An Architecture Synthesis System for Embedded Processors

by

George Ioannou Hadjiyiannis

S.M. (EECS) Massachusetts Institute of Technology (Sep. 1995)
B.S. (EECS) Massachusetts Institute of Technology (Sep. 1993)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

Author .....
Department of Electrical Engineering and Computer Science
May 3, 2000

Certified by .....
Srinivas Devadas
Professor
Thesis Supervisor

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

An Architecture Synthesis System for Embedded Processors

by

George Ioannou Hadjiyiannis

Submitted to the Department of Electrical Engineering and Computer Science
on May 3, 2000, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Design requirements for embedded systems call for architectures with small size, low power consumption and low cost. These requirements can be met by designing custom architectures for every single application. However, the commercial viability of embedded systems calls for short design cycles. These requirements are conflicting: custom architectures take a long time and substantial effort to produce, because of the need to manually generate design evaluation tools, such as simulators and compilers, for each architecture candidate. This conflict can be eliminated by providing a system capable of generating all design evaluation tools for a given candidate architecture.

This thesis presents two components of the ARIES environment for architecture synthesis: the machine description language ISDL and the GENSIM simulator generator system. We also briefly describe the HGEN hardware model generator. In the ARIES system, candidate architectures are described in the ISDL language. From this machine description, component tools can automatically generate design evaluation tools, namely an assembler, Instruction Level Simulator, and disassembler. These tools can be used to evaluate each candidate architecture and make improvements. The whole process can be used to implement an architecture exploration loop which provides the benefits of custom architectures while maintaining short design cycles. Preliminary results also show that it is possible to generate retargetable compilers and hardware models from the same machine description.

ISDL is a flexible machine description language that supports a wide range of architectures and features with special emphasis on VLIW architectures. It provides sufficient information to generate all the tools from a *single* machine description, and provides constraints which make the machine descriptions concise and intuitive. The GENSIM simulator generator is a tool that can automatically produce fast (4.5 Mops), cycle-accurate, bit-true Instruction Level Simulators from ISDL descriptions in a short time. We present experimental results which show that ISDL is flexible enough to describe a wide range of architectures implementing a broad range of architectural features. We also present experimental results demonstrating the capabilities of the GENSIM system and the generated simulators. Finally we present results that demonstrate the feasibility of architecture exploration based on automatically

generated design evaluation tools.

Thesis Supervisor: Srinivas Devadas
Title: Professor

Acknowledgments

During the five years that I have worked on the ARIES project and the research described in this document, a number of people stepped forth to provide their contributions to either my work or my life and support this task directly or indirectly. These are the people that I wish to thank for supporting me in my endeavors, both in my academic and my personal life.

First of all I would like to thank my Thesis Supervisor, Prof. Srinivas Devadas. Prof. Devadas undertook the demanding role of attempting to guide me through my work, providing both academic help with the nature of the problems I encountered, as well as help in the direction and the manner in which I performed my work. He has always made himself available, and always did his best to help, even when - on occasion - I made it hard for him to do so. He has always shown the enthusiasm of a child and the wisdom of experience in everything that we did together. I must say I thoroughly enjoyed all the time I spent working with him. I was also fortunate enough to have enjoyed a relationship of friendship with Prof. Devadas which made it even easier and more enjoyable to work with him. He takes a personal interest in the lives of his students and I hope that he realizes that all his students appreciate it.

I would also like to thank my colleagues, Silvina Hanono, and Daniel Engels. Silvina's help in developing ISDL was invaluable. I truly believe that without it the project would have suffered dramatically. She also provided help with almost all other aspects of my work, from co-authoring and proof-reading papers, to helping with coding problems. Dan has always provided a sounding board for all my ideas and provided endless help with papers and algorithms. Both Silvina and Dan where kind enough to let me know when I was heading down the wrong path and to persist until I saw it, thus saving me an incalculable amount of wasted effort.

I would also like to thank a number of friends who supported me on a more personal level: Mike Ehrlich, Sabine Bendiek, Jennifer Trickett, Radu Rugina and Maria Marinescu. Their support has always made it possible for me to maintain my motivation and their judgment has always forced me to do things the right way.

Last, but definitely not least, I would like to thank my parents for setting me on this path at a very young age and never letting me stray from it. While they have never judged the manner in which I conducted myself with regard to my studies and my work, they instilled in me the courage to judge myself. They taught me the value of education and left it up to me to seek it for my own benefit. And for years they struggled to provide me with any resource and any chance I needed to get to where I am today.

Thanks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In recent years there has been an tremendous growth in the use of consumer electronic products such as personal digital assistants, multi-media systems, and cellular phones. The digital circuits that control and process data in these devices are called *embedded systems*. If these systems contain programmable processors, these are referred to as *embedded processors*.

Architecture design for embedded processors is substantially different from architecture design for general-purpose processors. The latter strives to create processors of maximum possible performance for a given cost. Additionally, because of their general nature, the domains in which these processors are applied often overlap substantially, and as a result not many designs exist and most of them share a lot of features. Finally, the applications that these processors will be called upon to run are not known at design time, and therefore the processors cannot be customized to any specific application. By contrast, the world of embedded processors contains a much richer set of designs and covers a substantially larger portion of the architecture design space. Furthermore, the applications that embedded processors are called upon to execute are known at design time so theoretically each processor could be customized for the particular application it will be used for. This results in complex architectures containing a variety of custom architectural features.

This document is concerned with the task of producing architectures for embedded processors.

1.1.1 Hardware/Software Co-Design

To reduce the cost, size, and power consumption of embedded systems, manufacturers often integrate an *entire system* on a single integrated circuit (IC) [11, 5]. In addition, the pressure for short design cycles forces designers to implement an increasing amount of functionality in *software* relative to hardware. The software implementations can accommodate late changes in the requirements or design, thus reducing the length of the design cycle. Figure 1-1 illustrates a typical embedded system, consisting of a digital signal processor (DSP) core or an application-specific instruction-set

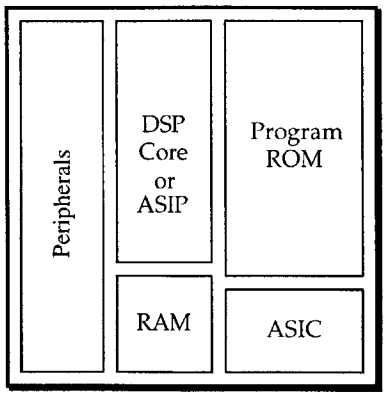

Figure 1-1: **A System-on-a-Chip**

processor (ASIP), a program ROM, RAM, application-specific circuitry (ASIC), and peripheral circuitry.

Since the embedded processor, the program ROM and the custom ASIC are implemented on the same chip, the hardware and software components of embedded systems are strongly coupled. Small changes in the hardware (software) can affect the software (hardware) dramatically. In order to address the interaction between the hardware and software, a *hardware-software co-design* methodology (e.g., [12]) is required.

A simplified view of a generic hardware-software co-design methodology is shown in Figure 1-2. In this design methodology, designers partition the system functionality (i.e., input application) into hardware and software components. Additionally, a target processor is chosen from existing processor designs, or an ASIP is designed to execute the software. The hardware, software, and ASIP are implemented, and the resulting system is evaluated using a hardware-software co-simulator. The partitioning and processor design are repeated until an acceptable system is developed. This document focuses on the software synthesis portion of hardware-software co-design. Software synthesis involves designing a target processor to execute the software component of the embedded system, and compiling the software component of the application for the target processor.

1.1.2 Embedded Processor Requirements

Because most embedded processors are used in consumer electronics, their commercial success requires low-cost architectures. Most applications (e.g., cellular phones) also have stringent requirements on size and power consumption for the sake of portability. Additionally, embedded processors are designed with a particular performance target in mind. In most cases, creating a processor that can provide more that the target performance does not provide any advantage. Finally, the commercial success of such products requires short time-to-market and therefore short design cycles.

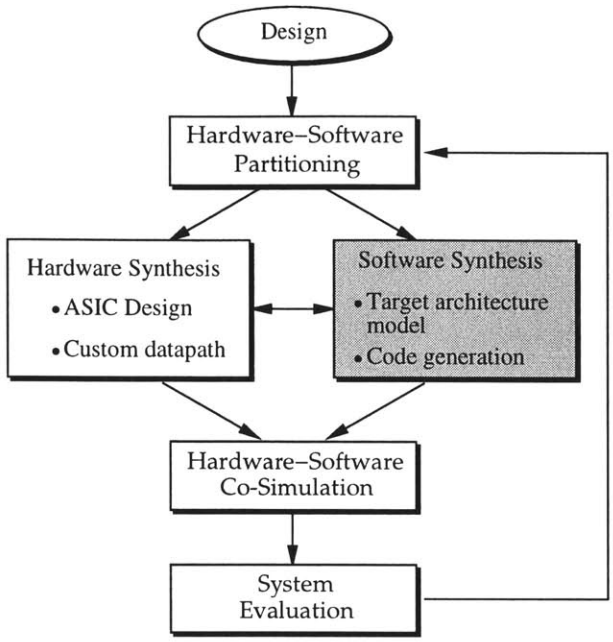

Figure 1-2: **A Generic Hardware/Software Co-design Methodology**

In order to produce embedded systems which satisfy, cost, power consumption and size requirements, it is desirable to customize the architecture of embedded processors to the application at hand. However, generating a custom architecture often involves a substantial amount of time and effort. This is because a number of candidate architectures must be designed, evaluated, and successively refined before a final architecture is selected. For every candidate architecture, a set of design evaluation tools (such as compiler, assembler, simulator, and hardware model) must be created, resulting in substantial development effort and long design cycles. This results in conflicting requirements.

