` node is to be covered, then the `mul` node should not be covered.

The only complication introduced by the complex operation representation in the Split-Node DAG is that if multiple instances of complex operations exist, then one must ensure that there is no overlap between them. If there is an overlap, then the overlapped node must be duplicated as illustrated by the shaded nodes in Figure 5-15. If the `mac` operation were not available, then both of the `mul` nodes could be merged into a single node.

Introducing complex operations into the Split-Node DAG also requires a slight modification of the function that prunes the search space of possible functional unit assignments. The modification required is that the incremental cost of a complex operation should be compared to the summation of the incremental cost of each of the basic operations that comprise the complex function. Thus, the `mac` operation's incremental cost should be compared to the incremental cost of the `add` plus the incremental cost of the `mul` in order to determine which paths to prune. Furthermore, if any nodes were duplicated to avoid overlapping complex operations, then the incremental cost of paths that do not contain any of the overlapping complex operations should evaluate the cost as if the node was not duplicated. This may reduce the cost of such paths because it would require covering fewer nodes. After the best assignments have been selected, the remainder of the covering procedure is identical to the case where no complex operations exist.

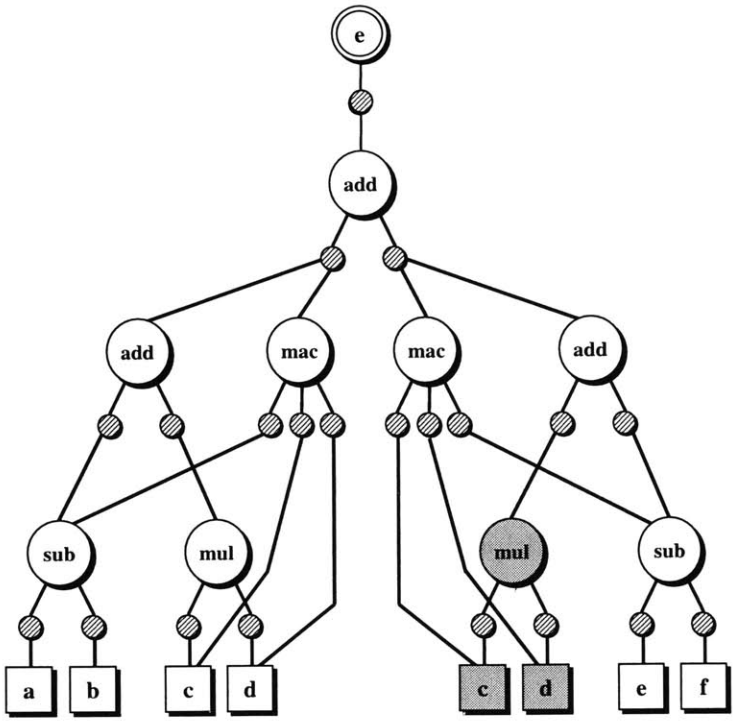



Figure 5-15: **Incorporating complex operations into the Split-Node DAG** The shaded nodes are nodes that were duplicated as a result of introducing the complex operation.

## 5.5 Handling Non-Uniform Operation Latencies

The covering algorithm described in this chapter considers all operation latencies to be equal. This implies that at each time step of the schedule, all functional units are available and can have an operation scheduled on them in the next instruction. This algorithm can be easily adapted to handle non-uniform operation latencies. Consider, for example, an architecture in which a multiply operation takes four cycles, and all other operation latencies are one cycle. To generate code for such an architecture, every time an instruction contains a multiply operation, the next three scheduled instructions cannot assign an operation to the unit that executes the multiply. In order to incorporate this type of restriction into AVIV's covering algorithm, the algorithm must keep track of the availability of each functional unit as the schedule is being created. In addition, the clique selection heuristic must also be modified slightly. This

modification affects the determination of the number of nodes of each clique that can be covered in the current time step. Instead of counting all nodes whose children have been covered and whose register requirements are available, the count must not include any nodes that require a functional unit that is not available in the current time step. The rest of the clique selection criteria remain unchanged.

# Chapter 6

# Control Flow

The previous chapters described the process of generating target processor code for basic blocks. This chapter describes how control flow is handled by AVIV. It describes how the code generated for the various basic blocks is combined into a coherent program representing an entire procedure. In particular, it examines the if-then-else and for-loop control flow constructs and describes how they are handled by AVIV. Other control flow constructs can be handled in a similar fashion. This chapter concludes with a presentation of several possible control flow optimizations.

## 6.1 The If-Then-Else Control Flow Construct

A program can be thought of as a number of basic blocks connected through control flow information. For example, an *if-then-else* statement consists of three basic blocks. The first basic block consists of the code for the *condition* of the *if* statement. Its value determines whether the *then* basic block or the *else* basic block should be executed. Figure 6-1 presents an *if-then-else* statement and its corresponding low-level SUIF representation.

In the SUIF representation of the if-then-else statement, the variables `b` and `c` are first loaded with the constant values 3 and 4 respectively. The variable `e1` represents a temporary variable whose value is the result of a signed less-than operation (`sl`) between variables `b` and `c`. The `bfalse` operation implies that if the value of `e1` is

```
b = 3;
c = 4;
if (b < c) {
    a = b + c;
}
else {
    a = b - c;
}
```

(a)

```
ldc b = 3
ldc c = 4
bfalse e1, L:L3
    e1:  sl b, c
add a = b, c
jmp L:L4
lab L:L3
sub a = b, c
lab L:L4
```

(b)

Figure 6-1: **Low-level SUIF representation of if-then-else statement**
(a) If-then-else statement (b) SUIF representation of if-then-else statement

`FALSE`, then a branch to label `L3` is performed in order to execute the *else* clause. Otherwise, the *then* clause is executed by continuing sequentially through the code with the `add` operation. A jump to label `L4` follows the `add` in order to skip over the code corresponding to the *else* clause.

This sequence of operations is translated by SPAM routines into the set of basic block DAGs illustrated in Figure 6-2. A Control Flow Graph (CFG) maintains the relationships between the multiple basic blocks. The CFG for this example is shown in Figure 6-3.

Each of the basic block DAGs is then converted into Split-Node DAGs for the target processor. The target processor for this example is shown in Figure 6-4. It is an extended version of the example architecture of Figure 3-3. The complete ISDL specification for this new architecture is presented in Appendix B. The main difference between the two architectures is that the architecture of Figure 6-4 includes a field for control flow. In particular, it can perform `bfalse` and `jmp` operations. In addition, unit `U1` has a `not` operation added to its capabilities, a `sl` operation has been added to unit `U3`, and a `st_im` (store immediate constant) operation has been added to the data memory. The resulting Split-Node DAGs are illustrated in Figure 6-5.

Note that although the architecture contains a single `bfalse` operation, there are three nodes representing this operation in the Split-Node DAG. This occurs be-

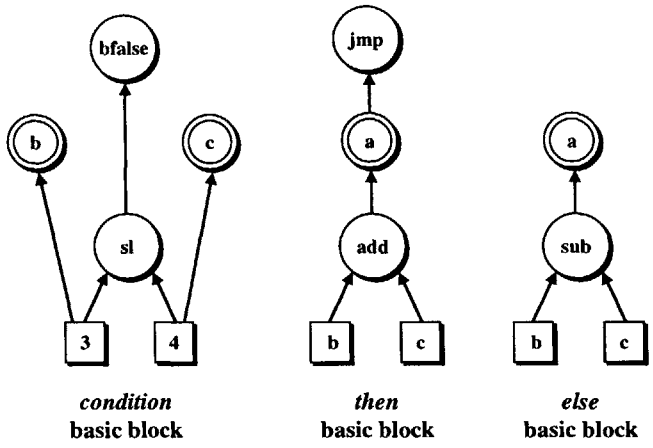



Figure 6-2: **Basic block DAGs for if-then-else statement**

cause the assembly syntax description of the `bfalse` operation is '`bfalse SRC, NAME`', where the `NAME` token represents the target label, and the `SRC` non-terminal represents any of the three register files `U1`, `U2`, or `U3` as the source operand. This is explicitly represented in the Split-Node DAG by splitting the `bfalse` node into three nodes, each representing a different source operand. In order to simplify the figure, however, the Split-Node DAGs created only illustrate the data transfers on one of the databuses rather than both. The dotted lines in the Split-Node DAGs represent control flow dependencies, whereas the solid lines represent data dependencies.

A mechanism for handling control flow is required in order to generate code for the complete if-then-else statement consisting of various basic blocks. In AVIV, control flow is handled in a straightforward manner with no optimizations across basic block boundaries. Possible optimizations are described in Section 6.3. In order to generate correct code, the control flow graph is traversed in the same order as the SUIF representation of the multiple basic blocks. This allows the control flow optimizations performed by the SUIF compiler to be retained.

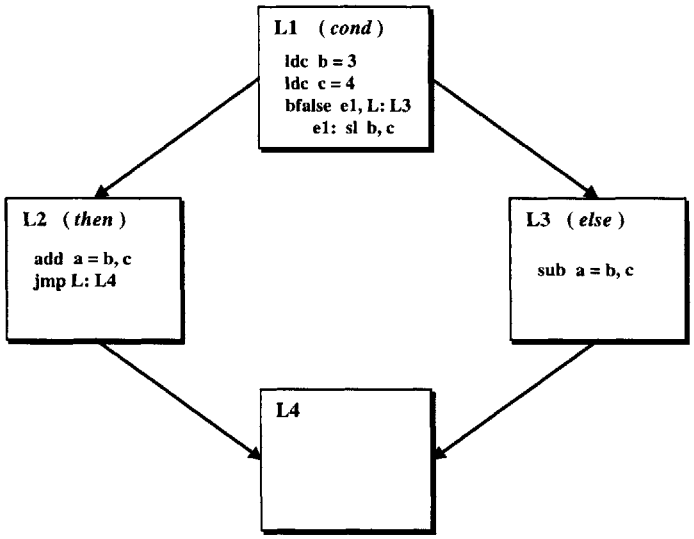



Figure 6-3: **Control flow graph for if-then-else example code**

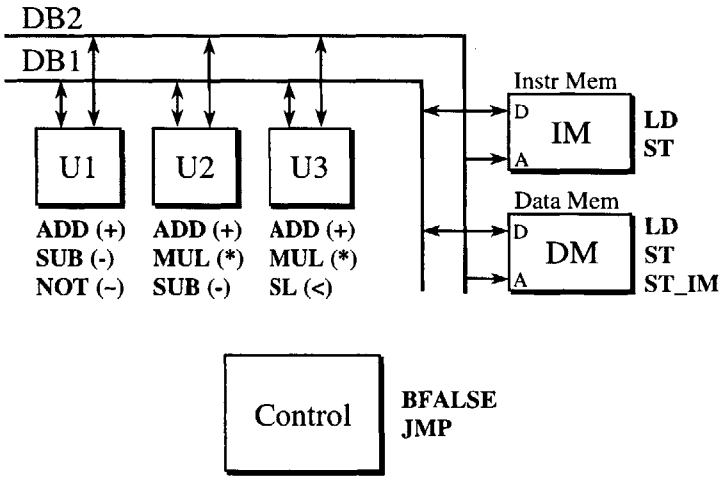



Figure 6-4: **Example target architecture with support for control flow**

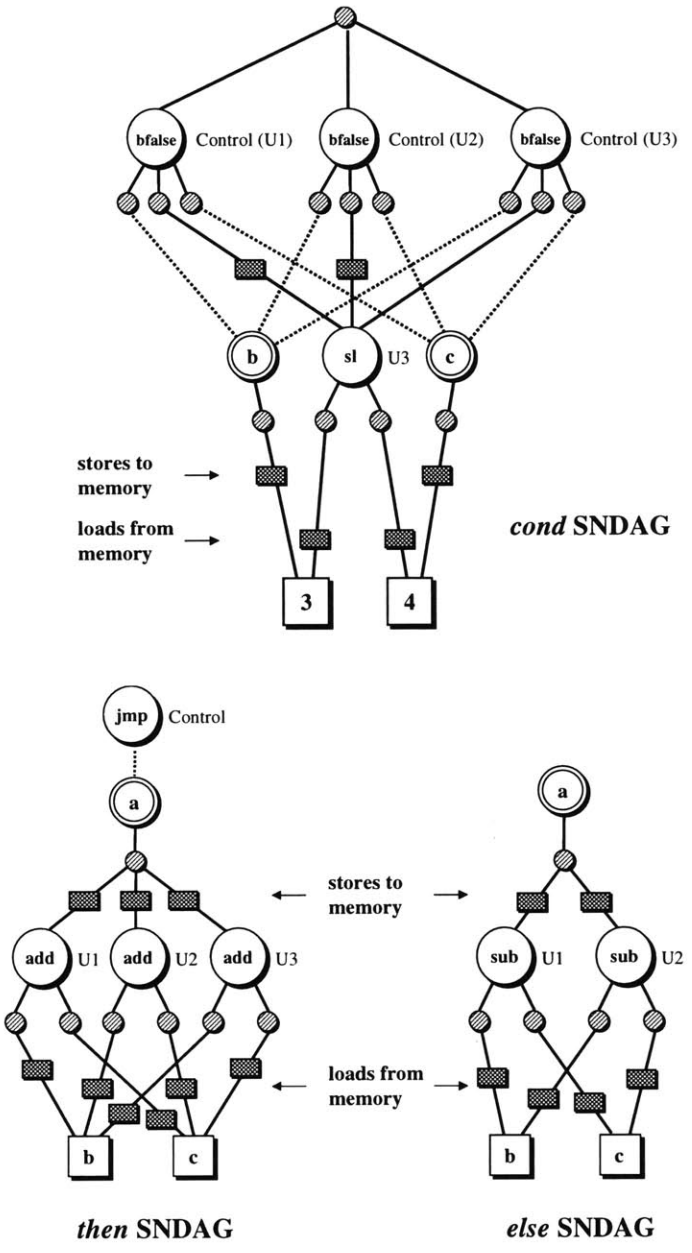



Figure 6-5: **Split-Node DAGs for if-then-else example code**

The order of traversal for an if-then-else control flow graph is to traverse the *condition* node first, followed by the *then* node, the *else* node, and finally, the node that follows the if-then-else nodes. For the CFG of Figure 6-3, the order of traversal is L1, L2, L3, L4. Variables that span basic block boundaries are denoted as primary inputs and outputs of the basic blocks. AVIV assumes that all primary inputs of a basic block are loaded from memory, and all primary outputs are stored to memory. Thus, communication between basic blocks occurs through memory. Each node of the control flow graph has a label associated with it. By outputting the label associated with each basic block before outputting the code corresponding to that basic block, and by traversing the control flow graph in the appropriate order, correct code is guaranteed. The code generated for this example is:

```
L1:
     { ... DB1_move_im 3, U3.R0; DB2_move_im 4, U3.R1;
           DM_st_im 3, 0; ... }
     { ... U3_sl U3.R0, U3.R1, U3.R0; DM_st_im 4, 1; ... }
     { Control_bfalse U3.R0, L3; ... }
L2:
     { ... DM_ld U1.R0, 0; ... }
     { ... DM_ld U1.R1, 1; ... }
     { ... U1_add U1.R0, U1.R1, U1.R0; ... }
     { ... DM_st U1.R0, 2; ... }
     { Control_jmp L4; ... }
L3:
     { ... DM_ld U1.R0, 0; ... }
     { ... DM_ld U1.R1, 1; ... }
     { ... U1_sub U1.R0, U1.R1, U1.R0; ... }
     { ... DM_st U1.R0, 2; ... }
L4:
```

The `DB1_move_im` and `DB2_move_im` operations are equivalent to the load constant

operations and are used to load the constants 3 and 4 into the register file of unit U3. The 'DM_st_im const, x' operation stores the constant const into memory location x. In this example, address 0 stores the constant 3 that corresponds to the primary output b. Address 1 stores the constant 4 that corresponds to the primary output c. The result of the sl operation is stored in register U3.R0. Based on the value of this register, the Control_bfalse operation determines whether or not to branch to label L3. Label L2 corresponds to the *then* clause of the *if* statement, label L3 corresponds the *else* clause, and label L4 corresponds to the code following the *if* statement. The basic blocks at labels L2 and L3 both load their operands from the appropriate memory location before executing the add or sub operation. Both blocks store their final result at address 2 corresponding to the primary output a. Basic block L2 then performs an unconditional jump to label L4 in order to skip the basic block at label L3.

## 6.2 The For-Loop Control Flow Construct

Another form of control flow is the *for-loop* construct. The control flow graph for the for-loop follows the SUIF representation of for-loops and is illustrated in Figure 6-6. The CFG of the for-loop consists of five nodes:

- **Initial Test Node:** The basic block of code corresponding to this node initializes the loop index variable to the loop lower bound. If it is not known at compile time whether the loop body will be executed at least once, then the initial test node also compares the lower and upper bounds of the loop. It causes control flow to bypass the body of the loop if the comparison result states that the loop will not be executed.

- **Landing Pad:** The landing pad code is executed exactly once prior to the execution of the body of the loop. Loop-invariant code can be moved from the *loop body* to the *landing pad* in order to execute that code once rather than on every iteration of the loop. The code in the landing pad is only executed if the

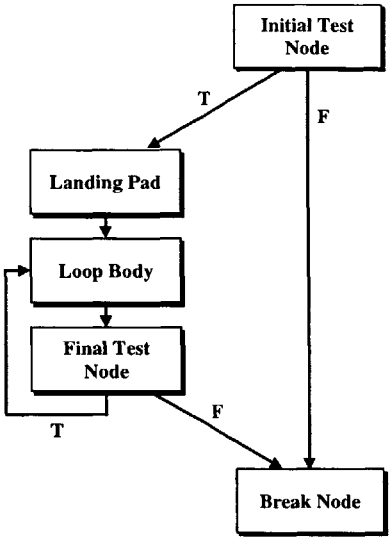



Figure 6-6: **Control flow graph for a for-loop**

body of the loop is executed at least once.

- **Loop Body:** This node contains instructions executed on every iteration of the loop.