1.2 Architecture Synthesis for Embedded Systems

One way of performing architecture selection is to explore the architectural design space until a suitable design is found. To implement such a search, an initial architecture is created and evaluated in the context of the application at hand. Possible improvements are located and implemented resulting in a new architecture. This architecture is once again evaluated and the process repeated until no further improvement can be obtained. This process is called *architecture exploration by iterative improvement.*

In order for architecture exploration to cover a large portion of the design space, the design evaluation tools (i.e., assembler, disassembler, compiler, simulator, and hardware model) must be either automatically retargeted or generated for each candidate architecture.

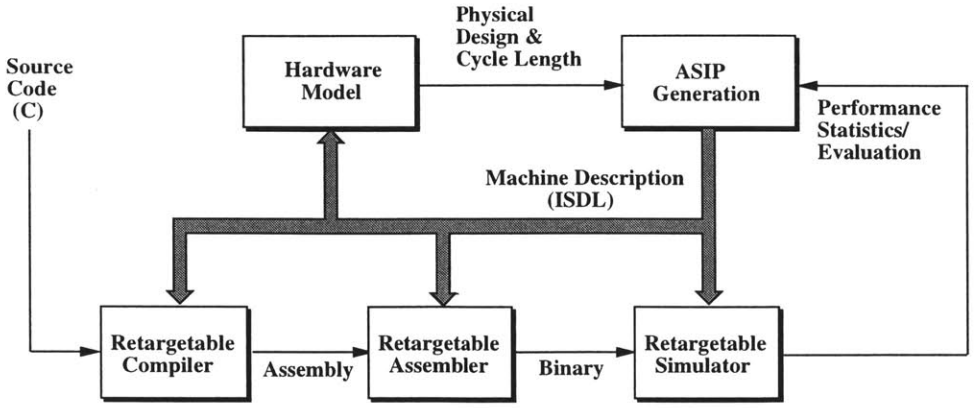

Figure 1-3: **Architecture Exploration by Iterative Improvement**

1.2.1 Overview of Architecture Exploration by Iterative Improvement

Figure 1-3 shows a fully automated system based on architecture exploration. In this approach, the application code is analyzed, and an initial architecture is generated and described using a *machine description language*. This machine description is then fed into a retargetable compiler along with the application source code. The retargetable compiler translates the source code into assembly code for the target processor, optimized for speed and code size. The same machine description is also passed to a retargetable assembler that converts the assembly code into binary code for the target processor. Next, a retargetable simulator receives the machine description and the binary code as inputs and executes the code. The simulator generates a set of measurements that can be used to evaluate the architecture and identify possible improvements. The machine description is also used to generate a hardware model of the processor that provides the physical costs of the design (i.e., die size, cycle length, and power consumption). If the performance of the processor does not satisfy the design specifications, or if the cost, size, or power consumption can be reduced without sacrificing performance, then the appropriate changes are made to the processor architecture and a new machine description is generated. The entire design process is repeated using the new machine description until no further improvements can be made. Once the final architecture has been determined, the hardware model can be used to generate an implementation of the architecture.

1.2.2 Architecture Exploration for Embedded Systems

Section 1.2.1 describes a completely automated architecture exploration system. However, for the domain of embedded DSP applications not all the tasks have to be automated. In embedded DSP applications only a small portion of the code is critical to performance, and this is typically implemented as hand-coded assembly libraries. These libraries contain code commonly used in DSP applications such as Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters, Fast Fourier Trans-

form (FFT) functions, and so on. Most of the time taken for DSP applications is consumed by such functions for almost all DSP applications.

Given that the set of tasks that require good performance is small and well-known it is possible to implement these libraries by hand-coded assembly for every iteration of the loop. Note that this does not mean that an architecture optimized for these libraries is the optimal architecture for any application that uses them. The code that will form the bottleneck in performance is indeed in these libraries, however, the code that will incur the most significant code-size cost is the remainder[1]. Additionally, size and power consumption still need to be optimized, and the tradeoffs may be such that the optimal point is not the same for every application. Consider for example, an application where an FIR is used to filter a low quality audio channel. The typical sample rate for such applications is 8KHz, thus performance may not be a big concern and instead code size and power consumption might become the dominant factors in architecture selection. If instead the FIR filter is to be used in a high-speed military radar system where sample rates of MHz are not uncommon, performance will be the primary concern in architecture selection since it is the hardest constraint to meet.

Given that for embedded applications small amounts of highly optimized code are typical, it is possible to perform architecture exploration without having some of the design evaluation tools. For example, a compiler is not necessary since the amount of code that has to be optimized is small. It is possible to hand-code assembly for each iteration of the architecture exploration for the particular application. Additionally, a hardware model may not be necessary if the designer is familiar with the physical properties of various architectural features. Finally, the fact that most of the code is not performance critical and the code that is performance critical is small decouples issues of code-size from performance allowing a manual architecture exploration loop supported simply by an Instruction Level Simulator and an assembler. In this process, a designer generates an initial architecture based on intuition and experience with typical applications in the DSP domain. She then describes the architecture in a machine description language and uses this description to generate an assembler and an instruction level simulator. She codes in assembly the time-critical portions of the application and simulates them to get performance measurements. She modifies the architecture, generates another description, and repeats the loop until she is satisfied that no more improvements can be made. Then, she codes the rest of the application in hand-coded assembly and repeats the process trying to minimize code size without degrading performance.

1.2.3 Requirements for Architecture Exploration

For architecture exploration to be successful, any implementation of it must provide the following:

- A system that can *automatically* produce design evaluation tools given a description of a target architecture. For a fully automated process *all* the design

[1]This is a result of the well-known 80/20 rule: 80% of the time is spent on 20% of the code. The corollary to the rule is that 80% of the code size is incurred by code that is not performance critical.

evaluation tools must be produced automatically. If a single tool has to be developed manually, then the effort and time required for a single iteration of the architecture exploration algorithm will be dominated by the effort and time to develop that tool. The required evaluation tools are: an assembler, a disassembler, an optimizing compiler, an Instruction Level Simulator (ILS), and a hardware model. For a partly automated process, such as the one described in Section 1.2.2, an assembler and instruction level simulator are necessary. A hardware model would be beneficial if provided but not critical.

- A machine description language which can describe as wide a variety of architectures as possible at a fine granularity. This language should provide features that support the generation of all required design evaluation tools from a *single* machine description. This avoids possible problems of consistency and the effort of translating machine descriptions between different languages.

- A way of extracting useful information from the design evaluation tools to identify possible improvements to the architecture.

1.2.4 Overview of the ARIES System

The work presented in this document was performed in the context of the ARIES system. The ARIES system is our planned implementation of a Hardware/Software Co-Design system implementing architecture exploration by iterative improvement. Figure 1-4 shows the framework of the system.

The main components of the ARIES system are:

- A top-level application analysis and partitioning tool that performs the partitioning between hardware and software.

- An ASIC generation tool which implements the hardware component.

- An ASIP tool which implements and evaluates candidate architectures for the embedded processor.

- The AVIV code generator tool which generates retargetable compilers and assemblers for a candidate architecture given a machine description.

- The ISDL machine description language which describes the candidate architecture to the necessary components.

The ASIP generation tool, AVIV code generator, and ISDL machine description language implement the architecture exploration portion of ARIES.

Figure 1-5 shows the methodology of the ARIES system. A description of the application in a high-level language is provided as a system specification. This forms the input to the analysis and partitioning tool which partitions the specification into hardware and software components. The hardware component is passed to an ASIC

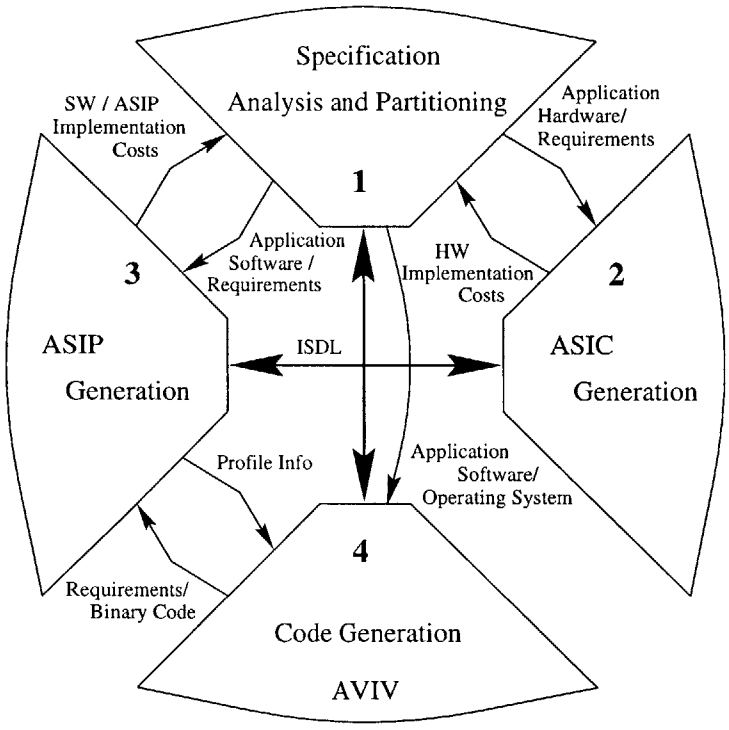

Figure 1-4: **The ARIES Framework**

Synthesis tool which produces an HDL model in synthesizable Verilog. The partitioning tool also produces an interface specification which is passed to the Interface Synthesis tool. This tool implements the interface in HDL and also creates software stubs which are used by the Operating System Generation tool to communicate with the ASIC. Finally, the partitioning tool passes the software component to the Code Synthesis tool which produces specifications for the operating system and source code for the retargetable compiler. Finally, the partitioning tool provides a set of area, power, and performance requirements and constraints to the ASIP generation tool, which produces an initial architecture and describes it in ISDL. This machine description and the source code are used by the AVIV retargetable compiler to produce an implementation of the software component on the target architecture. It is also used to create an Instruction Level Simulator and a hardware model. The ILS and hardware model are used to evaluate the candidate architecture and recommend improvements resulting in a new architecture description. The loop is repeated until no further improvements can be made. The final hardware model is linked with the ASIC and interface models and the complete system is evaluated. Improvements are made by re-partitioning and re-iterating through the whole procedure.

This document focuses on the ASIP generation process and in particular, the ISDL machine description language[19, 15, 17, 14], the ILS generator (called GENSIM)[16, 18] and the hardware model generator (called HGEN)[16, 18]. Most of the components of the ARIES system have not been fully implemented yet. In particular, the

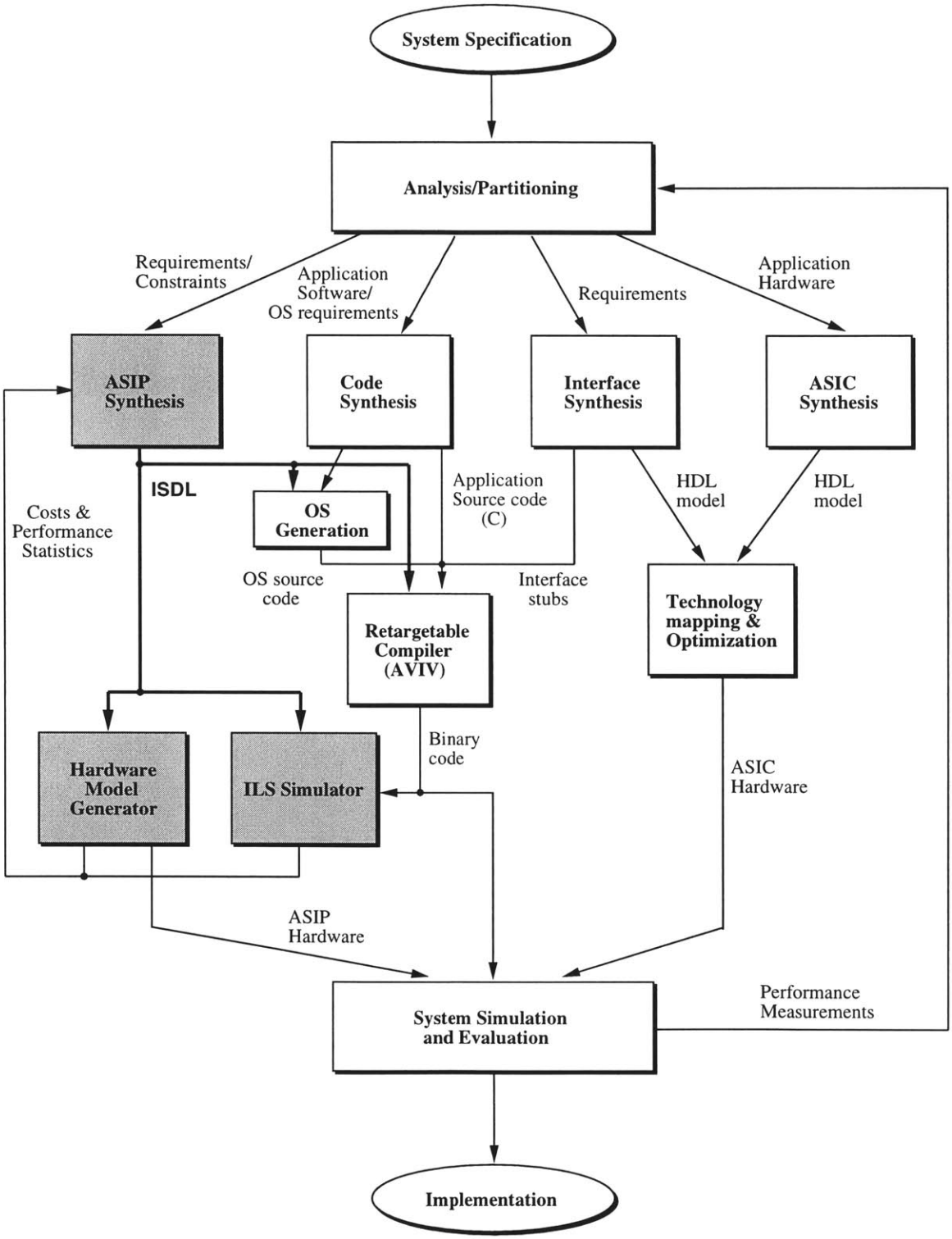

Figure 1-5: **The ARIES Methodology**

retargetable compiler is the most challenging task. AVIV was an attempt to provide some basic functionality for the retargetable compiler and investigate the issues involved in the design and implementation of such a compiler. While it has succeeded in those goals, it is still not a full-fledged compiler and cannot be used to automate the architecture exploration loop. The details of the AVIV retargetable compiler can be found in [20, 21]. Similarly, preliminary implementations of the partitioning tool exist, which where used to investigate the issues involved and the performance of some proposed algorithms. Again, however, these are the result of an incomplete implementation and cannot be used within an automated system. The details of the partitioning algorithm can be found in [6]. The results and conclusions presented in this document are therefore based on the partially automated approach of Section 1.2.2. Note, however, that the current implementation of ISDL and the simulator generator are complete and can be used within a fully automated system once full implementations of the remaining tools are provided.

1.3 Contributions

The work described in this document is based on the following thesis:

- Architecture exploration by iterative improvement is an effective way of obtaining customized architectures for embedded systems.
- It is possible to obtain the advantages of a custom architecture while reducing design cycles by automating the generation of the design evaluation tools.
- In order to support architecture exploration and automatically generated evaluation tools a flexible machine description language is necessary. This language should cover as much of the architecture design space as possible and support the automatic generation of at least an assembler, disassembler, and Instruction Level Simulator. Availability of tools that can generate a retargetable compiler and a hardware model will enable a fully automated architecture exploration loop. ISDL provides all the features necessary to describe a wide variety of architectures and architectural features and contains enough information to generate *all* of the above tools.
- It is possible to generate fast, cycle-accurate and bit-true simulators, and efficient hardware models form ISDL. These simulators and hardware models are effective tools in evaluating candidate architectures.

The main goal of this document is to present the ISDL machine description language and the GENSIM simulator and HGEN hardware model generation systems, and provide experimental results supporting the above thesis.

1.3.1 ISDL

We present the ISDL machine description language for architecture exploration. ISDL has proven to be well-suited to the task of supporting architecture exploration. It can describe a wide range of architectures including VLIW, RISC, CISC, DSP and micro-controller processors and supports features as varied as register aliases, complex instructions and predicated execution. It supports the automatic generation of *all* design evaluation tools from a single machine description. It is easy to read, write and modify ISDL descriptions manually, and easy to produce them automatically. By providing constraints, ISDL allows for concise and intuitive descriptions of real-world architectures. ISDL modifications resulting in improvements can be performed in about an hour by an engineer skilled in the use of the language.

1.3.2 Simulator and Hardware Model Generation

We also present the GENSIM simulator generator and HGEN hardware model generator. The GENSIM simulator generator is an implementation of an ILS generator based on ISDL. It processes machine descriptions in a reasonable amount of time, and produces simulators which are cycle-accurate to the instruction set level, and bit-true. The generated simulators are also fast, and provide full debugging support. Their ability to produce execution address traces means that dynamic counts for instructions can be obtained which can be used to derive utilization statistics and thus point out possible improvements to the architecture. The GENSIM simulator generator supports a wide variety of architectures and architectural features. The HGEN hardware generation model is a sister-tool based on the same algorithms implemented in GENSIM. It produces efficient hardware models of a candidate architecture given a machine description in ISDL. These models (implemented in synthesizable Verilog) can then be used to obtain the physical costs and parameters (such as cycle-length, die size, and power consumption) of the architecture.