- **Final Test Node:** This node contains code that increments the loop index variable by the step value and compares the resulting loop index to the loop upper bound. If the loop index is within the upper bound, then the flow of control is directed back to the *loop body*; otherwise, control flows to the *break node*.

- **Break Node:** The break node corresponds to the code that follows the for-loop construct.

In order to emit correct code for this control flow graph, the order of traversal of the CFG is the *initial test node*, *landing pad*, *loop body*, *final test node*, and finally the *break node*. Code is emitted for each basic block in the order specified. Note that the control flow embedded into the initial test node will ensure that if the loop should not be executed at all, the control will flow directly to the break node.

The rest of the control flow constructs, such as while loops and multi-way branches, are translated in a similar manner. The SUIF control flow optimizations are retained for all control flow constructs.

## 6.3 Possible Control Flow Optimizations

### 6.3.1 Trace Scheduling

A common control flow optimization is to try to predict the most commonly executed path and treat the set of basic blocks along that path as a single larger basic block. Larger basic blocks make it easier to extract higher levels of parallelism from the code. This optimization is commonly known as *trace scheduling* [15].

Consider the control flow diagram of figure 6-3. If path `L1, L2` is followed more frequently than path `L1, L3`, then it may be beneficial to consider the two blocks `L1` and `L2` as a single larger basic block. A larger basic block typically results in increased available parallelism. Thus, treating the two blocks as one larger block may result in more efficient code. The only complication with such a scheme is in correctly handling the execution of blocks not on the most likely path (in this example, block `L3`). The code in block `L1` must be executed before block `L3`. Therefore, block `L1` either has to be duplicated, or the code for the most common path must execute and then undo the effects of basic block `L2`. The former is generally easier to implement. Both options may result in a larger code size, but the resulting code may be more efficient in terms of the number of cycles required for it to execute.

### 6.3.2 Code Motion

Another possible control flow optimization involves *code motion.* This optimization moves code out of basic blocks that are not necessarily executed (e.g., `L2` or `L3`) into basic blocks that are always executed (e.g., `L1`) provided that the semantics of the code remain unchanged. Thus, in the if-then-else control flow diagram, if common code exists between blocks `L2` and `L3` and moving this code into block `L1` would not

affect the functionality of the code, then the common code can be moved into block L1. This optimization can increase the basic block size of shared code while decreasing the block sizes of unshared code. The resulting code may be more efficient in terms of both cycle count and code size.

It is also possible to move code that only exists in one of the blocks L2 or L3 provided that moving the code does not affect the other block. Suppose, for example, that the sl operation of block L1 could be executed with the add operation of block L2 which adds b and c. If block L2 is executed much more frequently than block L3, then moving the add operation into block L1 would improve the overall performance. The performance would improve because every time the path to L2 is followed, two fewer instruction are executed (Two load from memory instructions and one add instruction would be removed from block L2. However, one additional instruction in block L1 is required to load the values of b and c into the registers of the functional unit that will perform the add operation). Note that this code motion does not affect the functionality of block L3. If that path is followed, then the sub operation on variables b and c would be executed, and the result of that operation would be stored in variable a rather than storing the result of the add operation. The resulting code in this case would be:

```
L1:
     { ... DB1_move_im 3, U3.R0; DB2_move_im 4, U3.R1;
           DM_st_im 3, 0; ...  }
     { ... DB1_move_im 3, U1.R0; DB2_move_im 4, U1.R1;
           DM_st_im 4, 1; ...  } /* load operands for add */
     { ... U1_add U1.R0, U1.R1, U1.R0; /* add from L2 */
           U3_sl U3.R0, U3.R1, U3.R0; ...  }
     { Control_bfalse U3.R0, L3; ...  }
L2:
     { ... DM_st U1.R0, 2; ...  } /* loads and add removed */
     { Control_jmp L4; ...  }
```

```
L3:
        { ... DM_ld U1.R0, 0; ... }
        { ... DM_ld U1.R1, 1; ... }
        { ... U1_sub U1.R0, U1.R1, U1.R0; ... }
        { ... DM_st U1.R0, 2; ... }
L4:
```

### 6.3.3 Loop Optimizations

As described above, the SUIF format for the body of a loop consists of a landing pad in addition to the main loop body. The landing pad allows loop invariant code to be moved out of the main body of the loop and executed exactly once before the first execution of the loop body. The landing pad code is only executed if the loop body is executed at least once. This capability supports the optimization of loops by being able to shorten the main body of the loop so that the total execution time is improved. This optimization, however, may not affect the code size.

Another loop optimization is support for *zero-overhead looping hardware*. Embedded processors, in particular DSP architectures, often support *zero-overhead looping hardware* which is specialized hardware that supports the efficient execution of loops. Such hardware avoids having to perform (in software) an update and comparison of the loop index variable. It also avoids the need for a conditional branch instruction to the beginning of the loop. Zero-overhead looping hardware follows one of the following configurations [32]:

- *A one-word repeat buffer:* This configuration supports the repeated execution of one instruction. This instruction must be fetched from the instruction memory during the first iteration of the loop. Subsequently, the instruction is fetched from the repeat buffer.

- *An N-word repeat buffer:* This configuration supports the repeated execution of N or fewer instructions. The instructions are fetched from the instruction memory once and subsequently fetched from the repeat buffer.

- *Arbitrary-sized loop support:* In this configuration, a repeat buffer is not used. Thus, an arbitrary number of instructions may exist in the loop. However, each instruction in the loop must be fetched from the instruction memory during each iteration of the loop. This precludes the use of the bus for fetching other data unless a dedicated bus exists for the instruction memory.

The SPAM compiler supports the exploitation of zero-overhead looping hardware. This is achieved by providing a set of routines that can be used to modify the control flow graph of for-loops to use the specialized hardware. These modifications allow the loop index updating and comparison operations as well as the conditional branching operation to be eliminated from the loop. In order to describe the zero-overhead looping hardware to the SPAM compiler, the designer must (1) provide routines that determine whether a particular for-loop can be implemented on the specialized hardware, and (2) provide assembly instructions that control the looping hardware. If it is determined that a particular loop can be implemented on the zero-overhead looping hardware, then the SPAM compiler modifies the for-loop control flow graph accordingly. This involves inserting the assembly instructions that control the hardware into the *landing pad node*. In addition, the software operations that perform the updating of the loop index and conditional branching to the loop body are removed from the *final test node*.

# Chapter 7

# The Constraints Checker

Recall that in order to support the description of VLIW architectures, ISDL differentiates between *operations* and *instructions*. An operation is the smallest part of a program that can specify a data manipulation independently. An instruction is the smallest unit of control that can be fetched and issued to the hardware. In the case of VLIW architectures, an *instruction* may consist of multiple independent *operations*.

The *Instruction Set* section of an ISDL description provides a complete list of *operations* that can be executed on the target processor. These operations are divided into a set of *fields* where each field generally corresponds to a functional unit of the processor. Thus, operations within the same field are mutually exclusive because they share a common resource (i.e., the functional unit). Furthermore, operations from different fields can generally be executed in parallel unless they result in a conflict. All conflicts (i.e., constraints) are described in the *Constraints* section of the ISDL description. An *instruction* is formed by selecting one operation from each field provided that the selected combination of operations satisfies all of the processor's constraints.

During code generation, operations from different fields are merged into potential target processor instructions (i.e., maximal cliques) that attempt to maximize the amount of parallelism within the instruction. However, these potential instructions can only be used if they are valid. In order to test for their validity, a constraints checker is required. The constraints checker receives a potential instruction and com-

pares it to all of the constraints listed in the ISDL constraints section. It returns a boolean value of *true* or *false* specifying whether or not all of the constraints were satisfied. If a `TRUE` value is returned, then the instruction is valid, and it may be used to cover the nodes of the Split-Node DAG.

Verifying instruction validity through constraint checking is an NP-complete problem. It contains the problem of matching regular expressions with back referencing which has been proven to be NP-complete [55]. Regular expressions with back referencing are expressions that allow variable matching to be expressed. This means that in order to match such expressions all instances of a given variable must match the same substring. Although constraint checking is NP-complete it must be solved exactly; thus, the execution time it requires is exponential in the worst case.

## 7.1 The Constraints Syntax

Constraints are described as a set of boolean expressions, all of which must be satisfied for an instruction to be valid.

A constraint *basic expression* consists of a string with special purpose wildcards embedded in it. This string represents an ISDL operation. The value of the expression is `TRUE` if the string (including the wildcards) matches the assembly representation of the operation that is being compared to the constraint.

A basic expression can include any of the following components:

- *A constant string* - a sequence of ASCII characters that form a string. Strings match if any part of the operation string is identical to the constant string.
- *?* - a special character that matches any single character in the operation string.
- *+* - a special character that matches one or more characters in the operation string.
- *** - a special character that matches zero or more characters in the operation string.

- *A range operator* - consists of a set of characters enclosed in `[ ]` and matches any single one of these characters. Ranges can be abbreviated using the `'-'` character (e.g., the characters a, b, c, d, e can be abbreviated as `[a-e]`). Furthermore, ranges can be used to match any character not included in the range if the first character of the range is a `'^'` character.

- *A variable match* - the syntax for a variable match is `@[<int>]` where `int` is an integer between 0 and 9. The first occurrence of a variable match `@[x]` will match zero or more characters from the operation string and store them in a variable (a separate variable is used for each value of `x` between 0 and 9). All subsequent references to this variable, within the context of the same constraint, will only match the sequence of characters stored in the variable. Thus, the first occurrence defines the variable, and subsequent occurrences must match the same variable.

- *One expression followed by another* - such an expression will match if the first and second expressions match portions of the operation string and the second match begins exactly where the first match ended. This allows all forms of expressions listed above to be concatenated as many times as necessary to represent an entire constraint expression.

A match on a basic expression implies that the value returned by the expression is `TRUE`. If no match was found, then the return value is `FALSE`.

A constraint can consist of multiple basic expressions combined through logical operators. The following logical operators can appear in a constraint definition:

- $\sim$: The unary *negation* operator can precede a boolean expression. The resulting expression is `TRUE` if the input expression is `FALSE`, and `FALSE` otherwise.

- |: The binary *or* operator can appear between two boolean expressions. The resulting expression is `TRUE` if either of the input expressions is `TRUE`, and `FALSE` otherwise.

- &: The binary *and* operator can appear between two boolean expressions. The resulting expression is TRUE if both of the input expressions are TRUE, and FALSE otherwise.

The following example illustrates the syntax of constraint expressions. Consider the example VLIW target processor presented in Figure 3-3. The following constraint exists in that processor:

```
~(DB*_move U@[1].R*, U@[1].R*)
```

This constraint specifies that a databus move operation on either bus DB1 or DB2 *cannot* be used to move data from one register to another within the same register file.

In order to understand this constraint, the first step is to understand the format of the constraint expression. The format consists of a negation (~) operator followed by a basic expression that describes an ISDL operation. The negation operator results in the constraint specifying a situation that *cannot* occur on the target processor. The ISDL operation being described is a DB*_move operation with two operands. In the example target processor, DB*_move can match either a DB1_move or a DB2_move operation; therefore, the expression places a constraint on both databuses. The left and right operands are both specified using the same expression, U@[1].R*. The operand specification states that a variable match must occur in order to match this expression. This means that the substring replacing the @[1] portion of the expression, in both the left and right operands, must be identical for that portion of the expression to match. In other words, 'U@[1].R*, U@[1].R*' will match an expression such as 'U2.R1, U2.R3' (both of the variable 1 references were replaced with the character 2) as well as the expression 'U3.R2, U3.R2' (both of the variable 1 references were replaced by the character 3), but not the expression 'U1.R3, U2.R3' (the two references of variable 1 cannot be replaced by equal substrings). Note that the variable match only constrains the register file being used not the specific register within the register file (this is specified by the R* portion of the expression which can match any register number).

| Instruction | Result | Explanation |
|---|---|---|
| { U1_add U1.R1, U1.R2, U1.R3; ... } | TRUE | The instruction does not contain a DB1_move or DB2_move operation. Thus, the inner expression produces a FALSE result, and the negation operator produces a TRUE final result. |
| { ...; DB1_move U2.R1, U2.R3; ... } | FALSE | DB1_move matches the operation of the inner expression, and the character 2 matches the variable reference. Thus, the inner expression produces a TRUE result, and the negation operator produces a FALSE final result. |
| { ... DB1_move U1.R1, U2.R1; DB2_move U2.R2, U2.R3; ...} | FALSE | The DB1_move operation does not match the inner expression. The DB2_move operation matches the inner expression with a variable match on the character 2. As a result, the inner expression returns a TRUE value, and the negation operator produces a FALSE final result. |

Table 7.1: **Sample results of constraint checker**

The constraint described above could have been expressed more precisely as: ~(DB?_move U@[1].R+, U@[1].R+). However, ISDL assumes that the constraints will only be compared to valid assembly instructions of the target processor. Thus, the added precision is not required. Replacing the '?' and '+' symbols with the '*' symbol will yield the same results from the constraint checker.

Table 7.1 presents several sample instructions checked against the above constraint to determine whether or not they are valid instructions. In order to simplify the examples only the relevant fields of the instructions are shown.

A complication arises when variables are defined across '|' or '&' operators. If the same variable occurs in both the left and right subexpressions of an '&' operator, then all of the variable matches (for the same variable) must be identical in order to get a complete expression match. If, however, the same variable reference occurs in

both the left and right subexpressions of an '|' operator, then one must differentiate between the defining occurrences of the variable and subsequent references to the variable. In '&' expressions, the first occurrence of a variable in a constraint defines the value of the variable, and all subsequent occurrences of the variable must match the defining variable. However, if that first occurrence is within an '|' expression, then either the left or right side of the '|' expression can define the variable (i.e., an occurrence other than the first may define the variable). Occurrences of the variable after it has been defined, must match the defining value of the variable in order to get a complete expression match.

For example, consider the constraint, `(( A@[1] & B@[1] ) & C@[1] )`, and the instruction, { `Aaa; Baa; Cbb;` }. In this constraint, the first occurrence of variable 1 is within the `A@[1]` expression. The `A@[1]` expression can only match the `Aaa` portion of the instruction, thus the only possible definition of variable 1 is the string `aa`. Expression `B@[1]` must then match a portion of the expression using the same variable definition, and it matches `Baa` which is consistent with the variable definition `aa`. However, expression `C@[1]` cannot match any portion of the instruction using the variable definition `aa`, thus the result of comparing this instruction to the given constraint is `FALSE`.

Now consider the constraint, `(( A@[1] | B@[1] ) & C@[1] )`, where the first occurrence of variable 1 is within an '|' expression. This implies that either the left (`A@[1]`) or the right (`B@[1]`) subexpressions of the '|' can define the variable. The `C@[1]` expression must then match one of the possible definitions. Suppose the input instruction is { `Aaa; Bbb; Cbb;` }. For this instruction, variable 1 could be defined as the string `aa` by expression `A@[1]`, or as the string `bb` by expression `B@[1]`. The `C@[1]` expression then matches expression `Cbb` with a variable match of `bb` which is one of the possible variable definitions. Thus, this combination of instruction and constraint results in a match, and a return value of `TRUE`.

These examples illustrate that the constraint checker must keep track of all possible definitions of variables and check the subsequent variable references against all possible definitions in order to determine whether or not a match exists.

## 7.2 Using the Constraint Checker

There are four different times in the process of code generation that require checking the validity of instructions. They are:

1. Testing all operation permutations in order to eliminate non-terminal options that lead to invalid operations.
2. Testing maximal cliques before instruction selection and register allocation.
3. Testing selected instructions with register allocation already performed.
4. Testing for time-shifted constraints.

### 7.2.1 Removing Invalid Non-Terminal Options

The first situation that requires the constraint checker occurs during the parsing of the ISDL description and creation of the databases. In ISDL, operations may include non-terminals that simplify the description of the operation by grouping multiple entities into a single non-terminal. Using non-terminals avoids listing each possible combination of non-terminal options as an individual operation. However, it is possible that not all of the options of a non-terminal lead to a valid instruction. All illegal cases are described using constraints.

Consider the following example. In the databus move operations of the example architecture of Figure 3-3, the operations' descriptions specify that the databus move operations can transfer a value from `SRC` to `DEST`, as shown below.

| | |
|---|---|
| *operation syntax:* | `DB1_move SRC, DEST` |
| *operation RTL:* | `DEST` $\leftarrow$ `SRC` |
| *operation syntax:* | `DB2_move SRC, DEST` |
| *operation RTL:* | `DEST` $\leftarrow$ `SRC` |

`SRC` and `DEST` are non-terminals that represent the registers of register files `U1`, `U2`, or `U3`. If there was no constraint on these operations, then the following forms of these operations would be valid:

| DB1 field | DB2 field |
|---|---|
| DB1_move U1_R, U1_R | DB2_move U1_R, U1_R |
| DB1_move U1_R, U2_R | DB2_move U1_R, U2_R |
| DB1_move U1_R, U3_R | DB2_move U1_R, U3_R |
| DB1_move U2_R, U1_R | DB2_move U2_R, U1_R |
| DB1_move U2_R, U2_R | DB2_move U2_R, U2_R |
| DB1_move U2_R, U3_R | DB2_move U2_R, U3_R |
| DB1_move U3_R, U1_R | DB2_move U3_R, U1_R |
| DB1_move U3_R, U2_R | DB2_move U3_R, U2_R |
| DB1_move U3_R, U3_R | DB2_move U3_R, U3_R |

However, the constraint:

```
~(DB*_move U@[1].R*, U@[1].R*)
```

specifies that not all combinations of the SRC and DEST non-terminals are valid. Specifically, it states that SRC and DEST cannot represent the same register file within a databus move operation. This constraint removes some of the combinations listed above, and the resulting valid databus move operations are presented below.

| DB1 field | DB2 field |
|---|---|
| DB1_move U1_R, U2_R | DB2_move U1_R, U2_R |
| DB1_move U1_R, U3_R | DB2_move U1_R, U3_R |
| DB1_move U2_R, U1_R | DB2_move U2_R, U1_R |
| DB1_move U2_R, U3_R | DB2_move U2_R, U3_R |
| DB1_move U3_R, U1_R | DB2_move U3_R, U1_R |
| DB1_move U3_R, U2_R | DB2_move U3_R, U2_R |

In order to guarantee that only valid operation formats are considered, all combinations within each individual operation (i.e., before the operations are merged into

instructions) must first be tested against the constraints. Any combination that does not satisfy the constraints should not be considered for inclusion in the Split-Node DAG.

### 7.2.2 Testing Maximal Cliques Prior to Instruction Selection

The constraint checker is also required for checking the validity of the generated maximal cliques. Each maximal clique must be checked before it is considered for selection in covering the Split-Node DAG. This is required because the maximal clique generator only accounts for the available parallelism on the target processor in the form of independent functional units. It does not take the constraints into consideration. Thus, after generating the maximal cliques, each clique must be converted into its assembly representation and passed through the constraint checker. Any clique that does not satisfy all of the constraints is recursively split into smaller cliques until the resulting cliques satisfy all of the constraints.

For example, in the example architecture there exist separate fields for the databus operations and the memory operations. Therefore, a maximal clique may merge two databus move operations with a data memory load operation as shown in the following operation grouping:

```
{ DB1_move U1.R0, U2.R1; DB2_move U2.R2, U3.R3;
  DM_ld U3.R4, 0x1; }
```

Note that the register assignments in the operation groupings are meaningless when testing clique validity prior to instruction selection. They are simply being used as unique placeholders to ensure that any constraints on the register allocation will not be trigerred during this phase of constraint checking.

Although this operation grouping can be generated as a maximal clique, the following constraint specifies that a databus move operation cannot be executed in parallel with any memory operation.

```
~( ((DB*_move* *,*) | (DB*_nop)) & ((DM_* *) | (IM_* *)) )
```

The reason is that the memory operations require the use of both databuses; therefore, they cannot be executed in parallel with any operation that would also require the use of either databus.

When testing the operation grouping mentioned above against this constraint, a FALSE value is returned. Therefore, the operation grouping must be split into smaller groupings until all of the constraints are satisfied. Splitting the operation grouping into the two groups shown below satisfies all of the constraints in the example architecture.

```
{ DB1_move U1.R0, U2.R1; DB2_move U2.R2, U3.R3; }
{ DM_ld U3.R4, 0x1; }
```

### 7.2.3 Testing the Final Selected Instructions After Register Allocation

After instruction selection and resource allocation have been performed, the selected instructions must be passed through the constraints checker to test that the register allocation satisfies all of the constraints. If the constraints are not satisfied, then the register allocation must be redone. Note that by this phase of constraint checking, the operation grouping into instructions has already been tested against the constraints. Therefore, only the register allocation remains to be tested.

Suppose that the following instruction was included in the Split-Node DAG cover:

```
{ ...  DB1_move U2.R0, U1.R1; DB2_move U3.R1, U1.R1; ...  }
```

Upon comparison of this instruction with the last constraint of the example architecture, namely,

```
~( (DB1_move* *,@[1]) & (DB2_move* *,@[1]) )
```

it would be determined that the assigned register allocation was invalid because both databus move operations wrote into the same register. Thus, one of the destination

operands would have to be reallocated, and any subsequent use of the corresponding value would have to be renamed.

Although this constraint is explicitly stated in the ISDL description, AVIV should never allocate registers in a manner that would not satisfy this constraint. AVIV would identify the two destination operands as unique variables and allocate a unique register for each. This implies that this constraint should always be satisfied. Nevertheless, checking for such constraints is still a useful form of verification.

### 7.2.4 Checking for Time-Shifted Constraints

The constraint checker is also used to test for time-shifted constraints. Such constraints require the entire schedule of selected instructions to be examined rather than a single instruction at a time. All pairs of instructions, that have been scheduled $n$ instructions away from each other, are tested against a time-shifted constraint that has an $n$ instruction shift constraint in it. If any pair of instructions does not satisfy the constraint, then the instructions either need to be rescheduled, or `NOP` operations must be inserted between the instructions, so that all of the constraints are satisfied.

A constraint from the Motorola 56000 commercial DSP processor is used to illustrate this form of constraint checking because the example architecture of Figure 3-3 does not contain any time-shifted constraints. In the Motorola 56000 processor, an instruction containing a `Main_DO` operation cannot immediately follow an instruction containing a `Main_REP` operation. This constraint is expressed in ISDL as follows:

```
~( (Main_REP *) & [1](Main_DO *) ).
```

This constraint is a 1 instruction time-shifted constraint. As a result, every pair of instructions that are 1 instruction away from each other must be tested. If the first instruction tested contains a `Main_REP` operation, and the instruction immediately following it contains a `Main_DO` operation, then the constraint checker will return `FALSE`, and the instruction schedule must be altered. In order to satisfy the constraint, the code generator can either insert `NOP` operations, or move the `Main_DO` operation

to a later instruction provided that all precedence constraints are still satisfied. This process is repeated for every pair of instructions until all pairs satisfy the constraint.

## 7.3 The Constraint Checker

There are two main techniques that can be used to implement the constraint checker. The first technique is a general-purpose expression matcher that supports combining expressions using the '~', '&', and '|' logical operators. The second technique uses information from the ISDL description to simplify the matching algorithm. It, too, supports combining expressions through logical operators. Pseudo-code for each technique will be presented as well as timing measurements for the two techniques. It will be shown that the second technique is significantly faster and, therefore, the preferable algorithm for the constraint checker. The time required to check the potential instructions against the constraints is significant because checking for constraints is an NP-complete problem that must be solved exactly. Hence, it is important to explore various algorithms for constraint checking in order to identify a solution that achieves feasible runtime.

Both techniques use the regular expression matching functions, `regcomp` and `regexec` [27], to find the matches within a constraint basic expression. The main difference in the techniques lies in how they handle matches across the logical operators.

In order to use the regular expression functions, the constraint basic expressions specified in the *Constraints* section of an ISDL description must first be converted into regular expressions. The original syntax of the constraints is very similar to that of regular expressions except for the wild card and variable reference syntaxes. In order to convert the constraint basic expressions into regular expressions, the following conversions must be made. Each wild card is converted into its regular expression equivalent (e.g., * is converted to `[^[:space:];]*`). A variable definition is converted to `([^[:space:];]*)` where the parentheses specify that the regular expression represents a variable. Each variable reference is converted into the equivalent regular

expression notation of variable references (e.g., an `@[1]` reference is converted to `\1`). The logical operators are not altered.

The `regcomp` regular expression function takes a regular expression as input and compiles it into a pattern buffer that is used by the `regexec` function. The `regexec` function matches the pattern buffer to an input string. It returns a value specifying whether or not the string matched the pattern buffer. In addition, it stores the result of any variable matches for future examination.

The `regcomp` and `regexec` routines can only be used to find matches within a constraint basic expression. They do not handle matching across the logical operators. The two techniques described below handle the logical operators and determine whether the variable matches are consistent across the logical operators.

### 7.3.1 Technique I

Figure 7-1 presents the pseudo-code for technique I of the constraint checker. It defines two main routines, `match` and `redo_match`. The `match` routine is first called to try and find a match between each expression in the constraint and a field in the instruction. When a match is found, any variable definitions occurring within that expression are stored so that they can be used when matching the rest of the constraint.

As expected, the result of a match on an expression preceded by a negation operator is the negation of the result of a match on the expression. Furthermore, a match on an *or* expression returns a `TRUE` value if either the left or right subexpressions match the input instruction. Finally, a match on an *and* expression only returns `TRUE` if both the left and right subexpressions match the input instruction.

```
check_constraints( instr ) {
    for ( i = 0; i < number_of_constraints; i++; ) {
        if ( match( instr, constr[i] ) == FALSE ) return FALSE;
    }
    return TRUE;
}

match( instr, constr_expression ) {
    switch (constr_type)
        case EXPR: {
            replace all variable references with variable definition if already defined
            for ( j = 0; j < instr_num_fields; j++; ) {
                try to match expression against field j of instr
                if a match is found
                    set a flag for field j
                    store any newly defined variable matches
                    break;
            }
            if a match was found return TRUE;
            else return FALSE;
        }
        case NOT: {
            // return opposite of return value for expression following negation character
            return ( ! match( instr, expression following negation character ));
        }
        case OR: {
            // return TRUE if either left or right expressions match
            if ( match( instr, left expression of or ))
                return TRUE;
            if ( match( instr, right expression of or ))
                return TRUE;
            return FALSE;
        }
        case AND: {
            // return TRUE if both left and right expressions match
            if ( match(instr, left expression of and ))
                while TRUE {
                    if ( match( instr, right expression of and ))
                        return TRUE;
                    else
                        if ( redo_match( instr, left expression of and ))
                            continue while loop // now try right expression again
                        else return FALSE;
                }
        }
}
```

Figure 7-1: **Pseudo-code for technique I of constraints checker**

Figure 7-1 continued.

```
redo_match( instr, constr_expression ) {
    switch (constr_type) {
        case EXPR: {
            undo previous matches of current expression before trying again
            replace all variable references with variable definition if already defined
            for ( j = 0; j < instr_num_fields; j++; ) {
                // try to find a new match for the expression,
                // skip current field if it was already used to obtain a match
                if flag for field j is set
                    continue;
                try to match expression against field j of instr
                if a match is found
                    set a flag for field j
                    store any newly defined variable matches
                    break;
            }
            if a match was found return TRUE;
            else return FALSE;
        }
        case NOT: {
            undo previous matches of current expression before trying again
            // return opposite of return value for expression following negation character
            return ( ! redo_match( instr, expression following negation character ));
        }
        case OR: {
            undo previous matches of left or right expressions before trying again
            // return TRUE if either left or right expressions match
            if ( redo_match( instr, left expression of or ))
                return TRUE;
            if ( redo_match( instr, right expression of or ))
                return TRUE;
            return FALSE;
        }
        case AND: {
            undo previous matches of right expression before trying again
            // first attempt to redo right expression without undoing the left expression match
            if ( redo_match( instr, right expression of and ))
                return TRUE;
            else
                undo previous matches of left expression before trying again
                // start all over with left expression (fields that were already matched will be skipped)
                if ( redo_match( instr, left expression of and ))
                    if ( match( instr, right expression of and ))
                        return TRUE
                else return FALSE;
        }
    }
}
```

Figure 7-1: **Pseudo-code for technique I of constraints checker**

The algorithm is complicated by the fact that just because an expression match is found, using a particular instruction field, does not mean that that was the correct match. It is possible that the identified match may preclude the rest of the constraint's basic expressions from matching, whereas another field match may not. However, this cannot be determined until a `FALSE` result is returned from the original match selections. In such a case, the previous matches need to be undone and any alternative matches must be tried as well. This undoing and redoing of matches affects the runtime of the constraint checking procedure significantly.

As an example of this technique, consider the constraint `(( A@[1] | B@[1] ) & C@[1])` compared to the instruction `{ Aaa; Bbb; Cbb; }`. In this technique, the match routine would first be called with an *and* expression type where the left expression is `( A@[1] | B@[1] )` and the right expression is `( C@[1] )`. The left expression would then be searched for a match. Since the left expression type is an *or* expression, it only needs to match either the left or the right expressions. It first tries the left expression `A@[1]`. This expression is a basic expression and is thus compared to the fields of the instruction. It finds a match on the first field `Aaa` and sets variable 1 to be `aa`. The right expression of the *and* is now checked for a match. However, when the variable reference is replaced by the value `aa`, the right expression becomes `Caa` for which no match is found in the instruction. This does not mean that the return value is `FALSE`, rather it means that the *and* match must be redone.

The `redo_match` routine is now called on expression `( A@[1] | B@[1] )`. It is aware of the fact that expression `A@[1]` was already matched to the first field of the instruction. In the pseudo-code this translates to knowing that the field 1 flag of expression `A@[1]` is set. Redoing an *or* expression match implies redoing the left expression. If a match is not found for the left expression, then the right expression is redone. The routine first tries to redo the left expression `A@[1]` with the field 1 flag set. Because the field 1 flag is set, it does not try to match the first field of the instruction, and when checking the second and third fields, no match is found. Thus, the right expression of the *or*, `B@[1]`, is checked. Now a match is found on field 2 and variable 1 is reset to `bb`. The `match` routine then tries to match `C@[1]` again. Now

when the variable reference is replaced with `bb`, the resulting expression is `Cbb` which matches field 3. Hence, the final return value of the constraint checker is `TRUE`.

This technique results in a general purpose constraint checker. It does not make use of ISDL information to simplify its search space. Thus, it places no restrictions on the order of the operations within an instruction. This allows a partial instruction to be passed as input rather than an entire instruction containing an operation from each field. However, this implies that every constraint basic expression must be compared to all fields of an input instruction in order to see whether or not a match is found.

### 7.3.2 Technique II

The second technique uses information from the ISDL description to reduce the search space of matches in the constraint checker. This is accomplished by making the following two assumptions. First, each constraint basic expression represents exactly one instruction field. As a result, when searching for a match, the constraint basic expressions only need to be compared to the corresponding field in the input instruction rather than being compared to every field. Second, the operations within an instruction are ordered according to the ordering of the instruction fields defined in the ISDL description. This is permissible because any instruction that will be input to the constraints checker will be coming from the code generator whose output follows this format.

In order to ensure that the first assumption is maintained, the constraints specified in the ISDL description may need to be slightly altered. For example, the constraint:

```
~(DB*_move U@[1].R*, U@[1].R*)
```

is split into two constraints that distinguish between the `DB1` and `DB2` operations. The resulting constraints are:

```
~(DB1_move U@[1].R*, U@[1].R*)
~(DB2_move U@[1].R*, U@[1].R*)
```

Although these types of modifications may be necessary in order to use technique II, this process only needs to be performed once for each ISDL description. The time required to determine the required modifications is linear in terms of the number operations in the ISDL description.

Another distinction between the two techniques is that in technique I, when a match is found on an expression that contains a *definition* of a variable, the variable definition is stored. Subsequently, every time a variable *reference* is encountered, the variable is replaced by the stored definition, and the resulting expression is matched against the instruction fields. In technique II, a two-pass scheme is used where, initially, all variable definitions and references are treated as variable definitions. Thus, any expression containing a variable stores the resulting variable match. Then, in a second pass, all the variable matches are compared across the logical operators to determine whether or not a complete match exists.

The pseudo-code for technique II is shown in Figure 7-2. As an example of this technique, consider again the constraint `(( A@[1] | B@[1] ) & C@[1])` compared to the instruction `{ Aaa; Bbb; Cbb; }`. In technique II, the first pass (i.e., the `match` routine) first compares each basic expression to its corresponding field. In this example, the constraint basic expression `A@[1]` corresponds to the instruction field `Aaa`, expression `B@[1]` corresponds to field `Bbb`, and expression `C@[1]` corresponds to field `Cbb`. In other words, there is a clear one-to-one correspondence between the constraint basic expressions and the instruction fields. Thus, `A@[1]` is compared to `Aaa`. It finds a match and sets its copy of variable 1 to `aa`. Expression `B@[1]` is compared to `Bbb`. It, too, finds a match and sets its copy of variable 1 to `bb`. Finally, expression `C@[1]` is checked against field `Cbb`. It finds a match and sets its copy of variable 1 to `bb`. The result of the `match` routine is the value `TRUE` (i.e., (`TRUE` | `TRUE`) & `TRUE`) = `TRUE` ). The variable matches are now compared because the match routine returned a `TRUE` value. The `var_match` routine needs to determine whether the variable matches of `((aa | bb) & bb )` are acceptable. It determines that variable 1 could have been defined as either `aa` or `bb`, and therefore the reference of `bb` is acceptable, and the final return value is `TRUE`. This is accomplished by maintaining a

```
check_constraints( instr ) {
    for ( i = 0; i < number_of_constraints; i++; ) {
        if ( match_constraint( instr, constr[i] ) == FALSE ) return FALSE;
    }
    return TRUE;
}

match_constraint( instr, constr_expression ) {
    if ( match( instr, constr_expression ))
        if ( var_match ( instr, constr_expression, matches ))
            return TRUE;
    else return FALSE;
}

// This routine stores all variable matches (as if they were definitions)
match ( instr, constr_expression ) {
    switch (constr_type) {
        case EXPR: {
            try to match expression against corresponding field of instr
            if a match is found
                store the variable matches of this expression
                return TRUE;
            else
                return FALSE;
        }
        case NOT: {
            try to match expression following negation character
            always return TRUE;
        }
        case OR: {
            left_value = match ( instr, left expression of or )
            right_value = match ( instr, right expression of or )
            return ( left_value || right_value );
        }
        case AND: {
            left_value = match ( instr, left expression of and )
            right_value = match ( instr, right expression of and )
            return ( left_value && right_value );
        }
    }
}
```

Figure 7-2: **Pseudo-code for technique II of constraints checker**

Figure 7-2 continued.

```
// If match returned TRUE then check if variables match
var_match ( instr, constr_expression, *matches ) {
    switch (constr_type) {
        case EXPR: {
            if match returned FALSE
                return FALSE;
            else
                set up a two dimensional array of matches: one dimension for each possible match;
                the other dimension for each variable within the match
                store the variable matches for current expression as first possible match in array
                return TRUE;
        }
        case NOT: {
            // return opposite of return value for expression following negation character
            return (! var_match( instr, expression following negation character, matches ));
        }
        case OR: {
            if match returned FALSE
                return FALSE;
            else
                left_value = var_match ( instr, left expression of or, left_matches )
                right_value = var_match ( instr, right expression of or, right_matches )
                return_value = left_value || right_value;
                if ( return_value == FALSE ) return FALSE;
                else
                    find union of left and right variable matches
                    store resulting matches in two dimensional array of matches
                    corresponding to current expression.
                    if each existing variable has at least one match return TRUE;
                    else return FALSE;
        }
        case AND: {
            if match returned FALSE
                return FALSE;
            else
                left_value = var_match ( instr, left expression of and, left_matches )
                right_value = var_match ( instr, right expression of and, right_matches )
                return_value = left_value && right_value;
                if ( return_value == FALSE ) return FALSE;
                else
                    find intersection of left and right variable matches
                    store resulting matches in two dimensional array of matches
                    corresponding to current expression.
                    if each existing variable has at least one match return TRUE;
                    else return FALSE;
        }
    }
}
```

Figure 7-2: **Pseudo-code for technique II of constraints checker**

list of possible matches for each subexpression. The *intersection* of the matches for the left and right subexpressions determines the possible matches for *and* expressions. The *union* of the subexpression matches determines the possible matches for *or* expressions. The process of finding the intersection and union of subexpression matches continues until a set of matches corresponding to the complete expression is derived. If the resulting set of matches is non-empty for each variable index present in the constraint, then a valid match exists and the return value is `TRUE`. Otherwise, the return value is `FALSE`.