Chapter 2

Instruction Set Description Language (ISDL)

ISDL (Instruction Set Description Language) is a machine description language specifically designed to support retargetable tools for architecture exploration and development. In particular, it is designed to allow the automatic generation of an assembler, disassembler, code generator, instruction level simulator, and hardware model from a single description of the architecture. At the same time, it is designed to support architecture descriptions that are either generated automatically by another tool, or generated and/or modified manually by an engineer. Figure 2-1 shows how ISDL descriptions can be used to generate a set of tools to support a design environment.

ISDL supports the description of a wide variety of architectures. Its main focus is on VLIW architectures; however, it also supports standard microcontrollers, and other unifunctional architectures. Unifunctional architectures can be considered degenerate cases of VLIW architectures. ISDL supports the description of multiple functional units, different interconnect 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 handle 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 many commercial architectures also require such constraints (e.g., the Motorola 56000 DSP cannot perform a `REP`etition of a `DO` loop).

2.1 Requirements and Features

The machine description language is a critical component in the architecture exploration design flow and should be capable of performing the following functions:

- Specify a wide variety of architectures. In particular, it should support VLIW (Very Long Instruction Word) architectures because they are very efficient for

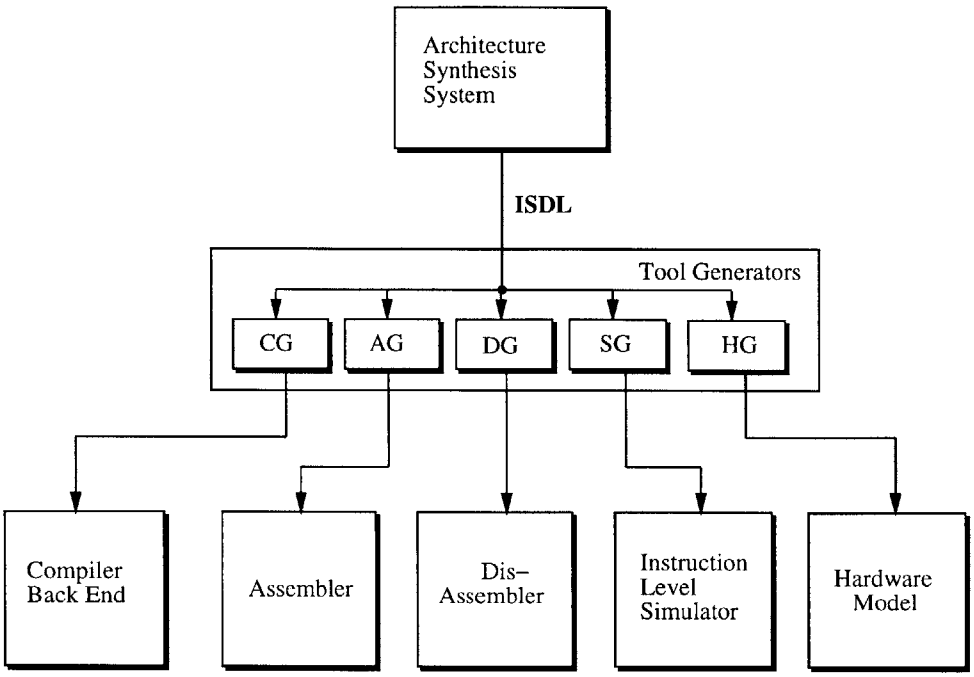

Figure 2-1: ISDL **and Generated Tools**

custom applications, and because more traditional architectures can be treated as degenerate cases of VLIW architectures.

- Explicitly support constraints that define valid instructions.
- Be easily understandable and modifiable by a designer.
- Support automatically retargetable code generation.
- Support the automatic generation of an assembler, disassembler, compiler, Instruction Level Simulator and hardware model.
- Provide adequate information to allow for code optimizations.
- Decouple the description of an Instruction Set Architecture (ISA) from a particular implementation of the ISA.

In order to ensure that all of the design evaluation tools receive a consistent machine description, it is desirable that a *single* machine description be used to retarget or generate *all* of the design evaluation tools. In addition, the effort of generating multiple machine descriptions for the different tools can be avoided by using a single machine description.

Various proposed machine description languages lack support for one or more of the above features (see Section 2.5 for a review). In contrast, ISDL provides all of the above features. We have written complete ISDL descriptions for ASIPs and

commercial DSP cores including a powerful seven-way VLIW ASIP and the Motorola 56000 DSP.

2.1.1 Specifying a Wide Variety Of Architectures

It is important that the range of architectures that are supported by the design environment is not limited by the abilities of ISDL, but rather by the abilities of the various tools that handle ISDL descriptions and those that are generated from ISDL descriptions. In order to achieve this, ISDL attempts to encompass as wide a variety of architectures as possible. In particular, ISDL supports Very Long Instruction Word (VLIW) architectures; these are not explicitly supported by most other machine description languages, and are a superset of more traditional architectures. VLIW architectures have more than one functional unit, and these functional units can be used in parallel. This parallelism is reflected in the instruction set, where instructions are groups of operations that can be performed in parallel[1]. One can consider the more traditional architectures that only have one unit active at any given time in their instruction set, (from now on called *unifunctional* architectures), to be degenerate cases of VLIW architectures.

VLIW architectures are very important for two reasons:

- By amortizing control overhead over a number of functional units they actually result in more efficient use of silicon area.
- Given that the set of VLIW architectures can be considered as a superset of the set of unifunctional architectures, then if we can describe VLIW architectures, we can automatically describe unifunctional architectures.

ISDL has specific features to support the efficient description of VLIW architectures. It can also gracefully handle unifunctional architectures. These two classes of architectures cover traditional micro-controllers, CISC and RISC processors, vector processors, most DSP cores, and a large number of Application Specific Instruction-set Processors (ASIPs).

2.1.2 ISDL Descriptions as a Programmer's Manual

In order to make the task of understanding, writing and modifying ISDL descriptions easier, ISDL was designed to look like the conventional Programmer's Manual that accompanies most commercial processors and DSPs. It turns out that this also makes it easier to generate the support tools. In particular, ISDL models an architecture by describing the user visible state and then listing every operation that can affect this state. Thus, ISDL is a behavioral language (rather than a structural one). It does, however, contain enough structural information to allow for the complete description

[1]This is in contrast to parallelism which is not apparent in the instruction set (such as the parallelism in super-scalar architectures).

of an architecture (although it tends to disguise such structural information in a behavioral-like fashion).

The 6 sections of an ISDL description correspond closely with the sections found in most Programmer's Manuals for commercial architectures. The Instruction Word Format describes how long the instruction word is, and how it can be broken down into subfields. Global Definitions correspond to the various data types and tables of common sub-expressions (such as addressing modes) common in most processors and DSPs. They are the main source of abstraction in ISDL. The Storage Resources section explicitly lists all the visible state of the processor[2]. The Instruction Set section lists all possible operations that the architecture provides, grouped in *fields*. The effect of each operation on visible state, the assembly representation of the operation, the binary representation of the operation, the timing characteristics of each operation, and various costs are all listed on a per-operation basis. The Constraints section describes various restrictions as to how these operations may be assembled into instruction words and programs. This corresponds to the restrictions commonly listed in the Programmer's Manual.

2.2 Syntax and Semantics of ISDL

An ISDL description consists of six sections:

1. Instruction Word Format
2. Global Definitions
3. Storage Resources
4. Instruction Set
5. Constraints
6. Optional Architectural Details

ISDL also supports `cpp` style macros which allow common patterns to be easily reused.

Each of the sections listed above is described below, and a detailed example of each is provided in Section 2.3. There is also a BNF description of ISDL in Appendix B.

2.2.1 Instruction Word Format

The *Instruction Word Format* section defines the hardware instruction word. The instruction word is divided into one or more fields each containing one or more subfields. This section specifies the field and subfield division of the hardware instruction word. In particular, it specifies the ordering of the fields and subfields as well as the

[2]Visible state includes control registers and memory-mapped I/O.

bitwidth of each subfield. The instruction word is assembled by concatenating all of the subfields in the order specified, beginning with the most significant bit.

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

2.2.2 Global Definitions

The second section of an ISDL description contains a list of definitions used in the later sections. These definitions form the main abstraction mechanism in ISDL. There are three types of global definitions in ISDL: *Tokens*, *Non-terminals*, and *Split functions*.

Tokens are a symbolic representation of the primitives in the assembly syntax of the target processor. Tokens are used to represent entities such as register names, memory bank names, and immediate constants. In addition, tokens can be used to group syntactically related entities such as the names of registers within a register file. In order to be able 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 to 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 any. Non-terminal and operation definitions can refer to tokens by their name.

In addition to the tokens explicitly listed in an ISDL machine description, there are additional tokens that are automatically defined (e.g., operation names). Furthermore, ISDL also includes predefined tokens for integers, hexadecimals, floating point numbers, single characters, and labels (symbolic names that represent instruction memory locations).