In order to find the intersection of a pair of matches, all of the common variables (i.e., variables that appear in both the left and right subexpressions) must first be identified. The variable matches of the common variables must be identical in the left and right subexpressions. If this is not the case, then the intersection of the two matches is empty. If this is the case, then the common variables are defined as they were in the left and right subexpressions. Furthermore, all variables, that are unique to the left or right subexpression, are defined as they were in their respective expression.

The union of a pair of matches results in two matches if the matches are different and both the left and right subexpressions returned a `TRUE` value (i.e., found a possible match). If only one of the subexpressions returned a `TRUE` value, then the union of the matches consists of exactly one match which is the match of the expression that resulted in a `TRUE` value.

In this constraint checking technique, it is possible that redundant expressions will be checked in the first pass (e.g., if expression `A@[1]` matched and its variable match corresponded to expression `C@[1]`'s match, then checking `B@[1]` for a match was unnecessary). However, because the variable matching is not performed until the second pass, this redundancy is required. Nevertheless, the fact that it is not necessary to redo matches recursively, as is required in technique I, and the fact that each expression is only compared with a single field in the instruction, reduce the CPU time required to determine whether or not an instruction satisfies the defined constraint.

Note that, within the match routine of technique II, the NOT constraint expression type is treated differently from the EXPR, OR, and AND expression types. In the EXPR, OR, and AND constraint expression types, the sequence of events is that a match is first attempted on the expressions without worrying about variable consistency across the logical operators. In other words, if a match exists given any variable definition, then a TRUE value should be returned. Subsequently, the variable consistency is checked in order to determine whether the final return value should also be TRUE. For these three types of constraint expressions, if the match routine could not find a match regardless of the variable definition, then it is known that the return value is FALSE, and the var_match routine does not need to be executed. However, in the case of a NOT constraint expression, the model is slightly different. In such a case, the match routine always returns a TRUE value. The explanation for this follows from the two cases described below:

- Case 1: In the match routine, the expression following the NOT returns FALSE because regardless of the variable definitions, a match could not be found. In such a case, the NOT expression should return TRUE, and then executing the var_match routine will simply return the same value.

- Case 2: In the match routine, the expression following the NOT returns TRUE because there exist some variable definitions for which a match could be found. In such a case, it would be incorrect to return FALSE for the NOT expression because this would result in the var_match routine never being executed. However, it is possible that if the var_match routine was executed, then it would be determined that variable consistency across the logical operators is impossible. As a result, the return value for the expression following the NOT should actually be changed to FALSE, which would then make the return value of the NOT expression be TRUE. Therefore, the FALSE return value determined by the match routine would have been incorrect.

For these reasons, the match routine always returns TRUE in the case of a NOT constraint expression, and the var_match routine must be executed to determine the

| Example | Technique I (secs) | Technique II (secs) |
|---|---|---|
| Ex1 | 0.53 | 0.28 |
| Ex2 | 10.05 | 5.12 |
| Ex3 | 27.69 | 13.72 |
| Ex4 | 122.66 | 59.70 |
| Ex5 | 144.09 | 70.14 |

Table 7.2: **Performance comparison of techniques I and II of the constraint checker**

actual return value.

## 7.4 Performance Comparison of Techniques I and II

Table 7.2 compares the performance of the two constraint checking techniques. The examples are based on several sample code segments compiled for the example architecture given in Figure 3-3. The times shown in the table are the total amount of CPU time required to compare all of the maximal cliques to the constraints. It is clear from these results that technique II is approximately a factor of two faster than technique I.

# Chapter 8

# The Assembler Generator

In order to decouple the development of the compiler from the rest of the ARIES system, it is preferable for the compiler to produce assembly code rather than binary code. An assembly program is easier for a developer to read and debug than a binary program. Therefore, producing the result of code generation in assembly allows the compiler to undergo initial testing without a simulation environment. However, for the entire ARIES system to function together, the input to the instruction-level simulator should be binary code. In order to automate the process of converting the assembly code to binary code, an assembler generator is needed. The availability of an assembler also facilitates testing of the simulation environment on its own. With an assembler, the developer can write assembly test programs and automatically convert them to the binary code format required by the simulator.

The assembler generator receives an ISDL description as input and automatically generates an assembler for the described target processor. The generated assembler then assembles the compiler's output into a binary file that can be used by the simulator and can be executed on the ASIP processor.

## 8.1 Lex and Yacc

The assembler generator consists of Lex and Yacc files which parse the ISDL description in order to extract all of the information required to create an assembler for the

target processor.

Lex and Yacc [35] are tools that assist in writing programs that transform structured input such as an ISDL description. There are two main tasks that need to be performed. The first is to divide the input into meaningful units, and the second is to discover the relationship between the units. *Lexical analysis* is the process of dividing the input into units, also known as *tokens*. The Lex tool takes a description of the possible tokens, such as keywords in the ISDL description, and produces a C routine, a *lexical analyzer*, that can identify those tokens. The tokens are described to the lexer using regular expressions [27].

A *parser* is a program that can determine the relationship between the tokens. It consists of a list of rules that define the relationships between the tokens. These rules are known as a *grammar*. Yacc takes a concise description of a grammar and produces a C routine, a *parser*, that can parse that grammar. The yacc parser detects whenever a sequence of input tokens matches one of the rules in the grammar. It also detects syntax errors in the input whenever the input does not match any of the parser's rules.

The yacc grammar consists of a set of *rules*. Each rule begins with a non-terminal name and a colon followed by a possibly empty list of symbols, tokens, and actions. The end of a rule is indicated by a ';'. For example, the following rule:

```
time:  hour ':'  minutes ;
```

specifies that the non-terminal *time* consists of an *hour* value, followed by a ':' symbol, followed by a *minutes* value. In this example, hour and minutes are tokens that have a value associated with them.

If a non-terminal can have several different options, then all of the options after the first begin with a vertical bar, '|', rather than the non-terminal name and colon. Thus, if time could be specified with or without seconds, the definition of time would be the following:

```
time:   hour ':' minutes
      | hour ':' minutes ':' seconds
      ;
```

The *action* of a rule is a set of C statements that are executed whenever the parser reaches the point in the grammar where the action occurs. The values of the tokens in the rule can be accessed by the action statement by using the '$n' notation where n is the index of the token or symbol within the rule. For example, the hour value can be accessed using the '$1' notation, and the minutes value can be accessed using the '$3' notation. In the example above, each option of the time non-terminal may have an action associated with it. For example, the time rule may be the following:

```
time:   hour ':' minutes
           { printf(''The time is %d:%d\n'', $1, $3); }
      | hour ':' minutes ':' seconds
           { printf(''The time is %d:%d:%d\n'', $1, $3, $5); }
      ;
```

where the action depends on the option that is matched.

## 8.2 ISDL Information Required for the Creation of an Assembler

The Lex and Yacc files that make up the assembler generator must extract the following information from the ISDL description:

- **The Instruction Format** - The instruction format specifies the exact order in which the instruction word subfields appear in the binary instruction word.
- **Tokens** - The *assembly syntax* of the tokens specifies the format in which the tokens appear in the assembly instruction word. The token *value* is used to determine the bitfield assignments of operations that refer to the token. The *name* of the token differentiates it from other tokens.

- **Non-Terminals** - The *return value* of non-terminals is used to determine the bitfield assignments of operations that use the non-terminal. The non-terminal *name* is used to differentiate it from other non-terminals.

- **Split Functions** - Split functions define how a long bitfield (such as a memory address) can be split up into existing subfields of the binary instruction word.

- **Operations** - The *assembly syntax* of the operations specifies the assembly representation of the operations. It is used to identify which operations are present in the assembly instruction. The operation *bitfield assignments* specify the value assigned to the bitfields associated with each particular operation.

The assembler generator automatically produces a new set of Lex and Yacc files which, when compiled, result in an executable capable of parsing the assembly input and generating the binary instruction words. In other words, the generated Lex and Yacc files are an *assembler* for the target processor.

## 8.3 The Generated Lex File

The generated Lex file contains the lexical analyzer that can split any valid assembly program into tokens. In addition, it creates a symbol table that stores any labels encountered in the input assembly program. The Lex file defines the following tokens:

- **Predefined types** - Predefined types include integers, hexadecimals, real numbers, and strings.

- **Operation names** - Operation names refer to all of the operation names specified in the *Instruction Set* section of the ISDL description.

- **ISDL defined tokens** - These tokens are the tokens that are defined in the *Global Definitions* section of the ISDL description.

- **Symbolic address labels** - These tokens refer to labels used in the symbol table to represent symbolic addresses.

The lexer defined tokens may or may not have values associated with them. The operation names, for example, do not have any value associated with them. They are just used to differentiate among all of the target processor operations. The predefined types, on the other hand, do require an associated value because they not only need to represent the fact that such a token appeared in the input program, but they also need to keep track of the value of that token. For example, an integer token represents the fact that an integer was detected in the input program, and also records the value of that integer for later use.

The tokens representing the ISDL defined tokens also store the value that the token is representing. For example, the following ISDL token definition:

```
Token "U1.R"[0..3] U1_R { [0..3]; };
```

specifies that token `U1_R` represents the assembly syntax `U1.R` followed by an integer between 0 and 3. The value associated with that token is an integer between 0 and 3 corresponding to the value in the assembly input. Thus, if the assembly input is `U1.R2`, then the returned token would be `U1_R` representing the `U1` register file. The value of this token would be 2 representing the second register in the register file.

Finally, the labels have a symbol table entry associated with them. This entry contains the name of the label along with the instruction address that it represents.

## 8.4 The Generated Yacc File

The generated Yacc file contains the main driver and the actual parser of the assembler. This parser is a two-pass parser capable of processing symbolic addresses (i.e., labels). The main driver performs initializations for file I/O, as well as some cleanup in the case of a syntax error. The parser contains a top-level rule that performs the two passes.

The first pass fills in the symbol table and produces a listing file as a debugging aid. This pass enables forward and reverse references in the labels. The second pass produces and outputs the binary instruction words. Two passes are required because

in order to embed the label addresses into the binary instruction words, the parser must first determine the address value of the labels. If all label references were reverse references then a second pass would not be required because the address values would be determined as the instructions were parsed. However, a one-pass parser cannot support forward references because when parsing the forward reference the address value of the label is still unknown.

In order to determine the address corresponding to a given label, it is insufficient to simply count the number of assembly instructions preceding the label. The reason is that a single assembly instruction may require more than one binary instruction word. This can occur if the instruction contains a large constant (e.g., the target address of a jump operation). In such cases, the assembly instruction is mapped into multiple binary instruction words. Therefore, in the first pass of the parser, each instruction must actually be parsed in order to determine the total number of binary instruction words required to represent it. The bitfield assignment information embedded in each operation and non-terminal description specifies when and how many *additional* binary words are required. The address counter is incremented by one or more for each assembly instruction depending on the required number of additional instruction words.

The two passes of the parser use the same parsing rules, but the actions performed by each pass are different. In the first pass, the address of each assembly instruction is determined and stored. In the second pass, the instruction words are created and output.

### Components of the Parser

The binary instruction word consists of multiple fields each of which may consist of multiple subfields. These fields are not necessarily correlated with the fields defined in the *Instruction Set* section of the ISDL description. The instruction word field and subfield division is specified in the Format section of the ISDL description. Each subfield consists of a subfield name along with its width in bits. The ordering of the fields and subfields specifies the order in which the subfields should be concatenated

to form the instruction word. This information is used to generate a routine for the parser which will concatenate all of the subfields into an instruction word. Also, each of the subfields is declared as a variable in the parser so that it may be assigned a value individually.

In order to clarify the components of the parser, the example architecture of Figure 3-3 is used. Its ISDL description is provided in Appendix A.

The Format section of the example architecture specifies the following instruction format:

```
Section Format

U1 = OP[2], RA[2], RB[2], RC[2];
U2 = OP[2], RA[2], RB[2], RC[2];
U3 = OP[2], RA[2], RB[2], RC[2];
DB1 = SRC[5], DEST[5];
DB2 = SRC[5], DEST[5];
```

The first line of the format description specifies that the instruction word begins with field `U1` which consists of four subfields `OP`, `RA`, `RB`, and `RC`. Four unique variables in the assembler represent these subfields. They are `U1_OP`, `U1_RA`, `U1_RB`, and `U1_RC`. Similar variables exist for each field. These variables are assigned a value during the parsing of the assembly instruction. The assigned bitfields are then concatenated to form the instruction word.

The parser rules consist of a top-level rule that composes an instruction out of the *fields* described in the Instruction Set section of the ISDL description. Each *field* is a subrule that consists of a list of possible patterns, one for each *operation* described in the specification for that field. The action for each pattern sets the appropriate bitfields in the instruction word according to the particular operation that the field is representing.

For the example architecture, the top-level rule that composes the instruction is the following:

```
word:  '{' U1f U2f U3f DB1 DB2 DMf IM '}'
          { ... } ;
```

The fields U1f, U2f, ..., IM each have their own rule that lists all of the possible operations for that field. The actions corresponding to each operation set the appropriate bitfields in the instruction word. For example, the U1f field rule is:

```
U1f:    U1_NULL ';'
           { }
      | U1_add U1_RA ',' U1_RB ',' U1_RC ';'
           { U1_OP = 0x0 ;
             U1_RA = $2 ;
             U1_RB = $4 ;
             U1_RC = $6 ;
           }
      | U1_sub U1_RA ',' U1_RB ',' U1_RC ';'
           { U1_OP = 0x1 ;
             U1_RA = $2 ;
             U1_RB = $4 ;
             U1_RC = $6 ;
           }
      | U1_nop ';'
           { U1_OP = 0x3 ;
           }
      ;
```

In the case where the assembly instruction includes a U1_sub operation, the U1_sub U1_RA ',' U1_RB ',' U1_RC ';' portion of the U1f rule is followed. It specifies that a U1_sub operations assembly representation consists of the operation name followed by three parameters separated by commas, and ends with a semicolon. The parameters are represented using the ISDL defined non-terminals U1_RA, U1_RB,

and U1_RC. Its action sets the opcode bitfield, U1_OP, to the hexadecimal value 1, the U1_RA bitfield to the value of non-terminal U1_RA, the U1_RB bitfield to the value of non-terminal U1_RB, and so on. Note that some rules, such as the U1_nop rule, do not assign a value to each of the bitfields associated with the field. The bitfields are initialized to zero. Therefore, if no assignment is made, the value is assumed to remain zero unless another operation assigns a value to that bitfield.

In addition to having a rule for each *field*, each *non-terminal* defined in the ISDL description has its own rule as well. The non-terminal rules specify the options that the non-terminal can represent. Each non-terminal option has a value associated with it. In order to obtain the value of the non-terminals U1_RA, U1_RB, U1_RC, etc., the rule for the corresponding non-terminal is followed. For example, non-terminal U1_RA, has a rule consisting of a single option:

```
U1_RA:  U1_R
            { $$ = $1 ; }
        ;
```

It specifies that non-terminal U1_RA can represent the token U1_R, and the value associated with the non-terminal is the value of the token (this is specified by the {$$ = $1;} statement).

Non-terminal SRC can represent multiple tokens, namely U1_R, U2_R, and U3_R. Its rule is shown below:

```
SRC:    U1_R
            { $$ = 0 | $1 ; }
      | U2_R
            { $$ = 4 | $1 ; }
      | U3_R
            { $$ = 8 | $1 ; }
      ;
```

In this case, the value of the non-terminal is not necessarily the value of the represented token. Instead, the action for each of the non-terminal options sets the

non-terminal value to the result of performing a bitwise *or* between a specified constant value and the value of the token. Since the value of tokens `U1_R`, `U2_R`, and `U3_R` lies between 0 and 3, the actions' effect is to add 0, 4, or 8 respectively to the token value. This is necessary because the value of the tokens is simply the register number within the register file. However, in order to distinguish between registers of different register files, additional encoding is required.

If split functions are defined for the target processor, then the split function definitions are used to create routines that take a long constant and split it across the multiple bitfields corresponding to that split function. The parsing rules for operations and non-terminals may include function calls to the split function routines when necessary.

Once all of the subrules of the *word* top-level rule have been followed, each of the bitfields of the binary instruction word has been assigned its appropriate value. The remaining task is to concatenate all of the bitfields into a single binary word according to the order and bitwidths specified in the Format section of the ISDL description. If *additional* words are required, the bitfield assignments for each additional word would have been set as well. These bitfields are also concatenated according to the same bitfield ordering in order to form any additional words required.

### Macros and File Inclusion

In order to provide support for macro substitutions and file include mechanisms, before parsing the input assembly programs, the assembler first preprocesses the input files using the C preprocessor. Macro substitutions can simplify the writing of assembly programs. This is particularly helpful when the assembly programs are hand written. Supporting file include mechanisms enables the inclusion of a common kernel that acts as an operating system for the compiled code. The kernel performs various initialization tasks such as setting up trap and exception vectors and enabling interrupts.

# Chapter 9

# Results

ISDL has been used to describe a number of different architectures varying from the example VLIW processor described in Chapter 3 to a seven-way VLIW with non-homogeneous data-paths. The following five architectures were used to obtain the results presented in this chapter. Figure 9-1 provides a pictorial view of these architectures.

- **ARIES1:** ARIES1 is a five-way VLIW with three non-homogeneous functional units and two databuses. This architecture was previously illustrated in Figure 3-3. Functional unit `U1` can perform addition and subtraction. Unit `U2` can perform addition, multiplication, and subtraction. Unit `U3` can perform addition and multiplication. Each functional unit has a dedicated register file with four registers per register file. This processor is a Harvard architecture (i.e., contains separate instruction and data memories). In this processor, all data-paths are 8 bits wide and the instruction width is 44 bits. The complete ISDL description for this architecture is provided in Appendix A.

- **ARIES2:** ARIES2 is a modified version of ARIES1 with functional unit `U3` removed, and the subtraction operation removed from unit `U1`. All data-paths are 8 bits wide and the instruction width is 35 bits.

- **ARIES3:** ARIES3 is a slightly modified version of ARIES1. In this architecture, each register file contains two registers rather than four. This results in an

instruction width of 35 bits.

- **ARIES4:** ARIES4 is a slightly modified version of ARIES1. In this architecture, unit U2's register file has four registers, and each of the other two register files has two registers. The resulting instruction width is 38 bits.

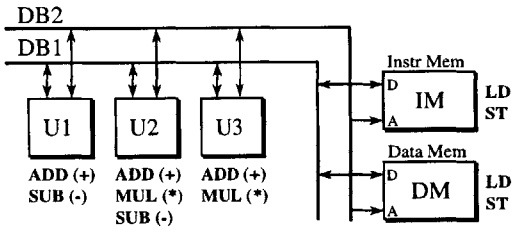
- **ARIES5:** ARIES5 is a six-way VLIW with four homogeneous functional units and one databus. In this architecture, each functional unit can perform addition and multiplication. Each functional unit has a dedicated register file with four registers per register file. This processor is a Harvard architecture. All of its data-paths are 8 bits wide and its instruction width is 42 bits.

All results presented in this chapter were obtained by running the code generator on a Sun Ultra 30/300 running Solaris 2.6.

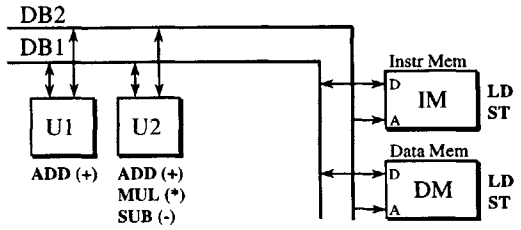
## 9.1 Code Generation Results

Several application code segments were run through the AVIV retargetable code generator. The results for five segments are reported. The C code for these examples is provided in Appendix C. These examples were selected in order to test AVIV's ability to exploit the parallelism available on the target processor while optimizing for minimum code size. In order to optimize for minimum code size, AVIV must consider trade-offs between increasing the amount of parallelism in the schedule and reducing the number of required data transfer operations. Small examples were chosen so that optimal code could be generated manually and used as a benchmark against which the solutions found by AVIV could be measured. Examples 1-2 are simple basic blocks that could be found as part of a conditional statement or for-loop. Examples 3-5 are simple basic blocks of loops that have been unrolled two or three times.

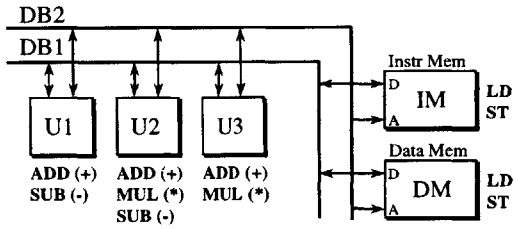
AVIV generated assembly code for these segments targeting minimum code size. The results for the ARIES1 target processor are summarized in Table 9.1. The table summarizes the number of nodes in the original machine-independent DAG representation of the application code, as well as the number of nodes in the equivalent



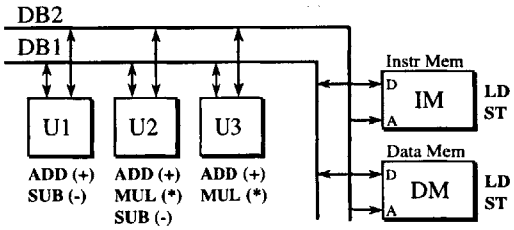

**ARIES1** (Four registers per register file. Instruction width is 44 bits.)



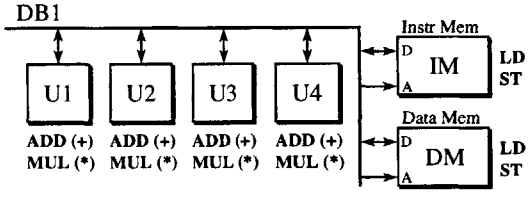

**ARIES2** (Four registers per register file. Instruction width is 35 bits.)




**ARIES3** (Two registers per register file. Instruction width is 35 bits.)




**ARIES4** (Four registers in U2's register file, and two registers elsewhere. Instruction width is 38 bits.)




**ARIES5** (Four registers per register file. Instruction width is 42 bits.)

Figure 9-1: **The ARIES1-ARIES5 target architectures**

| Basic Block | Original DAG #Nodes | Split-Node DAG #Nodes | #Instr In Solution | | CPU Time (secs) |
|---|---|---|---|---|---|
| | | | By Hand | Aviv | |
| Ex1 | 7 | 45 | 4 | 4 | 0.5 |
| Ex2 | 13 | 86 | 7 | 7 | 5.7 |
| Ex3 | 15 | 86 | 7 | 8 | 15.3 |
| Ex4 | 20 | 175 | 9 | 10 | 67.8 |
| Ex5 | 21 | 130 | 9 | 10 | 79.8 |

Table 9.1: **Code generation experiments for** ARIES1

Split-Node DAGs using the ARIES1 target architecture. It compares the number of instructions found by hand-coding to the solution found by AVIV. The hand-coded results are optimal. The number of instructions equals the number of clock cycles required to execute the application code on the target architecture. Examples 1-5 were run with four registers per register file. These examples did not require any spills to memory. For these examples, the results generated by AVIV were within one instruction of the optimal solution.

Examples 4 and 5 were then rerun with two registers per register file (ARIES3) to illustrate what happens when the required number of registers exceeds the available resources. Both of these examples resulted in spills to memory, and their results are presented in Table 9.2. Note, however, that the optimal solutions for examples 4 and 5 did not require spills. These solutions were not found by AVIV because the initial functional unit assignment cost function did not detect that the functional unit assignments it made would result in spills to memory. This situation could be avoided in one of three ways. The first approach is to modify the initial cost function to model some of the register resource requirements. This is discussed in Section 10.3. The second possible solution is to allow a greater number of functional unit assignments to be explored in detail. However, this approach would increase the total CPU time required to find a solution. The third alternative is to modify the ASIP so that the resource constraints are not as tight. Such a modification will produce better results by the code generator which can lead to an overall reduction in the size of the design. The process of modifying the ASIP will be described in greater detail in Section 9.3.

The flexibility of the AVIV retargetable code generation system allows for the ex-

| Basic Block | #Registers per RegFile | #Spills Inserted | #Instr In Solution | | CPU Time (secs) |
|---|---|---|---|---|---|
| | | | By Hand | Aviv | |
| Ex4 | 4 | 0 | 9 | 10 | 68.5 |
| Ex5 | 4 | 0 | 9 | 10 | 80.5 |
| Ex4 | 2 | 2 | 9 | 15 | 76.1 |
| Ex5 | 2 | 3 | 11 | 16 | 87.9 |

Table 9.2: **Comparison of code generation results on ARIES1 and ARIES3**

| Basic Block | Original DAG #Nodes | Split-Node DAG #Nodes | #Instr In Solution | | CPU Time (secs) |
|---|---|---|---|---|---|
| | | | By Hand | Aviv | |
| Ex1 | 7 | 22 | 4 | 5 | 0.2 |
| Ex2 | 13 | 38 | 7 | 7 | 1.2 |
| Ex3 | 15 | 31 | 8 | 9 | 2.0 |
| Ex4 | 20 | 77 | 10 | 11 | 10.0 |
| Ex5 | 21 | 39 | 13 | 14 | 6.0 |

Table 9.3: **Code generation experiments for ARIES2**

ploration of a variety of architectures until an architecture that is suited for the input application is identified. To illustrate this, the examples were recompiled targeting the ARIES2 architecture. All of the examples were run with four registers per register file, and none resulted in spills to memory. The results, summarized in Table 9.3, show that for several of the applications removing a functional unit does not degrade performance. This is better illustrated in Table 9.4 which compares the resulting code sizes for examples 1-5 targeted to the ARIES1 and ARIES2 architectures. Overall code size is defined as the number of instructions multiplied by the instruction width. Code size may decrease even if the total number of instructions increases. The results of examples 1-4 demonstrate that a more efficient architecture for those examples would be the ARIES2 architecture. However, in example 5, the number of instructions increases from 10 to 14 when it is retargeted to ARIES2. The corresponding code sizes are 440 bits on ARIES1 and 490 bits on ARIES2. These results demonstrate that the ARIES1 architecture produces smaller code for example 5. An analysis of the performance of example 5 will be provided in Section 9.3.

| Basic Block | ARIES1 | | ARIES2 | |
|---|---|---|---|---|
| | # Instructions | Code Size (bits) | # Instructions | Code Size (bits) |
| Ex1 | 4 | 176 | 5 | 175 |
| Ex2 | 7 | 308 | 7 | 245 |
| Ex3 | 8 | 352 | 9 | 315 |
| Ex4 | 10 | 440 | 11 | 385 |
| Ex5 | 10 | 440 | 14 | 490 |

Table 9.4: **Code size produced for ARIES1 versus ARIES2 architectures**

## 9.2 Analysis of AVIV

Table 9.5 presents the breakdown of the CPU time among the various tasks of code generation. This analysis reveals that the most time consuming tasks of the AVIV code generation process are the generation and validation of the maximal cliques. The generation of *valid* maximal cliques consists of generating all of the maximal cliques and then checking each clique against all of the target processor constraints in order to determine which are valid. Hence, the generation of valid maximal cliques corresponds to the summation of the two columns.

In order to reduce the total time required to generate valid maximal cliques, different constraint checking techniques were explored and optimized. There exist additional optimizations that could be incorporated into the constraint checker in order to reduce its runtime. These optimizations are discussed in Section 10.4. Nevertheless, the two subproblems involved in generating the valid maximal cliques remain NP-complete problems that are being solved exactly. As a result, long execution times are to be expected. In order to reduce the CPU time, a new algorithm that does not use maximal cliques would be required. For example, it may be possible to find cliques that are close to maximal within a shorter period of time, and base the covering algorithm on these submaximal cliques. However, in embedded system design, software synthesis is essentially as important as hardware synthesis, and as a result longer compilation times are permissible. Therefore, using maximal cliques is acceptable even though it results in long execution times.

| Basic Block | Building Precedence Matrix | Selecting Assignments | Generating Maximal Cliques | Checking Clique Validity | Covering Nodes |
|---|---|---|---|---|---|
| Ex1 | 0.00 | 0.02 | 0.03 | 0.28 | 0.02 |
| Ex2 | 0.02 | 0.06 | 0.18 | 5.12 | 0.21 |
| Ex3 | 0.03 | 0.12 | 0.90 | 13.72 | 0.35 |
| Ex4 | 0.28 | 0.95 | 3.23 | 59.70 | 2.29 |
| Ex5 | 0.12 | 1.43 | 4.08 | 70.14 | 2.62 |

Table 9.5: **Breakdown of CPU time**

## 9.3 Modifying the Target Architecture

Section 1.2 described the close relationship between the AVIV code generator and the ASIP generation subsystems. Recall that the code produced by AVIV is passed to an instruction-level simulator (ILS) that analyzes the resulting code. The ILS produces performance statistics that are passed to the ASIP generation subsystem. Based on the performance statistics, the ASIP generation subsystem can determine how, and if, to modify the ASIP in order to better satisfy the requirements of the input application. This process, referred to as the software synthesis loop, is repeated until a suitable ASIP design is identified.

In particular, the ILS produces an execution trace of the program that provides the dynamic instruction frequencies. This information combined with the code generated by AVIV provides the dynamic operation frequencies. The operation frequencies are used to derive the utilization statistics of each operation and functional unit. The ASIP generation module uses the utilization data to identify bottlenecks as well as redundant hardware in the architecture.

In evaluating the code generated in the experiments of Table 9.1 (results for ARIES1 architecture), utilization statistics show that some of the operations are redundant for some of the applications. If the ARIES1 architecture is modified by removing these redundant operations, ARIES2 results. In terms of overall code size, the modified architecture (ARIES2) results in improved performance over the original architecture (ARIES1). Table 9.6 shows the utilization statistics for example 4 for both the ARIES1 and ARIES2 target processors. These statistics illustrate that in

| Target Processor | U1 | | U2 | | | U3 | | DB1 | DB2 |
|---|---|---|---|---|---|---|---|---|---|
| | ADD | SUB | ADD | SUB | MUL | ADD | MUL | | |
| ARIES1 | 4 | 2 | 1 | 0 | 1 | 1 | 1 | 7 | 6 |
| ARIES2 | 1 | – | 5 | 2 | 2 | – | – | 6 | 6 |

Table 9.6: **Utilization statistics for example 4** (– means the operation is nonexistent for the current architecture)

| Target Processor | U1 | | U2 | | | U3 | | DB1 | DB2 |
|---|---|---|---|---|---|---|---|---|---|
| | ADD | SUB | ADD | SUB | MUL | ADD | MUL | | |
| ARIES1 | 1 | 2 | 0 | 2 | 3 | 0 | 3 | 8 | 6 |
| ARIES2 | 0 | – | 1 | 4 | 6 | – | – | 5 | 5 |

Table 9.7: **Utilization statistics for example 5** (– means the operation is nonexistent for the current architecture)

ARIES1 functional units `U2` and `U3` are underutilized in comparison to the databuses; hence, some of the functionality of these units could be removed without affecting the resulting code size. The most highly utilized operation is the `add` operation; thus, it remains available on two distinct functional units in ARIES2. The `sub` and `mul` operations, on the other hand, are not as heavily utilized. Therefore, they can coexist on a single functional unit without a performance penalty. The utilization statistics for ARIES2 illustrate that the functional units and databuses are more equally balanced. The resulting code size is 385 bits on ARIES2. This is indeed smaller than the code size of 440 bits on ARIES1.

Consider the utilization statistics of example 5, presented in Table 9.7 on the two target architectures. In this case, the functional unit utilization in ARIES1 is more balanced, and removing the third functional unit plus the extra subtract operation results in a bottleneck between the `sub` and `mul` operations. As a result, the average parallelism achieved on ARIES2 compared to ARIES1 is 1.5 operations per instruction versus 2.5 operations per instruction, respectively. These results demonstrate that for example 5, ARIES1 is the more suitable architecture because its resulting code size is smaller. For this example, the code size is 440 bits on ARIES1 and 490 bits on ARIES2.

In addition to examining the utilization statistics, the spill information provided

by the code generator is used to determine whether or not additional registers should be provided. The addition of a few registers could reduce the total number of instructions, thereby reducing the total code size. For example, Table 9.2 illustrated that when example 4 was compiled for the ARIES3 architecture (with two registers per register file), it resulted in two spills to memory and a total of 15 instructions. A closer examination of the utilization statistics for this example reveals that the spills to memory are only required from unit U2's register file. The same code segment run on ARIES4 (where two additional registers are added to unit U2's register file) results in no spills to memory and a total of nine instructions[1]. The cost of the additional registers is 64 transistors (1 functional unit $\times$ 2 registers $\times$ 8 bits $\times$ 4 transistors), whereas the cost of the additional six instructions is 183 transistors ((35 bits $\times$ 15 instructions) $-$ (38 bits $\times$ 9 instructions)). The area requirements for the additional registers is less than the area saved in the program ROM. These results indicate that the spill information provided by AVIV assists the ASIP generation module in determining how to modify the ASIP to better satisfy the requirements of the application.

## 9.4 Exploring the Performance Capabilities of the AVIV Code Generator

In order to explore the capabilities of AVIV, the ARIES5 architecture was used. Recall that ARIES5 is a six-way VLIW with four identical functional units. Each functional unit is able to perform addition and multiplication, and each functional unit has its own dedicated register file.

The following experiment tests how the size of the basic blocks and the amount of parallelism available on the target architecture affect the runtime of the AVIV compiler and the quality of the results it produces. The input application for this experiment consists of multiple parallel strands of operations each of which consists of identical sequential operations. The experiment tests the effect of varying the number

[1]Although ARIES4 has tighter register constraints than ARIES1, the register constraints lead to a more efficient solution.

of parallel strands and the size of each strand. The basic strand of operations contains the following sequential operations:

```
a = a + a;
a = a * a;
a = a + a;
a = a * a;
a = a + a;
```

These operations are then used as a base for growing the size of the application. The code above is a one-way parallel application whose basic strand contains five operations. To create a similar application with two-way parallelism, the basic strand is duplicated using a new variable. This results in two parallel strands of identical sequential operations shown below:

```
a = a + a;
a = a * a;
a = a + a;
a = a * a;
a = a + a;

b = b + b;
b = b * b;
b = b + b;
b = b * b;
b = b + b;
```

Three-way and four-way parallel applications are created by further duplicating the basic strand using new variables. The reason for testing such a regular application is because it is easy to determine the optimal solution. In these examples, the optimal solution (ignoring the required load operations) is to produce five compacted

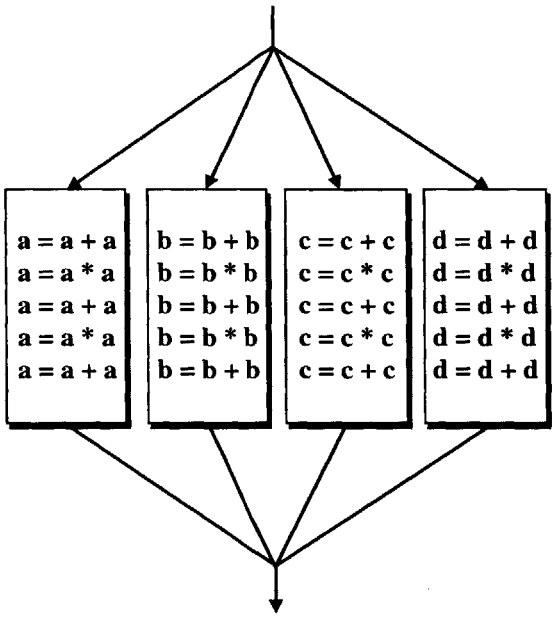



Figure 9-2: **Data flow of a four-way parallel application**

instructions, each of which includes one operation in the case of the one-way parallel application, two operations in the case of the two-way parallel application, and so on.