Non-terminals are used to abstract common patterns in operation definitions (e.g., addressing modes). For example, consider a `Move` operation that moves data across a bus that has seven units attached to it:

```
Move SRC DEST
```

where `SRC` and `DEST` can each be one of seven different options. Without non-terminals, 49 rules are required to describe all possible syntax combinations. However, the source and destination could be factored out into non-terminals. This factorization would result in only three rules: one for the operation, and one each for the `SRC` and `DEST` non-terminals.

Non-terminal definitions consist of a name followed by a list of options that the non-terminal can represent. Operation and other non-terminal definitions can refer to any non-terminal using its name[3]. Non-terminal options consist of the following components:

- The *assembly syntax* of the option which can include references to tokens or other non-terminals.

[3]Non-terminals can be nested to an infinite level.

- A *return value* that identifies the option. This value can be used in operation bit-field assignments (Section 2.2.4). The return value of tokens and non-terminals referred to in the assembly syntax of the option can be used to define the return value for the option.

- An *RTL action clause* that describes the RTL equivalent of the option when the non-terminal is used in the RTL action portion of an operation definition (Section 2.2.4). 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 of the option.

- An *RTL side effects clause* which is similar to the RTL action clause but refers to side effects.

- A *cost modifier clause* that contains a set of expressions describing the effect of the non-terminal option on the operation costs. The cost expressions can include the return value of the tokens, or the cost modifier clause of the non-terminals, referred to in the assembly syntax of the option.

- A *timing modifier clause* that contains a set of expressions describing the effect of the non-terminal option on the timing parameters of the operation. The timing expressions can include the return value of the tokens, or the timing modifier clause of the non-terminals, referred to in the assembly syntax of the option.

Split functions define how long constants (e.g., a long memory address, or immediate data) can be split among multiple subfields of the binary instruction word. A split function definition consists of the function name followed by a list of subfields.

2.2.3 Storage Resources

The *Storage* section lists all storage resources visible to the programmer. It lists the names and sizes of the memories, register files, general purpose registers, 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). Multiple units of each type may be defined. The instruction memory and program counter must be explicitly identified.

ISDL recognizes the following types of storage units:

- `Memory` - Used to declare both data memories and the instruction memory.

- `RegFile` - Used to declare register files.

- `Register` - Used to declare single registers used for data computation.

- `CRegister` - Used to declare control and status registers. These registers have side effects when written (e.g., may cause a change in processor mode) and do not necessarily return the last value written to them when read (e.g., status of peripherals).

- `Stack(SP)` - Used to declare hardware stacks. `SP` represents the name of the stack pointer which must be a single register that is also defined in the storage section.

- `MMIO` - Used to declare memory mapped I/O ports. These ports may have side effects when written and do not necessarily return the last value written to them when read.

- `ProgramCounter` - Used to explicitly declare the Program Counter.

2.2.4 Instruction Set

The *Instruction Set* section lists all of the operations available on the target processor. It groups the operations into mutually exclusive sets called *Fields*. Each field roughly corresponds to the operations that can be performed on a single functional unit. A VLIW instruction consists of a group of operations, one from each field.

The instruction set section consists of a list of field definitions. Each field definition consists of a number of operation definitions. Each operation definition consists of the following elements:

- **Syntax:** This declares the assembly syntax of the operation. It consists of an operation name followed by a list of parameters, each of which is the name of a token or a non-terminal.

- **Bitfield Assignments:** The bitfield assignments define the assembly function for the operation. The assignments are a set of statements that assign the appropriate binary values to the subfields defined in the instruction word format section. The bitfield assignment statements may make use of the return values of the tokens and non-terminals in the operation's parameter list.

- **RTL Action:** This describes the effect of the operation on the processor state using 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 same RTL language as the RTL action description. It may make use of the return value of tokens and the RTL side effects clause of non-terminals in the parameter list.

- **Costs:** Multiple costs are permitted including operation execution time, code size, costs due to resource conflicts, etc. ISDL predefines three cost parameters:

1. `Cycle` declares the number of cycles that the operation requires to execute on the hardware.
2. `Size` declares the number of instruction words needed to represent the operation.
3. `Stall` declares the number of stall cycles that will be inserted if the next instruction attempts to use the results of the operation.

The cost parameters are defined as a set of arithmetic expressions resulting in numerical values. The arithmetic expressions can include arithmetic operators (i.e., `+`, `-`, `*`, `/`, and `%`) and relational operators (e.g., `==`, `<`, `>`). They may also use the return value of tokens and the cost modifiers of the non-terminals in the operation's parameter list. Furthermore, the arithmetic expression may use the cost clauses of operations in other fields and the values of storage references.

- **Timing:** The timing parameters describe when the various effects of the operation take place. ISDL predefines two timing parameters:

  1. `Latency` specifies the number of instructions (including the one containing the current operation) that must be fetched before the results of the current operation become available.
  2. `Usage` specifies the number of instructions (including the one containing the current operation) that must be fetched before the corresponding functional unit becomes available again.

  The timing clauses are complex arithmetic expressions resulting in numerical values. The timing arithmetic expressions obey the same syntax and semantics as the cost clauses.

A set of examples can better illustrate the use of the costs and timing parameters to describe the effect of pipelines:

- **Case 1:** Consider a simple architecture with no pipelining where each instruction completes before the next one is fetched and requires three cycles to do so. Thus, `Cycle` = 3 and `Stall` = 0 because it is not possible for any operation to stall (no pipeline). `Latency` = 1 because the next instruction can use the results of the current operation.

- **Case 2:** Consider an architecture with a simple four-stage pipeline with no protection (no bypass logic or stall capability). The architecture can execute one instruction per clock cycle for all instructions. Thus, `Cycle` = 1 because each instruction effectively takes one clock cycle. `Stall` = 0 since stalls are not necessary in an unprotected pipeline. `Latency` = 4 because the current instruction plus three additional instructions must be fetched before another instruction can use the results of the current operation.

- **Case 3:** Consider an architecture with a four-stage pipeline with full protection (bypass logic). The architecture can execute one instruction per clock cycle for all instructions. Thus, `Cycle` = 1 since each instruction effectively takes one clock cycle. `Stall` = 0 since stalls are not necessary because of the bypass logic. `Latency` = 1 because the bypass logic guarantees that the results will be available to any subsequent instruction.

- **Case 4:** Consider an architecture with a four-stage pipeline protected by bypass logic for all operations except loads. Loads are protected by a single stall cycle. For the load instruction `Stall` = 1 because one cycle will be inserted if the next instruction attempts to use the results of the current operation. `Cycle` = 1 since all instructions effectively take one instruction, except for the case of a load with a stall which was already taken into account. Finally, `Latency` = 1 since all operations (including loads) are protected, thus the next instruction will be able to use the results of the current operation.

- **Case 5:** Finally, consider an architecture with a four-stage pipeline and a branch-if-zero instruction which flushes two stages of the pipeline if the branch is taken. The effect of the pipeline flush would appear as an additional two cycles added to the `Cycle` cost of the operation if the branch is taken. Therefore, `Cycle` = $(R == 0) * 2 + 1$, `Latency` = 1 since the next operation will be fetched from the target of the branch, and `Stall` = 0. The `Cycle` cost is 1 if register R is not zero (i.e., the branch is not taken) and 3 if R is zero (i.e., the branch is taken).

2.2.5 Constraints

The Instruction Set section describes a number of fields whose operations can generally be executed in parallel. However, there are certain combinations of operations that may not be executable by the hardware. The *Constraints* section is used to make these combinations visible to the instruction set.

Constraints are described as a set of Boolean rules. The syntax for the constraints is shown below (in BNF):

```
<constraint>        ::= <expression>
<expression>        ::= '~' <expression>                          |
                        '(' <expression> '|' <expression> ')'    |
                        '(' <expression> '&' <expression> ')'    |
                        <time_shift_op> <expression>             |
                        '(' <operation_match> ')'
<time_shift_op>     ::= '[' INT ']'
<operation_match>   ::= regular expression
<variable_match>    ::= '@' '[' INT ']'
```

The operation matches are regular expressions that can be used to match the text of an operation. The regular expressions can contain standard wild-card characters

(e.g., *, +, ?), range operators, and variable matches [4]. The wild-card characters may be used to simplify the description of the constraints. Variables may be used to enforce any restriction that requires different parts of a single constraint to match (i.e., require different parts of the operation or instruction to map to the same text).

The semantics of the constraint expressions are described below:

```
operation_match      = TRUE iff regular expression matches
                       any field in instruction.

(expr1 | expr2)      = TRUE iff either expr1 or expr2
                       evaluates to TRUE.

(expr1 & expr2)      = TRUE iff both expr1 and expr2 evaluate
                       to TRUE.

~expr                = TRUE iff expr evaluates to FALSE.

time_shift_op expr   = TRUE iff expr evaluates to TRUE with
                       respect to the instruction that is X
                       instruction slots away from the base
                       instruction where X is the INT in
                       time_shift_op.
```

To evaluate a constraint, each sub-expression must be evaluated against the base (or current) instruction or, if time-shifted, against the instruction following the base instruction by the appropriate number of slots. If a constraint yields *FALSE* when evaluated with a particular instruction as the base instruction, then the instruction has violated the constraint and it is considered invalid. Only when an instruction does not violate any constraints can it be considered valid.