The applications are further modified by expanding the size of the basic strand to 10, 15, and 20 sequential operations. The resulting data flow is unchanged. Figure 9-2 illustrates the data flow for the four-way parallel application. Note that this figure corresponds to a single basic block (i.e., it is a basic block data-flow graph not a control-flow graph), consisting of multiple parallel strands of operations.

The timing measurements produced by running these applications on AVIV targeting the ARIES5 architecture are shown in Figure 9-3. Each curve in the graph corresponds to a single strand size. The total basic block size corresponding to any point on the graph can be determined by multiplying the strand size by the number of parallel strands. For example, if the strand size is 10 operations and the amount of parallelism is 3-way, then the total basic block size is 30 operations. The results presented indicate that along any given curve, although an application may be growing linearly in size (by adding parallel strands), the computation time grows exponen-

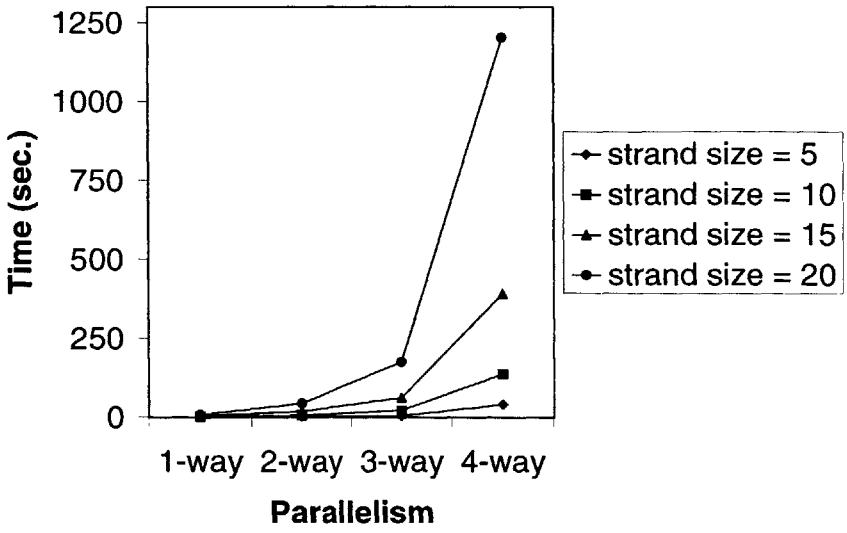



Figure 9-3: **The effect of block size and data parallelism on runtime**

tially. Furthermore, as the length of each strand increases linearly, the computation time increases significantly faster. These relationships illustrate the complexity of finding optimal solutions as the size of the basic blocks and the amount of parallelism in the application increase. Although the complexity of the problem increases as the size of the strands is enlarged and additional parallelism is added, AVIV continues to find the optimal solution for this application.

The experiment described above consisted of a very regular input application for which AVIV was able to produce optimal code. The next experiment considers an input application that is not comprised of such a regular structure. It will be shown that there too AVIV is able to exploit the parallelism available on the target architecture in order to arrive at an optimal solution.

The data flow graph for the next experiment is shown in Figure 9-4. Each block is marked according to the strand of code to which it corresponds and the number of operations in that strand. The strand corresponding to variable `a` consists of 15 total instructions. It is split into two blocks because strands `b` and `c` depend on the data

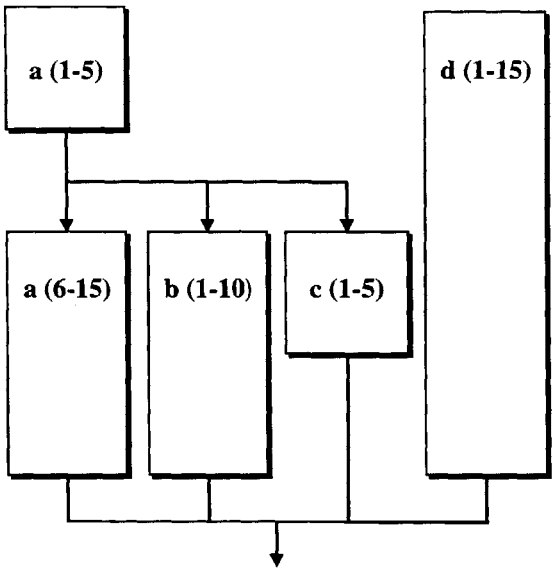



Figure 9-4: **Data flow graph example II**

produced by the fifth operation of strand a. Strand d is independent of strands a, b, and c. In this data flow graph, the amount of parallelism is not constant.

Figure 9-5 shows the schedule produced by AVIV for this application. It includes the required load operations and data transfer operations. The operations are numbered according to their corresponding strand operation index. Note that the $b_i$ and $c_i$ operations cannot be scheduled until after $a_5$ has been scheduled. Furthermore, each compacted instruction cannot contain more than one load or data transfer operation. This generated schedule demonstrates that AVIV was able to find the parallelism available in the input application and schedule the instructions optimally.

One observation to note about this schedule is that the load operation for strand c is only scheduled in cycle 9 rather than as early as cycle 4. This is an artifact of the heuristic used by AVIV to reduce the number of maximal cliques that must be generated. As mentioned in Chapter 5, this heuristic disallows the merging of nodes whose levels measured from the top, or bottom, of the Split-Node DAG differ by a value that is greater than the acceptable level difference. The level difference used in the current implementation is a distance of two. This implies that operation $a_3$,

| Cycle | U1 | U2 | U3 | U4 | Memory | Databus |
|---|---|---|---|---|---|---|
| 1 | | | | | load a | |
| 2 | | $a_1$ | | | load d | |
| 3 | | $a_2$ | $d_1$ | | load b | |
| 4 | | $a_3$ | $d_2$ | | | |
| 5 | | $a_4$ | $d_3$ | | | |
| 6 | | $a_5$ | $d_4$ | | | |
| 7 | | $a_6$ | $d_5$ | | | tr $a_5$ to U4 |
| 8 | | $a_7$ | $d_6$ | $b_1$ | | tr $a_5$ to U1 |
| 9 | | $a_8$ | $d_7$ | $b_2$ | load c | |
| 10 | $c_1$ | $a_9$ | $d_8$ | $b_3$ | | |
| 11 | $c_2$ | $a_{10}$ | $d_9$ | $b_4$ | | |
| 12 | $c_3$ | $a_{11}$ | $d_{10}$ | $b_5$ | | |
| 13 | $c_4$ | $a_{12}$ | $d_{11}$ | $b_6$ | | |
| 14 | $c_5$ | $a_{13}$ | $d_{12}$ | $b_7$ | | |
| 15 | | $a_{14}$ | $d_{13}$ | $b_8$ | | |
| 16 | | $a_{15}$ | $d_{14}$ | $b_9$ | | |
| 17 | | | $d_{15}$ | $b_{10}$ | | |

Figure 9-5: **Schedule for data flow graph II**

whose level from the bottom is three, cannot be merged with the load of variable c, whose level from the bottom is zero[2]. As a result, the load of variable c is postponed until cycle nine. At cycle nine, operation $a_9$, whose level from the top is six, can be merged with the load, whose level from the top is five. Although in this example, the heuristic did not deter AVIV from finding an optimal schedule, it is clear that there are situations in which the heuristic will fail. In these situations, either the acceptable level distance could be increased, or the heuristic could be turned off. In either case, the computation time will increase, but the solutions generated will be closer to optimal.

The levels heuristic plays a significant role in reducing the runtime of the AVIV code generation process. This heuristic prevents the merging of nodes whose levels differ by an amount that is greater than the acceptable level difference. The smaller the acceptable level difference, the fewer the number of maximal cliques generated.

[2]The level count begins at zero. Note that the top of the Split-Node DAG corresponds to the bottom of the data flow graph.

Consequently, a smaller acceptable level difference results in reduced runtimes. However, it can potentially produce less optimal solutions.

The levels heuristic is influenced by the structure of the Split-Node DAG. The Split-Node DAG structure is directly related to the structure of the input DAG. Depending on how the input program is written, the structure of the input DAG may vary. These relationships are explored through an example by reexamining the application of Figure 9-2. This application consists of four parallel strands of operations. The actual application tested added the results produced by each strand. The parallel strands were augmented with these additional add operations to prevent their removal during the process of dead-code elimination. In other words, the code within the parallel strands would have been removed if the result of each strand was not subsequently utilized.

A noteworthy observation about this application is that the format in which the final add operations are specified, in the input program, can affect the results produced by AVIV. The reason is that, depending on how the input program is written, different DAGs could be created. For example, if the results of the four strands were added using the following statement:

```
out = (a + b) + (c + d);
```

then the add operations would be converted into the DAG illustrated in Figure 9-6 (a). However, if the order of execution of the add operations is not specified, as shown in the statement below:

```
out = a + b + c + d;
```

then the compiler front-end would convert the operations into the DAG illustrated in Figure 9-6 (b).

The example described above demonstrates that the format of the input program can affect the input to the AVIV code generator. The levels of the nodes in the Split-Node DAG depend on the structure of the input DAG. The levels of the nodes influence the heuristic that determines which nodes can be merged into single cliques. As a result, different input specifications result in the generation of different sets of

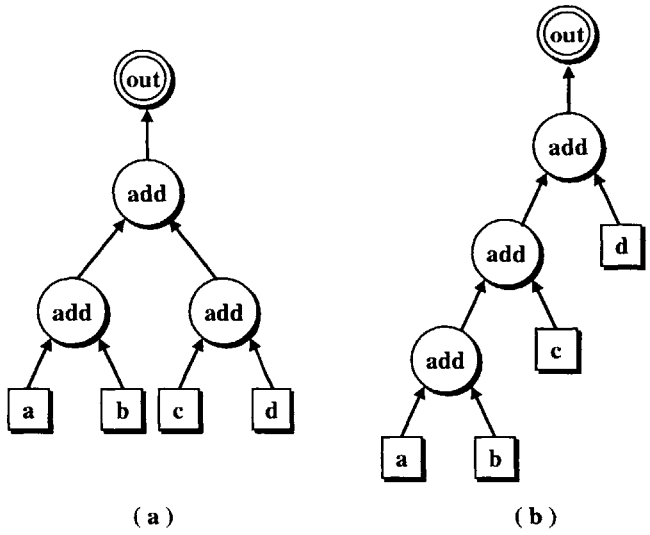



Figure 9-6: **Input program structure can produce different DAGs (a)** DAG for `out = (a + b) + (c + d);` statement **(b)** DAG for `out = a + b + c + d;` statement

maximal cliques. Furthermore, both the runtime and the results can be affected by the format of the input program. Specifically, the four-way parallel application tested with strand sizes of 20 operations and augmented with the operations of Figure 9-6 (a) resulted in a runtime of 1203 seconds and found the optimal solution that requires 26 instructions. The same application augmented with the operations of Figure 9-6 (b) resulted in a significantly shorter runtime of 357 seconds and a solution consisting of 27 instructions. This occurred because the operations in Figure 9-6 (b) removed the symmetry in the levels of the nodes within each parallel strand. Consequently, fewer combinations of nodes could be merged into cliques. As a result, a shorter runtime and a less optimal solution were obtained.

# Chapter 10

# Conclusions

This thesis demonstrates the feasibility of developing an automatically retargetable code generator. It addresses the interdependence of the instruction selection, resource allocation, and scheduling phases of code generation by providing an integrated environment for solving all three problems concurrently. The AVIV retargetable code generator was constructed to explore these ideas. Given an input application written in C and an ISDL target processor description, AVIV produces optimized machine code that implements the functionality of the input application and can be executed on the target processor. The Split-Node DAG machine-specific representation, used by AVIV, explicitly represents all possible ways of implementing the input application on the target processor. This includes the representation of data transfer operations required to route the data from one resource to another. The information embedded in the Split-Node DAG, together with the detailed information provided by AVIV's databases, enable AVIV's heuristic scheduling algorithm to select and schedule target processor operations that implement the input application with a minimal-cost set of target processor instructions. The instructions selected correspond to compacted instructions that cover both the operation and data transfer nodes of the Split-Node DAG simultaneously. In covering the nodes, AVIV targets minimum code size because code size is a primary concern in the design of embedded systems that are implemented on a single integrated circuit.

This chapter summarizes AVIV's contributions and presents an analysis of the

various components of the AVIV retargetable code generator. In addition, suggestions of possible optimizations that may improve the overall performance of AVIV are provided.

## 10.1 A Complete Working System

This thesis presents a complete working retargetable code generation system. Retargetable code generation in AVIV includes parsing the ISDL description of the target processor in order to extract all of the information required by the code generator. The extracted information is stored in databases that simplify its access during later stages of the code generation process. The compiler front-end transforms the input application into a set of basic block DAGs whose nodes correspond to SUIF operations. A control flow graph maintains the relationship among the multiple basic block DAGs. The AVIV compiler uses its databases to correlate the SUIF operations to operations available on the target processor. The databases also provide information about the data transfer paths of the target architecture. The operation correlations and the data transfer paths are used to convert the machine-independent basic block DAGs into machine-specific Split-Node DAGs.

Once the Split-Node DAGs have been created, AVIV's heuristic covering algorithm is applied to cover the Split-Node DAGs with a minimal-cost set of target processor instructions. This process includes: (1) identifying a set of functional unit and data transfer assignments to explore in detail, (2) creating maximal cliques for each assignment, (3) checking the validity of the cliques using the constraint checker, and (4) covering the Split-Node DAG with the best cliques. The cost function in the covering step accounts for the necessary operation nodes, the data transfer nodes, and any load and spill nodes resulting from insufficient register resources. The covering step outputs the selected instructions in the assembly format of the target architecture. Control flow, in AVIV , is handled by maintaining the control flow optimizations determined by the SUIF compiler. This is achieved by producing assembly instructions for each basic block in the order specified by the SUIF compiler. The instructions

within each basic block contain control flow operations that transfer the flow of control appropriately. The assembly instructions are then assembled into binary instructions using the automatically generated target processor assembler.

## 10.2 The Split-Node DAG

The Split-Node DAG representation is a novel way of representing all possible ways of implementing an input application on the target processor. One of the distinguishing characteristics of the Split-Node DAG is its inclusion of data transfer nodes. The data transfer nodes enable the code generation algorithm to account for all required data routing operations when it performs instruction selection. This permits instruction selection and resource allocation to be merged into a single code generation phase. The major benefit of the Split-Node DAG representation is that it contains all of the information necessary for considering the trade-offs in the various tasks of code generation.

The only drawback of the Split-Node DAG is that, as a result of incorporating all possible target processor operations that can implement each source operation and including all possible paths for the required data transfers, the size of the Split-Node DAG is significantly larger than the original basic block DAG. This implies that analyzing the Split-Node DAG data structure is more time consuming than analyzing its corresponding basic block DAG. However, the Split-Node DAG has a very regular structure that simplifies its analysis. Furthermore, the benefits achieved by using the Split-Node DAG for code generation warrant the use of a larger data structure.

## 10.3 The Quality of the Code Produced Using AVIV's Heuristics

AVIV's code generation algorithm prunes the search space of possible functional unit and data transfer assignments using a heuristic cost function. The quality of the code produced is dependent on the ability of this function to select assignments that lead

to optimal, or close to optimal, solutions. For architectures that are well suited for the input application, this heuristic cost function is quite accurate and can determine an appropriate set of assignments to explore in detail. In these cases, the code generated by AVIV is very close to optimal for basic blocks.

There are two types of situations that may lead to poor assignment selections. The first occurs when the register constraints are tight. The problem is that AVIV's search space pruning cost function does not consider register availability. As a result, there may exist a functional unit assignment that requires fewer register resources than the selected assignments. If the selected assignments require spills to memory, and the unselected assignments could avoid the spills, then AVIV's solution may not be close to optimal. It is not clear how the register resource constraints can be incorporated into the pruning cost function. Register resource constraints are dependent on the chosen schedule which is unknown during the search space pruning step. However, there are certain types of register resource constraints that could be modeled in the pruning cost function. For example, if the register resources are not uniform (e.g., the register file of one functional unit is larger than all the rest), then the pruning cost function could model the non-uniformity. In order to achieve this, an operation on a functional unit with few register resources can be modeled as a more expensive operation than one with greater register resources. Future work could explore more accurate modeling of the register resources in the pruning cost function.

The second situation that may lead to poor assignment selections occurs when the amount of parallelism in the basic blocks is greater than the available parallelism in the target architecture. In this case, the pruning cost function will likely determine that each set of sequential operations should be assigned to a single functional unit. The cost function selects these assignments in order to avoid inserting additional data transfer operations. This decision is correct if the available parallelism is greater than, or equal to, the amount of parallelism in the basic block. However, if the available parallelism is insufficient, then the optimal solution would probably interleave the functional unit use among the multiple parallel strands of operations. Generally this yields fewer total instructions than if each set of sequential operations is assigned to

a single functional unit.

Although these weaknesses exist in the pruning cost function, the AVIV code generator is optimized for generating high-quality code for architectures that match the needs of the application. For these types of architectures, it is unlikely that the two situations described above will be encountered. For architectures that are well matched to the application, AVIV will produce results that are close to optimal.

AVIV only optimizes code within basic blocks. It currently does not perform any control flow optimizations. Incorporating control flow optimizations, such as code motion or utilization of zero-overhead looping hardware, could further reduce the size of the generated code. Furthermore, optimizations such as trace scheduling can probably improve execution time. However, these optimizations may not reduce the total code size because several basic blocks may need to be duplicated. The study of control flow optimizations that target minimum code size is another area for future work.

## 10.4 Generating Maximal Cliques and Verifying their Validity

The goal of AVIV's code generation algorithm is to determine a set of target processor instructions that will cover the nodes of the Split-Node DAG using a minimal-cost set of target processor instructions. AVIV's focus is on architectures with instruction-level parallelism. In order to exploit the parallelism available on the target architecture efficiently, AVIV generates maximal cliques of Split-Node DAG nodes that can be used to cover the Split-Node DAG. A maximal clique represents the compaction of multiple operations into a single target processor instruction. Before beginning the covering procedure, AVIV must ensure that the generated maximal cliques satisfy all of the constraints of the target processor. Otherwise, the maximal cliques must be divided into multiple smaller cliques that satisfy the constraints. The resulting cliques represent the largest valid groupings of operations into instructions.

The runtime for the various components of the code generation process were measured and reported in Chapter 9. These runtimes indicated that the tasks of generating the maximal cliques and checking them against the constraints are the most time consuming portions of the code generation process. These results are expected because generating maximal cliques and checking the constraints are both NP-complete problems which AVIV solves exactly. In order to reduce these runtimes, several alternatives exist. These alternatives are summarized below.

Instead of generating all maximal cliques, it may be possible to identify a heuristic that can generate submaximal cliques that produce high quality results. Producing submaximal cliques reduces the runtime. Alternatively, a heuristic that creates a subset of maximal cliques could be formulated. Producing a subset of maximal cliques should be faster than producing all maximal cliques. Since there is substantial overlap among the maximal cliques, identifying a subset of maximal cliques could still lead to an acceptable solution. This will be the case if the heuristic can identify which nodes are good candidates for merging into a single clique, and which nodes are not.

Another approach to improving the overall runtime of the code generation algorithm is to attempt to incorporate all pairwise constraints (i.e., constraints between exactly two operations) directly into the pairwise parallelism matrix. This would reduce the number and size of the maximal cliques generated, thereby reducing the runtime required to produce the maximal cliques. This approach would also shorten the time spent in the constraint checker because all pairwise constraints could be eliminated. Examination of the constraints for multiple target architectures indicates that most constraints are in fact pairwise constraints. Therefore, such an optimization has a high likelihood of significantly reducing the runtime. The runtime will be reduced if the number of zeroes in the original matrix is smaller than the number of cliques that would have been generated from the original matrix. This is the case because each zero entry of the pairwise parallelism matrix must be compared to all of the pairwise constraints in order to determine whether or not the entry should remain zero. Recall that a zero entry implies that the pair of nodes corresponding to that entry can be executed in parallel.

Another optimization that can reduce the CPU time allocated to constraint checking analyzes the symmetry in the target architecture. Based on this symmetry, the proposed optimization identifies which cliques would return equal values from the constraint checker. This type of analysis would allow one clique to be passed through the constraint checker, and its result could be utilized by all of the cliques that are symmetrically equivalent. For example, consider two equivalent operations A and B that belong to two different functional units whose functionality and resource conflicts are identical. In this architecture, a clique of the form A, C, D would return the same value from the constraint checker as clique B, C, D. This implies that only one of the cliques needs to be passed through the constraint checker, and its result can be used by both cliques. Analyzing the target architecture in order to identify such symmetries could significantly reduce the total amount of time spent in the constraint checker.

A similar optimization could be made within the constraint checker to reduce its runtime. This optimization would identify common subexpressions that are present in multiple constraints. It would store the result of matching these subexpressions to the input instruction. Subsequently, every time the subexpression is encountered in the constraints, the stored information can be retrieved. This reduces the runtime of the constraint checker because fewer matches need to be executed.

The proposed optimizations discussed above suggest areas for further development. These types of optimizations reduce the runtime of retargetable code generation.

## 10.5 The Covering Algorithm

The covering algorithm used by AVIV covers the Split-Node DAG from the bottom-up. By covering the Split-Node DAG from the bottom-up, a schedule is automatically generated while performing instruction selection. The covering algorithm's selection criterion is based on covering as many nodes as possible within a single instruction. A node can only be covered if there are sufficient register resources available for its

execution. If the register resources are insufficient, the covering algorithm determines which operation's result should be spilled to memory. It then inserts the required load and spill nodes into the Split-Node DAG and regenerates maximal cliques for the remaining uncovered nodes. This is done in order to incorporate the load and spill operations into the set of instructions being considered for covering the Split-Node DAG. The result of this covering procedure is that instruction selection, resource allocation, insertion of loads and spills, scheduling, and compaction are all optimized concurrently. As a result, AVIV is able to produce highly optimized code for the input application.

## 10.6 Using AVIV in the Software Synthesis Loop

In an effort to identify the most appropriate architecture for the input application, various target processor architectures must be explored. AVIV's retargetability supports this exploration. The architecture exploration process begins with an analysis of the application's requirements in order to suggest an initial target processor. This analysis is performed by the ASIP generation system. It provides AVIV with a description of the processor using ISDL. AVIV produces code for this architecture. Based on utilization statistics produced by the simulator, load and spill information provided by AVIV, and design costs provided by the hardware model, the ASIP generation module modifies the ASIP architecture to better satisfy the application's requirements. AVIV's ability to consider the trade-offs in the code generation tasks and its ability to exploit the parallelism available on different target architectures permits the identification of an ASIP and corresponding code that are well suited for the input application. This design process supports the automated synthesis of the software component of embedded systems.

This design process could be enhanced by having the code generator provide additional feedback to the ASIP generation module. The results presented in Section 9.3 demonstrated that providing the ASIP generation module with information about the required load and spill operations can assist in modifying the target processor.

The code generator could also assist the ASIP generation module in identifying functional unit or operation bottlenecks in the architecture. AVIV is well equipped to provide this information. In order to identify areas with scarce resources, it could analyze the Split-Node DAG and the cliques generated. For example, if several nodes in the Split-Node DAG are independent and could be executed in parallel but are not merged into a single clique, then their required resources may warrant expansion. In addition, the code generator may be able to identify regions of the architecture that are critical and should not be modified. This could be achieved by searching for common operations in the functional unit and data transfer assignments selected. These common operations could represent operations that are critical to producing efficient code and should not be removed by the ASIP generation module. These types of analyses could assist in identifying the optimal architecture for a given application.

## 10.7 Summary of Directions for Future Work

In considering the performance of the AVIV retargetable compiler, areas that warrant continued development have become apparent. These areas fall into two categories. The first category deals with improving the code generated by AVIV by introducing additional optimizations that further decrease the size of the code generated. The second category addresses reducing the runtime required to generate optimized code.

In order to improve the quality of the code produced by AVIV, two areas of development were suggested. The first concerns the modeling of the register resources within AVIV's functional unit assignment pruning cost function. Functional unit assignments that result in many spills to memory should not be selected if better alternatives exist. Incorporating the register resource estimation into the pruning cost function could prevent this situation from occurring. The second area concerns the incorporation of control flow optimizations into AVIV. These control flow optimizations should be geared towards minimizing the overall code size.

There are additional optimizations that have not yet been discussed. The primary one concerns memory allocation. Significant code size improvements could be

attained if the data was distributed among multiple memories in an optimized manner. Furthermore, the ordering of the data within each memory unit could affect the code generator's ability to utilize specialized addressing modes that reduce the overall code size. Incorporating intelligent memory allocation into AVIV is another area that could be explored in future work.

In order to reduce the runtime of AVIV, several possible optimizations were suggested. These optimizations attempt to accelerate the most time consuming tasks of AVIV which are maximal clique generation and constraint checking. In order to reduce the runtime of clique generation, a subset of maximal cliques could be generated heuristically. In addition, by checking for pairwise constraints before generating the cliques, the number of cliques generated could be reduced. The pairwise constraint optimization reduces both the time spent in clique generation and the time spent in the constraint checker. In order to further reduce the time spent on constraint checking, two additional optimizations were suggested. The first concerns the use of symmetry in the architecture to reduce the number of times that the constraint checker must be called. The second identifies common subexpressions in the constraints in order to eliminate unnecessary matches within the constraint checker. These optimizations are among those that could improve the runtime of AVIV's code generation algorithm.

Finally, the AVIV code generator could analyze the application's resource requirements in order to provide feedback to the ASIP generation module. This feedback could assist in the identification of the most suitable architecture for the input application. As suggested above, the code generator could identify bottlenecks in the architecture by analyzing the Split-Node DAG structure and the resulting cliques. In addition, an analysis of the functional unit and data transfer assignments could identify critical resources of the architecture. Based on these analyses, the ASIP generation module should be better equipped to identify an appropriate target processor. The optimized AVIV code generator could then be used to generate code for this processor.

Collectively, these methods suggest an approach for further enhancement of the software component of embedded systems.

# Appendix A

# An Example ISDL Description

```
Section Format

U1      = OP[2], RA[2], RB[2], RC[2];
U2      = OP[2], RA[2], RB[2], RC[2];
U3      = OP[2], RA[2], RB[2], RC[2];
DB1     = SRC[5], DEST[5];
DB2     = SRC[5], DEST[5];


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

Section Global_Definitions

//    assembly      token       value
Token "U1.R"[0..3]    U1_R        { [0..3]; };
Token "U2.R"[0..3]    U2_R        { [0..3]; };
Token "U3.R"[0..3]    U3_R        { [0..3]; };

Non_Terminal U1_RA: U1_R { $$ = U1_R; } {U1[U1_R]} {} {} {} ;
Non_Terminal U1_RB: U1_R { $$ = U1_R; } {U1[U1_R]} {} {} {} ;
Non_Terminal U1_RC: U1_R { $$ = U1_R; } {U1[U1_R]} {} {} {} ;

Non_Terminal U2_RA: U2_R { $$ = U2_R; } {U2[U2_R]} {} {} {} ;
Non_Terminal U2_RB: U2_R { $$ = U2_R; } {U2[U2_R]} {} {} {} ;
Non_Terminal U2_RC: U2_R { $$ = U2_R; } {U2[U2_R]} {} {} {} ;

Non_Terminal U3_RA: U3_R { $$ = U3_R; } {U3[U3_R]} {} {} {} ;
```

```
Non_Terminal U3_RB: U3_R { $$ = U3_R; } {U3[U3_R]} {} {} {} ;
Non_Terminal U3_RC: U3_R { $$ = U3_R; } {U3[U3_R]} {} {} {} ;

Non_Terminal SRC:  U1_R { $$ = 0x00 | U1_R; } {U1[U1_R]} {} {} {} |
                   U2_R { $$ = 0x04 | U2_R; } {U2[U2_R]} {} {} {} |
                   U3_R { $$ = 0x08 | U3_R; } {U3[U3_R]} {} {} {} ;

Non_Terminal DEST: U1_R { $$ = 0x00 | U1_R; } {U1[U1_R]} {} {} {} |
                   U2_R { $$ = 0x04 | U2_R; } {U2[U2_R]} {} {} {} |
                   U3_R { $$ = 0x08 | U3_R; } {U3[U3_R]} {} {} {} ;

#define REG SRC
#define LOC INT


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

Section Storage

//                                      = entries , bits_per_entry
Instruction Memory INST                 = 0x100 , 0x2C
Memory DM                               =  0x20 ,  0x8
RegFile U1                              =   0x4 ,  0x8
RegFile U2                              =   0x4 ,  0x8
RegFile U3                              =   0x4 ,  0x8
ProgramCounter PC                       =           0x8


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

#define DEFINE_NULL_OP  {} { NULLOP(); } {} {} {}

#define ADDm(x,y)    ADD(x,y,8,"trn")
#define SUBm(x,y)    SUB(x,y,8,"trn")
#define MULm(x,y)    MUL(x,y,8,8,"trn")

Section Instruction_Set

Field U1f:
      U1_NULL DEFINE_NULL_OP
      U1_add U1_RA, U1_RB, U1_RC
         { U1.OP = 0x0; U1.RA = U1_RA; U1.RB = U1_RB; U1.RC = U1_RC; }
         { U1_RC <- ADDm(U1_RA,U1_RB); }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
```

```
        { Latency = 1; Usage = 1; }
      U1_sub U1_RA, U1_RB, U1_RC
        { U1.OP = 0x1; U1.RA = U1_RA; U1.RB = U1_RB; U1.RC = U1_RC; }
        { U1_RC <- SUBm(U1_RA,U1_RB); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
      U1_nop
        { U1.OP = 0x3; }
        { NOP(); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }

Field U2f:
      U2_NULL DEFINE_NULL_OP
      U2_add U2_RA, U2_RB, U2_RC
        { U2.OP = 0x0; U2.RA = U2_RA; U2.RB = U2_RB; U2.RC = U2_RC; }
        { U2_RC <- ADDm(U2_RA,U2_RB); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
      U2_sub U2_RA, U2_RB, U2_RC
        { U2.OP = 0x1; U2.RA = U2_RA; U2.RB = U2_RB; U2.RC = U2_RC; }
        { U2_RC <- SUBm(U2_RA,U2_RB); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
      U2_mul U2_RA, U2_RB, U2_RC
        { U2.OP = 0x2; U2.RA = U2_RA; U2.RB = U2_RB; U2.RC = U2_RC; }
        { U2_RC <- MULm(U2_RA,U2_RB); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
      U2_nop
        { U2.OP = 0x3; }
        { NOP(); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }

Field U3f:
      U3_NULL DEFINE_NULL_OP
      U3_add U3_RA, U3_RB, U3_RC
        { U3.OP = 0x0; U3.RA = U3_RA; U3.RB = U3_RB; U3.RC = U3_RC; }
```

```
        { U3_RC <- ADDm(U3_RA,U3_RB); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
    U3_mul U3_RA, U3_RB, U3_RC
        { U3.OP = 0x1; U3.RA = U3_RA; U3.RB = U3_RB; U3.RC = U3_RC; }
        { U3_RC <- MULm(U3_RA,U3_RB); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
    U3_nop
        { U3.OP = 0x3; }
        { NOP(); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }

// DB1 is used for the data
Field DB1:
    DB1_NULL DEFINE_NULL_OP
    DB1_move SRC, DEST
        { DB1.SRC = SRC; DB1.DEST = DEST; }
        { DEST <- SRC; }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
    DB1_move_im INT, DEST
        { DB1.SRC = 0x10 | (INT & 0xF); DB1.DEST = DEST; }
        { DEST <- INT; }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
    DB1_nop
        { DB1.DEST = 0x1F; }
        { NOP(); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }

// DB2 is used for the address
Field DB2:
    DB2_NULL DEFINE_NULL_OP
    DB2_move SRC, DEST
        { DB2.SRC = SRC; DB2.DEST = DEST; }
        { DEST <- SRC; }
```

```
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
    DB2_move_im INT, DEST
        { DB2.SRC = 0x10 | (INT & 0xF); DB2.DEST = DEST; }
        { DEST <- INT; }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
    DB2_nop
        { DB2.DEST = 0x1F; }
        { NOP(); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }

#define DMdata 0x0C
#define DMaddr 0x0D

Field DMf:
    DM_NULL DEFINE_NULL_OP
    // DB1.SRC gets code for DMdata
    // DB2.DEST gets code for DMaddr
    DM_ld   REG, LOC
        { DB1.SRC = DMdata; DB1.DEST = REG;
          DB2.SRC = LOC; DB2.DEST = DMaddr; }
        { REG <- DM[LOC]; }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
    // DB1.DEST gets code for DMdata
    // DB2.DEST gets code for DMaddr
    DM_st   REG, LOC
        { DB1.SRC = REG; DB1.DEST = DMdata;
          DB2.SRC = LOC; DB2.DEST = DMaddr; }
        { DM[LOC] <- REG; }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }

#define IMdata 0x0E
#define IMaddr 0x0F

Field IM:
    IM_NULL DEFINE_NULL_OP
```

```
      // DB1.SRC gets code for IMdata
      // DB2.DEST gets code for IMaddr
      IM_ld   REG, LOC
         { DB1.SRC = IMdata; DB1.DEST = REG;
           DB2.SRC = LOC; DB2.DEST = IMaddr; }
         { REG <- INST[LOC]; }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
      // DB1.DEST gets code for IMdata
      // DB2.DEST gets code for IMaddr
      IM_st   REG, LOC
         { DB1.SRC = REG; DB1.DEST = IMdata;
           DB2.SRC = LOC; DB2.DEST = IMaddr; }
         { INST[LOC] <- REG; }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }


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

Section Constraints

// SRC and DEST cannot be the same on either bus
~( DB*_move U@[1].R*, U@[1].R* )

// Cannot use buses for a move between register files if a memory
// operation is using the buses
~( ((DB*_move* *,*) | (DB*_nop)) & ((DM_* *) | (IM_* *)) )

// Cannot do both a DM and IM operation
// because they use the same buses
~( (DM_* *) & (IM_* *) )

// Cannot write to same register from two different operations
~( (DB1_move* *,@[1]) & (DB2_move* *,@[1]) )


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

Section Optional
```