We have identified three types of constraints (distinguished by the type of resource that causes the conflict):

- **Datapath Conflicts:** Datapath conflicts result when two parallel operations try to use the same datapath resources (e.g., competition for the bus).
- **Bitfield Conflicts:** Bitfield conflicts results when two parallel operations try to set the same bitfield in the instruction word.
- **Syntactic Constraints:** Syntactic constraints are restrictions that do not correspond to hardware conflicts but are artifacts of the assembler syntax.

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

[4]Note that the syntax for variable matches in ISDL is slightly different than that of conventional regular expression packages.

2.2.6 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 correct design evaluation tools, but may result in better tools if provided. For example, correct code can be generated without knowledge of the structure or presence of caches; however, a compiler may be able to optimize the generated code if information about the caches is provided.

2.2.7 Macro Definitions

In addition to its other abstraction mechanisms, ISDL also provides cpp style macro definitions which allow common patterns to be defined and then reused throughout the ISDL description. This feature can be used, for example, to abstract operations that are common to multiple functional units and only differ in the registers they access. This is achieved by defining a macro that describes the common operation. The operation definitions are then instantiated by using the macro within the instruction set section of the description.

2.2.8 ISDL Model of the Instruction Set

The instruction set of a processor consists of the state available in the architecture and 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.

The processor is capable of recognizing and executing a number of complete instructions. However, instructions in ISDL are described in terms of their components, the operations. In order to arrive at the set of instructions that the processor supports, it is necessary to group operations from the ISDL description into valid VLIW instructions.

ISDL divides the set of available operations into fields. The operations in a given field roughly correspond to the various functions that a single functional unit can perform. Therefore, operations defined within the same field are mutually exclusive and cannot appear in the same instruction. Fields roughly correspond to the functional units that make up the processor and as a general rule these 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 superset of the set of instructions available in the architecture. Some of these combinations are invalid and cannot be executed in hardware - these combinations are reflected in the constraints. The constraints declare a subset of all possible combinations as legal. This subset is the set of valid 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 2-2 shows this mapping function pictorially.

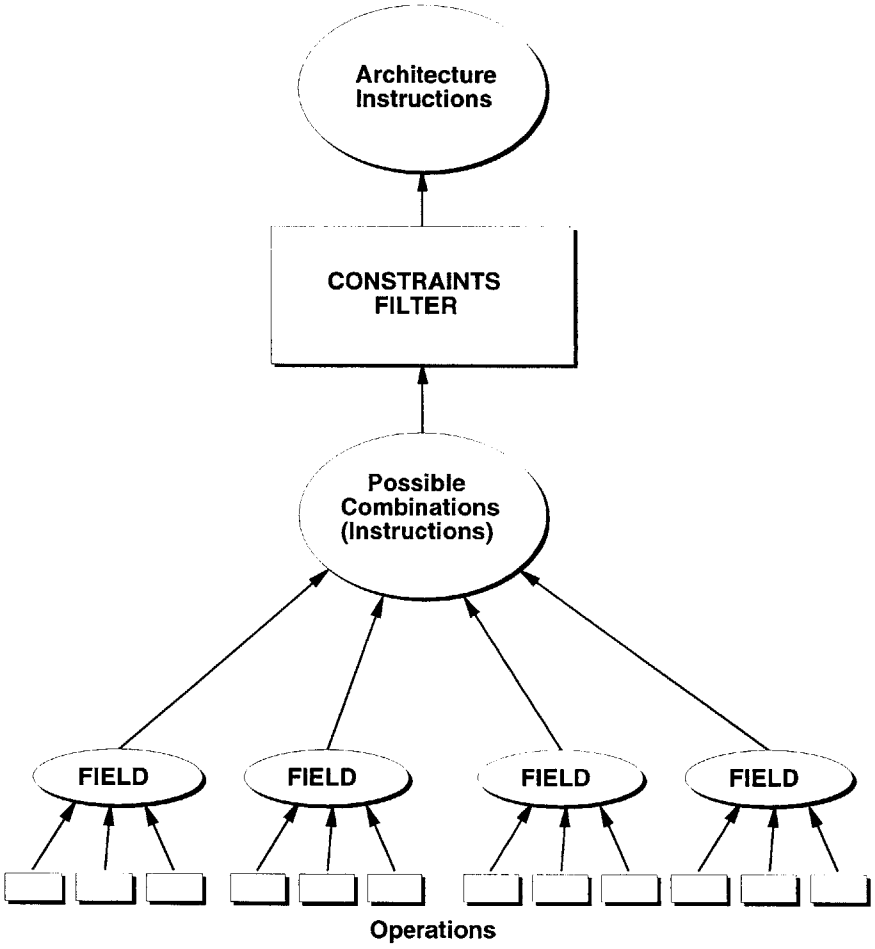

Figure 2-2: ISDL Model of Instructions

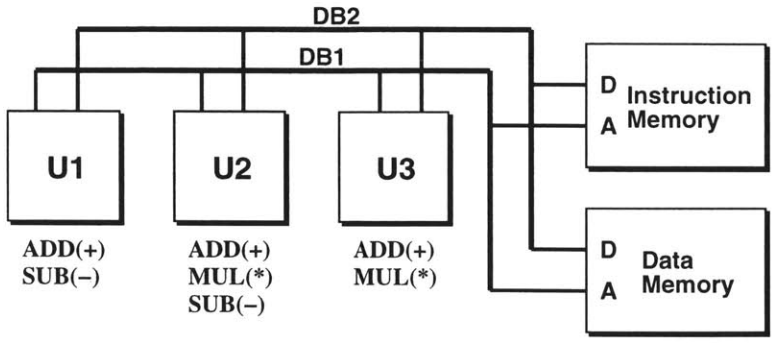

Figure 2-3: **The SPAM VLIW-1 Architecture**

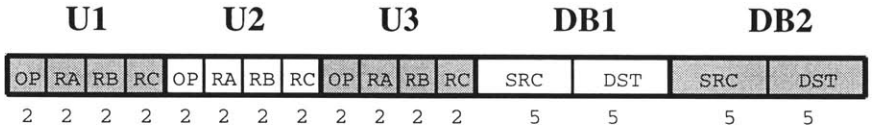

Figure 2-4: **The Instruction Word of the SPAM VLIW-1 Architecture**

2.3 An Extended ISDL Example

An extended example based on the simple architecture of Figure 2-3 is used to better illustrate the features of ISDL. It is a VLIW architecture with three functional units U1, U2, and U3. Each functional unit has its own register file consisting of four 16-bit registers. The architecture also includes a data memory of 4096 16-bit locations and an instruction memory capable of storing 4096 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 this processor, U1 can perform addition and subtraction, U2 can perform addition, subtraction, and multiplication, and U3 can perform addition and multiplication.

The instruction word for the example architecture is shown in Figure 2-4. Each of the functional units 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.

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 further divided into subfields, and each subfield is annotated with its length in bits. The concatenation of each subfield in MSB (most significant bit) to LSB (least significant bit) order results in the instruction word shown in Figure 2-4.

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

```
Section Storage

Instruction Memory INST          = 0x1000 x 0x2C
Memory DM                        = 0x1000 x 0x10
RegFile U1                       =    0x4 x 0x10
RegFile U2                       =    0x4 x 0x10
RegFile U3                       =    0x4 x 0x10
ProgramCounter PC                =          0x10
```

Each of the storage units (memories and register files) is explicitly 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 explicitly identified, and that the program counter must be included even though it is implied by the instruction set.

A sample token definition, which is part of the Global Definitions section, is presented below:

```
Section Global_Definitions

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

The line beginning with the keyword **Token** defines a token that groups 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 actually 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];}. In the full description of the example architecture, two additional tokens exist that define the register names for the other two register files in the same manner.

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 and other non-terminal definitions. This non-terminal consists of a single option. The single token `U1_R` forms the assembly syntax for this option. The return value of this option, which can be used in the bitfield assignments of operations, is the same as the return value of the token as denoted by the 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 return value of the `U1_R` token. The next set of braces contains the RTL side effects of the non-terminal. An empty side effects statement denotes that there are no side effects. The next two sets of braces contain the costs and timing modifiers of the non-terminal option. Undefined cost or timing modifiers imply a value of zero.

The next two lines define non-terminals identical to `U1_RA` except that they are named differently. The reason for defining identical non-terminals with different names is so that they can be distinguished when used in the same operation definition.

The next non-terminal defined is named `SRC` and consists of three options. The syntax of the first option is `U1_R` which represents the registers in the `U1` register file. Similarly, the second and third options represent the registers of register files `U2` and `U3`. This non-terminal can be used to represent a register in any of the three register files. The return value of the non-terminal is defined as follows: for registers in the `U1` register file, the return value is the constant `00` concatenated with a two bit value representing the index of the register, for registers in the `U2` register file the return value is the constant `01` concatenated with the value representing the index of the register, and for registers in the `U3` register file the return value is the constant `10` concatenated with the value representing the index of the register. The RTL action for each non-terminal option is a reference to the appropriate storage location, and there are no side effects or costs and timing modifiers specified. An identical non-terminal named `DEST` is also defined for use in conjunction with the `SRC` non-terminal when describing databus move operations.

Below is a portion of the Instruction Set section for the example architecture:

```
#define ADDm(x,y)     ADD(x,y,16,"trn")

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 <- ADDm(U1_RA,U1_RB); }
    { }
    { Cycle = 1; Size = 1; Stall = 0; }
    { Latency = 1; Usage = 1; }
  ...
Field U2f:
  ...
Field U3f:
  ...
```

This section defines the three functional unit operations, and the memory and databus operations. The functional unit definitions 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 three register names denoted by the non-terminals `U1_RA`, `U1_RB`, and `U1_RC` as parameters. The following is an example of an 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, and the result is stored in the register corresponding to the third parameter of the operation[5].

[5]The reference U1_RC inside the RTL action of the operation description refers to the RTL value of the non-terminal U1_RC. This is given by the corresponding RTL action of the non-terminal definition - in this case a reference to the appropriate register.

The third set of braces describe the side effects of the operation. An empty side effects statement denotes that there are no side effects. Note that in ISDL, the program counter, PC, is implicitly incremented in order to fetch the next instruction and thus does not appear in the operation side effects statement. Also note that explicit manipulation of the program counter denotes a control flow operation.

Finally, the costs and timing parameters of the operation are provided. This add operation takes one cycle to execute, and requires at most one instruction word. This operation will not introduce any stall cycles if a subsequent instruction attempts to access the result of the current operation. The results of this operation are available to all subsequent operations, and functional unit U1 is immediately available to perform another operation.

A load operation from the data memory field is shown below:

```
#define DMdata 0x0C
#define DMaddr 0x0D
#define REG SRC
#define LOC DEST
Field DMf:
  DM_ld  REG, LOC
    { DB1.SRC = DMdata; DB1.DEST = REG;
      DB2.SRC = LOC; DB2.DEST = DMaddr; }
    { REG <- DM[LOC]; }
    {}
    { Cycle = 1; Size = 1; Stall = 1; }
    { Latency = 1; Usage = 1; }
```

The main difference to note in this operation definition is the `Stall` cycle cost. A value of one means that if the next instruction attempts to use the result of the load operation, then the pipeline will be stalled for one cycle and the cycle count for this operation must be increased by one.

Finally, a portion of the Constraints section of the example VLIW architecture description is shown below:

```
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 violated if the instruction contains a databus move operation on either bus (represented by `DB*_move`), *and* the source and destination 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 is not possible to execute in hardware since each register file only has one port attached to each databus. Therefore, a constraint is used to disallow such an operation.

The following is a time-shifted constraint taken from the description of the Motorola 56000 DSP:

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

The first regular expression (i.e., `(Main_REP *)`) is evaluated against the base instruction, while the time-shift operator (i.e., `[1]`) denotes that the second regular expression (i.e., `(Main_DO *)`) should be evaluated against the instruction immediately following the base instruction. This constraint will be violated whenever a `Main_REP` instruction is followed by a `Main_DO` instruction. This particular sequence of instructions is forbidden according to the Motorola 56000 instruction set.

2.4 Describing Real-world Architectures

This section describes several complications that were identified when ISDL was used to describe real world architectures. All of these complications were handled successfully by ISDL.

2.4.1 Time-Shifted Constraints

While describing the Motorola 56000 DSP, we realized that many operations constraint which operations are permissible in the instruction slots that follow them. For example, a `REP`eat instruction cannot be used to repeat a `DO` instruction. There are numerous examples of such constraints in almost any DSP that provides zero-overhead loops. This problem is easily solved by providing a single time-shifted constraint:

```
// You cannot repeat a DO using REP
~((Main_REP *) & [1](Main_DO *))
```

This constraint specifies that a `DO` operation cannot immediately follow (within 1 instruction) a `REP`eat operation.

2.4.2 Register Aliasing

Another issue that seems to be common in many DSP processors is that of register aliasing. We first noticed register aliasing while trying to describe the Motorola 56000 DSP. In this DSP, the `X` register is 48 bits long but can also be accessed as two separate registers called `X0` and `X1`. Writes to `X1` are reflected in the top portion of the `X` register while the bottom portion remains unchanged. Similarly, writes to `X0` affect the bottom portion of `X`, and writes to `X` are reflected in both `X0` and `X1`. The `Y` register and the `A` and `B` accumulators are also aliased in this architecture.

ISDL provides an alias facility explicitly for this purpose. It allows a "new" storage element to be defined and composed from portions of existing state. This storage element does not have any state of its own - rather it shares state with the elements that were used in its composition.

2.4.3 Heavy Op-code Encoding

While trying to describe the SPAM VLIW-2 architecture (see Section 4.1), we discovered that the opcode for the addressing mode for each memory had to be combined with the opcode for the corresponding address generator in order to generate the final value to be inserted in the binary instruction. We call combining opcodes in this manner, *heavy opcode encoding* [6]. One way to handle heavy opcode encoding would be to combine each address generator operation with each data memory operation thus forming a cross product of the two fields. This, however, would violate the orthogonality of operations between the address generator fields and the data memory fields resulting in a much longer and much less intuitive description. Instead, we chose to maintain orthogonality by using the fact that in ISDL bitfield expressions the value of a subfield as set by a previous operation can appear in the right-hand side of another bitfield expression. Thus, the address generator operations are encoded as usual, and they set the opcode first. Then, the data memory operations read the existing opcodes that the address generator operations set and combine them with their own opcodes to produce the final encoding. Hence, the description of the two sets of operations remains completely orthogonal, yet the correct assembly function is described.

2.4.4 Field Opcode Takeover

Field opcode takeover is another property that seems to be common amongst DSP-style architectures with heavy encodings; both the SPAM VLIW architectures and the Motorola 56000 allow operations to take over the opcodes of other fields. Consider an operation `I` in field `M` that requires additional bits to encode large constants and may choose to take over bits that are allocated to the opcode of field `N`. Operation `I` is said to take precedence over field `N`. While the bitfield assignments in ISDL have no trouble describing such an occurrence, the semantics are somewhat tricky. Since the value stored in the bits that normally encode the opcode for field `N` now represent an unknown parameter, we should ignore the opcode of field `N` because it could be any random value. Therefore, when operation `I` is instantiated in an instruction, no operation should be executed for field `N` even though one may be encoded in its opcode. The disassembler and decode logic generated from such a description should take this into account. Note that if field `N` contains an operation `J` that takes precedence over field `M` then the function is undecodeable. If the opcode of `M` encodes `I` and the opcode of `N` encodes `J`, then the precedence is unclear.

[6] Heavy opcode encoding can be used to shorten the instruction word.

2.5 Related Work on Machine Description Languages for Embedded Processors

This section briefly reviews three representative research projects in the area of architecture exploration for embedded systems: MIMOLA [27], FLEXWARE [32], and CHESS [7]. The proceedings of the Dagstuhl workshop [29] contain a collection of papers documenting several other contributors' efforts. In addition, it reviews five other machine description languages: ISPS [1], MARIL [2], LISA [38], HMDES[13] and RADL [34].

2.5.1 MIMOLA

The MIMOLA design system is an environment for hardware-software co-design and includes a retargetable microcode compiler [27][28]. The MIMOLA microcode compiler infers rules for code generation directly from a structural description (e.g., a netlist) of the target architecture instead of a behavioral description (e.g., the instruction set). The advantage of this approach is that it provides a single machine description for the synthesis of the target architecture, the generation of microcode, and the simulation of the target architecture. However, MIMOLA descriptions are generally very low-level, and therefore laborious to write and modify. In addition, simulation in the MIMOLA environment is slow because of the low-level descriptions. Furthermore, the source code for the MIMOLA compiler must be written using the MIMOLA language itself. Finally, the MIMOLA system assumes that the target architecture implementation already exists and precludes using the machine description to generate an implementation. It is also not amenable to local changes in the instruction set because adding an instruction and evaluating the resulting instruction set would require significant changes to the machine description. Thus, MIMOLA cannot be used to explore various implementations of the same ISA.

2.5.2 FLEXWARE, CODESYN, and INSULIN

FLEXWARE consists of a code generator, CODESYN [31], and an instruction-set simulator, INSULIN [36]. The machine description for CODESYN includes three components: a *pattern set of micro-instructions*, the *available resources* and their classification, and an *interconnect graph*. The pattern set of micro-instructions is a behavioral level representation of the instruction-set. This machine description is used for code generation; however, the INSULIN simulator uses another model. It is based on a partially reconfigurable VHDL model of a generic instruction set processor. The user must define the target instruction set in terms of the generic assembler instructions supported by the VHDL model. The main disadvantage of the FLEXWARE system is that it does not permit the same machine description to be used for all of the design evaluation tools. Also, the VHDL model for INSULIN is a parameterized model and thus has a granularity that is coarse compared to that if ISDL.