# Appendix B

# An Example ISDL Description Including Control Flow

```
Section Format

Control = OP[2], SRC[4];
U1      = OP[2], RA[2], RB[2], RC[2];
U2      = OP[2], RA[2], RB[2], RC[2];
U3      = OP[2], RA[2], RB[2], RC[2];
DB1     = SRC[5], DEST[5];
DB2     = SRC[5], DEST[5];


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

Section Global_Definitions

//    assembly      token       value
Token "U1.R"[0..3]    U1_R        { [0..3]; };
Token "U2.R"[0..3]    U2_R        { [0..3]; };
Token "U3.R"[0..3]    U3_R        { [0..3]; };

Non_Terminal U1_RA: U1_R { $$ = U1_R; } {U1[U1_R]} {} {} {} ;
Non_Terminal U1_RB: U1_R { $$ = U1_R; } {U1[U1_R]} {} {} {} ;
Non_Terminal U1_RC: U1_R { $$ = U1_R; } {U1[U1_R]} {} {} {} ;

Non_Terminal U2_RA: U2_R { $$ = U2_R; } {U2[U2_R]} {} {} {} ;
```

```
Non_Terminal U2_RB: U2_R { $$ = U2_R; } {U2[U2_R]} {} {} {} ;
Non_Terminal U2_RC: U2_R { $$ = U2_R; } {U2[U2_R]} {} {} {} ;

Non_Terminal U3_RA: U3_R { $$ = U3_R; } {U3[U3_R]} {} {} {} ;
Non_Terminal U3_RB: U3_R { $$ = U3_R; } {U3[U3_R]} {} {} {} ;
Non_Terminal U3_RC: U3_R { $$ = U3_R; } {U3[U3_R]} {} {} {} ;

Non_Terminal SRC:  U1_R { $$ = 0x00 | U1_R; } {U1[U1_R]} {} {} {} |
                   U2_R { $$ = 0x04 | U2_R; } {U2[U2_R]} {} {} {} |
                   U3_R { $$ = 0x08 | U3_R; } {U3[U3_R]} {} {} {} ;

Non_Terminal DEST: U1_R { $$ = 0x00 | U1_R; } {U1[U1_R]} {} {} {} |
                   U2_R { $$ = 0x04 | U2_R; } {U2[U2_R]} {} {} {} |
                   U3_R { $$ = 0x08 | U3_R; } {U3[U3_R]} {} {} {} ;

Non_Terminal DATA: INT { $$ = INT; } {INT} {} {} {};
Non_Terminal LOC:  INT { $$ = INT; } {INT} {} {} {};

#define REG SRC


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

Section Storage

//                                        = entries , bits_per_entry
Instruction Memory INST                   = 0x100 , 0x32
Memory DM                                 =  0x20 ,  0x8
RegFile U1                                =   0x4 ,  0x8
RegFile U2                                =   0x4 ,  0x8
RegFile U3                                =   0x4 ,  0x8
ProgramCounter PC                         =         0x8


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

#define DEFINE_NULL_OP  {} { NULLOP(); } {} {} {}

#define ADDm(x,y)    ADD(x,y,8,"trn")
#define SUBm(x,y)    SUB(x,y,8,"trn")
#define MULm(x,y)    MUL(x,y,8,8,"trn")
#define NOTm(x)      NOT(x,8)
```

```
Section Instruction_Set

Field Controlf:
      Control_NULL DEFINE_NULL_OP
      Control_NOP
         { Control.OP = 0x0 ; }
         { NOP(); }
         {}
         { Cycle=1; Size=1; Stall=0; }
         { Latency=1; }
      Control_bfalse SRC, NAME
         { Control.OP = 0x1 ; Control.SRC = SRC ; }
         { if (SRC == 0)
              { PC <- NAME; }
           else
              { PC <- PC + 1 ; } ;
         }
         {}
         { Cycle=1; Size=1; Stall=0; }
         { Latency=1; Usage=1; }
      Control_jmp NAME
         { Control.OP = 0x2 ; }
         { PC <- NAME; }
         {}
         { Cycle=1; Size=1; Stall=0; }
         { Latency=1; Usage=1; }

Field U1f:
      U1_NULL DEFINE_NULL_OP
      U1_add U1_RA, U1_RB, U1_RC
         { U1.OP = 0x0; U1.RA = U1_RA; U1.RB = U1_RB; U1.RC = U1_RC; }
         { U1_RC <- ADDm(U1_RA,U1_RB); }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
      U1_sub U1_RA, U1_RB, U1_RC
         { U1.OP = 0x1; U1.RA = U1_RA; U1.RB = U1_RB; U1.RC = U1_RC; }
         { U1_RC <- SUBm(U1_RA,U1_RB); }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
      U1_not U1_RA, U1_RC
         { U1.OP = 0x2; U1.RA = U1_RA; U1.RC = U1_RC; }
         { U1_RC <- NOTm(U1_RA); }
         {}
```

```
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
    U1_nop
        { U1.OP = 0x3; }
        { NOP(); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }

Field U2f:
    U2_NULL DEFINE_NULL_OP
    U2_add U2_RA, U2_RB, U2_RC
        { U2.OP = 0x0; U2.RA = U2_RA; U2.RB = U2_RB; U2.RC = U2_RC; }
        { U2_RC <- ADDm(U2_RA,U2_RB); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
    U2_sub U2_RA, U2_RB, U2_RC
        { U2.OP = 0x1; U2.RA = U2_RA; U2.RB = U2_RB; U2.RC = U2_RC; }
        { U2_RC <- SUBm(U2_RA,U2_RB); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
    U2_mul U2_RA, U2_RB, U2_RC
        { U2.OP = 0x2; U2.RA = U2_RA; U2.RB = U2_RB; U2.RC = U2_RC; }
        { U2_RC <- MULm(U2_RA,U2_RB); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
    U2_nop
        { U2.OP = 0x3; }
        { NOP(); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }

Field U3f:
    U3_NULL DEFINE_NULL_OP
    U3_add U3_RA, U3_RB, U3_RC
        { U3.OP = 0x0; U3.RA = U3_RA; U3.RB = U3_RB; U3.RC = U3_RC; }
        { U3_RC <- ADDm(U3_RA,U3_RB); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
```

```
      U3_mul U3_RA, U3_RB, U3_RC
         { U3.OP = 0x1; U3.RA = U3_RA; U3.RB = U3_RB; U3.RC = U3_RC; }
         { U3_RC <- MULm(U3_RA,U3_RB); }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
      U3_sl   U3_RA, U3_RB, U3_RC
         { U3.OP = 0x2; U3.RA = U3_RA; U3.RB = U3_RB; U3.RC = U3_RC; }
         { if (U3_RA < U3_RB)
              { U3_RC <- 1; }
           else
              { U3_RC <- 0; } ;
         }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
      U3_nop
         { U3.OP = 0x3; }
         { NOP(); }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }

// DB1 is used for the data
Field DB1:
      DB1_NULL DEFINE_NULL_OP
      DB1_move SRC, DEST
         { DB1.SRC = SRC; DB1.DEST = DEST; }
         { DEST <- SRC; }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
      DB1_move_im DATA, DEST
         { DB1.SRC = 0x10 | (DATA & 0xF); DB1.DEST = DEST; }
         { DEST <- DATA; }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
      DB1_nop
         { DB1.DEST = 0x1F; }
         { NOP(); }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
```

```
// DB2 is used for the address
Field DB2:
      DB2_NULL DEFINE_NULL_OP
      DB2_move SRC, DEST
         { DB2.SRC = SRC; DB2.DEST = DEST; }
         { DEST <- SRC; }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
      DB2_move_im DATA, DEST
         { DB2.SRC = 0x10 | (DATA & 0xF); DB2.DEST = DEST; }
         { DEST <- DATA; }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
      DB2_nop
         { DB2.DEST = 0x1F; }
         { NOP(); }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }

#define DMdata 0x0C
#define DMaddr 0x0D

Field DMf:
      DM_NULL DEFINE_NULL_OP
      // DB1.SRC gets code for DMdata
      // DB2.DEST gets code for DMaddr
      DM_ld   REG, LOC
         { DB1.SRC = DMdata; DB1.DEST = REG;
           DB2.SRC = LOC; DB2.DEST = DMaddr; }
         { REG <- DM[LOC]; }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
      // DB1.DEST gets code for DMdata
      // DB2.DEST gets code for DMaddr
      DM_st   REG, LOC
         { DB1.SRC = (REG & 0xF); DB1.DEST = DMdata;
           DB2.SRC = LOC; DB2.DEST = DMaddr; }
         { DM[LOC] <- REG; }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
```

```
      DM_st_im  DATA, LOC
         { DB1.SRC = 0x10 | (DATA & 0xF); DB1.DEST = DMdata;
           DB2.SRC = LOC; DB2.DEST = DMaddr; }
         { DM[LOC] <- DATA; }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }

#define IMdata 0x0E
#define IMaddr 0x0F

Field IM:
      IM_NULL DEFINE_NULL_OP
      // DB1.SRC gets code for IMdata
      // DB2.DEST gets code for IMaddr
      IM_ld   REG, LOC
         { DB1.SRC = IMdata; DB1.DEST = REG;
           DB2.SRC = LOC; DB2.DEST = IMaddr; }
         { REG <- INST[LOC]; }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }
      // DB1.DEST gets code for IMdata
      // DB2.DEST gets code for IMaddr
      IM_st   REG, LOC
         { DB1.SRC = REG; DB1.DEST = IMdata;
           DB2.SRC = LOC; DB2.DEST = IMaddr; }
         { INST[LOC] <- REG; }
         {}
         { Cycle = 1; Size = 1; Stall = 0; }
         { Latency = 1; Usage = 1; }


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

Section Constraints

// SRC and DEST cannot be the same on either bus
~( DB*_move U@[1].R*, U@[1].R* )

// Cannot use buses for a move between register files if a memory
// operation is using the buses
~( ((DB*_move* *,*) | (DB*_nop)) & ((DM_* *) | (IM_* *)) )
```

```
// Cannot do both a DM and IM operation
// because they use the same buses
~( (DM_* *) & (IM_* *) )


// Cannot write to same register from two different operations
~( (DB1_move* *,@[1]) & (DB2_move* *,@[1]) )


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

Section Optional
```

# Appendix C

# Example Code Segments

This appendix provides the five example code segments used to generate the results presented in Section 9.1.

## C.1 Example 1

```
int a, b, c;
int d, e, f;

b = 3;
c = 4;
e = 6;
f = 7;

a = b + c;
d = e * f;
a = a - d;
return (a);
```

## C.2 Example 2

```
int a, b, c;
int d, e, f;

a = 2;
b = 3;
c = 4;
d = 5;
e = 6;
f = 7;

a = a + b;
c = c - d;
e = e * f;
a = a - 3;
c = c * e;
f = a + c;
return (f);
```

## C.3 Example 3

```
int i, j;
int a, b;

i = 4;
j = 2;
a = 1;
b = 1;
```

```
a = a*i;
i--;
a = a*i;
b = b*j;
j--;
b = b*j;

a = a + b;
return (a);
```

## C.4 Example 4

```
int i;
int a, b;

i = 1;
a = 1;
b = 4;
a = a + (i*2);
b = b + (i - 3);
i++;
a = a + (i*2);
b = b + (i - 3);

a = a + b;
return (a);
```

## C.5 Example 5

```
int i, j;
int a, b;

i = 6;
j = 3;
a = 1;
b = 1;

a = a*i;
i--;
a = a*i;
i--;
a = a*i;
b = b*j;
j--;
b = b*j;
j--;
b = b*j;

a = a + b;
return (a);
```

# Bibliography


[1] A. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.

[2] A. Aho, S. Johnson, and J. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, 1977.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers–Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1988.

[4] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. *The MIMOLA Language - Version 4.1.* University of Dortmund, September 1994.

[5] D. Berson, R. Gupta, and M. Soffa. URSA: A unified resource allocator for registers and functional units in VLIW architectures. In *Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254. Elsevier Science Publishers B. V., 1993.

[6] D. Bradlee, S. Eggers, and R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, April 1991.

[7] D. Bradlee, R. Henry, and S. Eggers. The Marion system for retargetable instruction scheduling. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–240, June 1991.

[8] J. Bruno and R. Sethi. Code generation for a one-register machine. *Journal of the ACM*, 23(3):502–510, 1976.

[9] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[10] M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno. Hardware-software codesign of embedded systems. *IEEE Micro*, pages 26–36, August 1994.

[11] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[12] D. Engels, G. Hadjiyiannis, S. Hanono, and S. Devadas. Aries: An embedded system design framework. Technical report, Massachusetts Institute of Technology, 1999.

[13] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings of the European Design and Test Conference*, March 1995.

[14] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction sets using nML (extended version). Technical report, Technische Universitat Berlin and IMEC, Berlin (Germany)/Leuven (Belgium), 1995.

[15] J. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[16] C. Fraser, D. Hanson, and T. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters of Programming Languages and Systems*, 1(3):213–226, September 1992.

[17] M. Freericks. The nML machine description formalism. Technical report, TU Berlin, 1993.

[18] M. Garey and D. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[19] J. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing*, pages 442–452, July 1988.

[20] G. Goossens, J. Van Praet, D. Lanneer, W. Geurts, and F. Thoen. Programmable chips in consumer electronics and telecommunications: Architecture and design technology. In G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design*, pages 135–164. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

[21] R. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, pages 29–41, September 1993.

[22] G. Hadjiyiannis. *ISDL: Instruction Set Description Language - Version 1.0.* MIT Laboratory for Computer Science, July 1998. (http://www.caa.lcs.mit.edu/~ghi/PostScript/isdl_manual.ps).

[23] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceedings of the 34th Design Automation Conference*, pages 299–302, June 1997.

[24] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability and architecture exploration. Technical report, Massachusetts Institute of Technology, 1999.

[25] G. Hadjiyiannis, P. Russo, and S. Devadas. A methodology for accurate performance evaluation in architecture exploration. In *Proceedings of the 36th Design Automation Conference*, June 1999.

[26] S. Hanono and S. Devadas. Instruction selection, resource allocation, and scheduling in the Aviv retargetable code generator. In *Proceedings of the 35th Design Automation Conference*, pages 510–515, June 1998.

[27] K. Hargreaves and K. Berry. *Regex - edition 0.12a.* Free Software Foundation, September 1992.

[28] R. Hartmann. Combined scheduling and data routing for programmable ASIC systems. In *Proceedings of the European Conference on Design Automation*, pages 486–490, March 1992.

[29] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[30] A. Kalavade and E. Lee. A hardware-software codesign methodology for DSP applications. *IEEE Design & Test of Computers*, pages 16–28, September 1993.

[31] D. Lanneer, J. Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable code generation for embedded DSP processors. In *Code Generation for Embedded Processors*, pages 85–102. Kluwer Academic Publishers, 1995.

[32] P. Lapsley, J. Bier, A. Shoham, and E. Lee. *DSP Processor Fundamentals - Architectures and Features.* IEEE Press, 1997.

[33] R. Leupers and P. Marwedel. Instruction set extraction from programmable structures. In *Proceedings of the European Design Automation Conference*, pages 156–161, 1994.

[34] R. Leupers and P. Marwedel. Retargetable generation of code selectors from HDL processor models. In *European Design and Test Conference*, 1997.

[35] J. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly & Associates, Inc., 1992.

[36] S. Liao, S. Tjiang, and R. Gupta. An efficient implementation of reactivity for modeling hardware in the Scenic design environment. In *Proceedings of the 34th Design Automation Conference*, pages 70–75, June 1997.

[37] C. Liem, T. May, and P. Paulin. Instruction-set matching and selection for DSP and ASIP code generation. In *Proceedings of the European Design and Test Conference*, pages 31–37, February 1994.

[38] P. Marwedel. Code generation for core processors. In *Proceedings of the 34th Design Automation Conference*, pages 232–237, June 1997.

[39] Mentor Graphics Corporation. *DSP Architect DFL User's and Reference Manual*, 1993.

[40] G. De Micheli. Computer-aided hardware-software codesign. *IEEE Micro*, pages 10–16, August 1994.

[41] Motorola, Inc. *DSP56000/DSP56001 Digital Signal Processor User's Manual*, 1990.

[42] S. Novack and A. Nicolau. An efficient global resource constrained technique for exploiting instruction level parallelism. In *International Conference on Parallel Processing*, August 1992.

[43] S. Novack and A. Nicolau. Mutation scheduling: A unified approach to compiling for fine-grain parallelism. In *Languages and Compilers for Parallel Computing*, 1994.

[44] P. Paulin, M. Cornero, C. Liem, F. Nacabal, C. Donawa, S. Sutarwala, T. May, and C. Valderrama. Trends in embedded system technology: An industrial perspective. In G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design*, pages 311–337. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

[45] P. Paulin, C. Liem, T. May, and S. Sutarwala. CodeSyn: A retargetable code synthesis system. In *Proceedings of the 7th International High-Level Synthesis Workshop*, Spring 1994.

[46] P. Paulin, C. Liem, T. May, and S. Sutarwala. FlexWare: A flexible firmware development environment for embedded systems. In *Code Generation for Embedded*

*Processors*, pages 67–84. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[47] R. Sethi and J. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, 1970.

[48] S. Sjoholm and L. Lindh. *VHDL for Designers.* Prentice Hall, 1997.

[49] SPAM Research Group. *SPAM Compiler User's Manual*, 1.0 edition, 1997.

[50] Stanford Compiler Group. *The SUIF Library*, 1.0 edition, 1994.

[51] A. Sudarsanam. *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors.* PhD thesis, Princeton University, 1998.

[52] S. Sutarwala, P. Paulin, and Y. Kumar. Insulin: An instruction set simulation environment. In *Proceedings of the 1993 Conference on Hardware Description Languages*, pages 355–362, 1993.

[53] D. Thomas, J. Adams, and H. Schmit. A model and methodology for hardware-software codesign. *IEEE Design & Test of Computers*, pages 6–15, September 1993.

[54] D. Thomas and P. Moorby. *The Verilog Hardware Description Language - Second Edition.* Kluwer Academic Publishers, 1995.

[55] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity. The MIT Press/Elsevier, 1990.

[56] J. Vanhoof, K. Van Rompaey, I. Bolsens, G. Goosens, and H. De Man. *High-Level Synthesis for Real-Time Digital Signal Processing.* Kluwer Academic Publishers, 1993.

[57] LSI Logic Website. G12 CMOS technology. http://www.lsilogic.com/products.

[58] Synopsys Website. Synopsys VHDL Compiler(TM) and Synopsys HDL Compiler for Verilog. http://www.synopsys.com/products/logic/hdl_comp_cs.html.

[59] B. Wess. Code generation based on trellis diagrams. In *Code Generation for Embedded Processors*, pages 188–202. Kluwer Academic Publishers, 1995.

[60] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J.Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. Computer Systems Laboratory, Stanford University. Available from http://suif.stanford.edu.

[61] T. Wilson, G. Grewal, S. Henshall, and D. Banerji. An ILP-based approach to code generation. In *Code Generation for Embedded Processors*, pages 103–118. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[62] V. Zivojnovic, S. Pees, and H. Meyr. LISA – Machine description language and generic machine model for HW/SW co-design. In *Proceedings of 1996 IEEE Workshop on VLSI Signal Processing*, October 1996.