2.5.3 CHESS and nML

CHESS is a retargetable code generation environment for fixed-point DSPs and ASIPs; it was developed in the context of the CATHEDRAL II high-level synthesis system [8]. The target machine is described using the language nML [10]. Just like ISDL, nML 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. This enables the code generator to recognize more optimization opportunities. The syntax of nML is a subset of the syntax of ISDL corresponding to the definition of non-terminals. In addition, the semantics of nML are substantially weaker than those of ISDL. For example, there is no notion of a field in nML. This limitation makes it difficult to determine which operations correspond to a functional unit. The timing model in nML is based on an execution model of the hardware consisting of a list of resources (e.g., pipeline stages) and reservation tables for each instruction. This means that the appropriate timing information must be derived from the execution model for each instruction. In addition, it requires that all of the pipelines described be of equal length. In contrast, ISDL allows pipelines for different functional units to be of different lengths. nML contains an explicit timing model; however, this timing model is less straightforward than the ISDL methodology which provides simple per-operation timing information. In addition, nML's timing model is not well integrated with the rest of the language because it was added after the initial language was defined.

The most serious shortcoming of nML (as well as most of the other languages reviewed in this section) compared to ISDL is its lack of explicit constraints. In nML, some constraints can be derived; however, the lack of explicit constraints makes descriptions longer and less intuitive. Additionally, the lack of constraints implies that the actual definition of the ISA must present only *valid* instructions thus coupling the ISA to its implementation. Both of these effects are explained in more detail in Section 2.5.8. Finally, there exist constraints that *cannot* be derived from an nML description (the time-shifted constraints) because nML must describe valid instructions and therefore cannot describe interactions between two instructions issued at different times. Architectures containing such constraints are not supported by nML.

An additional shortcoming of nML is its limited capability regarding the description of assembly functions. The expressions and semantics used to create the "image" of an operation are simple and cannot, for example, encode the case of an additional instruction word being used to encode a long constant.

2.5.4 LISA

LISA is a generic machine description language that was originally created to automatically generate fast, instruction-level simulators. Its timing model consists of declared pipeline stages (resources) and resource usage tables for each instruction. In this respect, it is similar to nML. It uses ASAP (As Soon As Possible) Gantt chart scheduling to determine the timing effects of the pipeline. This timing model

is more flexible than the timing model of ISDL. However, ISDL can support most common place architectures and can accurately represent commercial DSPs such as the Motorola 56000. Unlike ISDL, LISA exposes a large portion of the underlying implementation to the description, thus precluding the exploration of different implementations of the same ISA. Furthermore, we believe that the structure of LISA will unnecessarily complicate the retargetable code generator which is the hardest and most complex tool to produce. There is currently no implementation of a retargetable code generator based on LISA.

2.5.5 RADL

RADL[34] is a machine-description language geared towards DSP-style architectures. It contains a fully generic timing model that explicitly declares pipelines and their strategies. This timing model is even more flexible than the timing model in LISA, but this flexibility comes at the cost of increased complexity. Just as in LISA, the RADL timing model exposes the underlying implementation of the architecture to the description, and thus precludes the exploration of different implementations of the same ISA. Furthermore, the additional complexity of the timing model increases the complexity of the code generator because operation timing must now be derived from the pipeline structures instead of being explicitly defined in the description.

2.5.6 HMDES

HMDES[13] is a machine description language that was developed specifically for the TRIMARAN compiler system. It is based on a parameterizable architecture called PLAYDOH[23]. PLAYDOH represents a very general class of architectures which includes features as complicated as predicated execution and complex instructions. While PLAYDOH is very general and can encompass a wide variety of architectures, it is still a parameterized architecture and thus has a limited scope. Similarly HMDES supports a parameterizable instruction set and therefore has a more restrictive scope than ISDL. Like nML, HMDES does not support constraints which may result in longer and less intuitive descriptions.

2.5.7 Other Machine Description Languages

ISPS is a very flexible machine description language with semantics similar to structural Verilog. Due to its semantics, it tends to result in very long descriptions that are not very intuitive. Additionally, it is effectively a structural language even though it has the syntax of a behavioral language. For example, timing has to be emulated by explicit pipelines or some similar method. As a result, ISPS forces a strong coupling between the ISA and its implementation.

MARIL is a machine description language specifically geared towards retargetable code generation for RISC-style general purpose processors. The language cannot handle the complexities of heavily encoded DSP-style processors nor can it handle explicit parallelism. It is therefore unsuitable for a system geared towards embedded

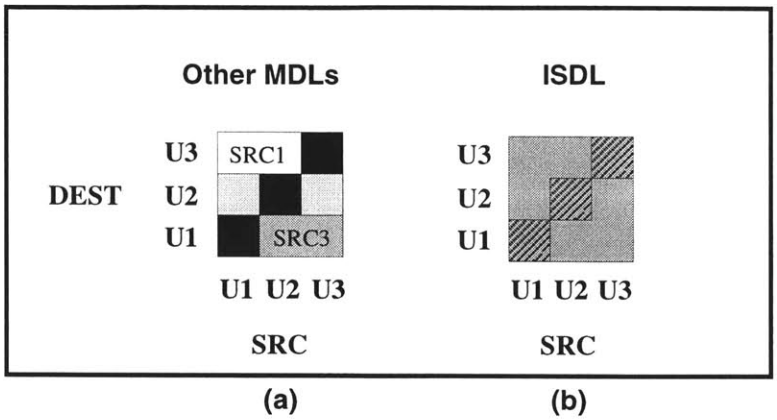

Figure 2-5: **Constraints Help to Simplify the Machine Description**

processors where DSP-style and VLIW architectures are at a distinct advantage over RISC processors.

None of the systems mentioned above provide support for explicit constraints. Without explicit constraints, descriptions for architectures with instruction level parallelism become very laborious to write because every legal combination of operations must be explicitly listed. Constraints can also be useful in decoupling the ISA of a processor from a particular implementation, and in determining resource sharing opportunities which result in more efficient hardware models. Section 2.5.8 elaborates on the benefits of explicit constraints.

2.5.8 Using Constraints to Simplify the Machine Description

Of the languages described in Section 2.5, the nML machine description language has the most generic abstraction mechanism and therefore results in the most concise descriptions. Both nML and ISDL use an attributed grammar as their abstraction mechanism. However, nML does not support explicit constraints. As a result, machine descriptions written in ISDL will generally be significantly simpler, and easier to use, than those written in nML.

Figure 2-5 illustrates how constraints simplify the description of a target processor. In the example architecture of Figure 2-3, the databuses cannot be used to transfer data from one register to another within the same register file since each register file has only one port connecting it to each data bus. Thus, in describing a `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, one would have to list all possible legal combinations of `SRC` and `DEST` as separate `Move` operations. In other words, there would be a non-terminal, `SRC1`, defined as all possible sources except for the `U3` register file, and then a corresponding operation which would specify that a `Move SRC1,U3` is supported by the target processor. This operation is legal because the `SRC1` non-terminal can never take on the value

of the U3 register file. This form would be repeated for each possible destination, as illustrated by each row in Figure 2-5 (a). Thus, resulting in three non-terminal definitions and three corresponding move operations, for this particular example. In ISDL, however, which explicitly supports constraints, one would describe a single operation `Move SRC,DEST` where SRC and DEST are non-terminals corresponding to all possible sources and destinations of the move operation. In other words, the move operation includes the entire space of sources and destinations, as shown in Figure 2-5 (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 is also described. The constraint corresponds to removing the patterned blocks from the instruction space. The combination of one operation and one constraint describes the same information in ISDL, that the three non-terminals and three operations described in nML. If SRC and DEST consisted of seven options each, the ISDL definitions are still valid while the nML definitions would now have to include seven non-terminals and seven operation definitions. It is easy to see that as the instruction space expands and additional constraint dimensions are introduced, this problem grows exponentially for machine description languages that do not support explicit constraints. This simplification is the most significant advantage of ISDL over nML.

Additionally, constraints allow the ISA to be decoupled from the particular implementation. If in the above example, each register file had two ports connecting it to a data bus, a new implementation would result that would not have this particular hardware restriction. In ISDL, this implementation detail can be expressed by simply removing the constraint without changing any of the operation definitions. In nML the actual definition of the ISA would have to be changed by collapsing all of the separate definitions of the operation into a single definition. Thus constraints (which naturally express implementation details) decouple the definition of the ISA from the actual implementation.

Finally, time shifted constraints cannot be expressed in nML. This is because nML can only describe valid instructions and its semantics prevent it from referring to more than a single instruction at a time. This is in contrast to the time-shift operator of ISDL which allows restrictions between multiple instructions to be expressed while still only requiring that the ISA be defined in terms of one instruction at a time. Thus there is a whole class of architectures which nML cannot describe but ISDL can.

Chapter 3

Simulator and Hardware Model Generation

3.1 Introduction

In order to be able to evaluate the suitability of a candidate architecture for a particular application, it is necessary to simulate the program on the particular architecture. This makes it possible to verify performance, determine the utilization of individual architecture features and functional units, and suggest possible improvements to the architecture. Additionally, estimates of the physical costs (e.g., silicon area, power consumption) as well as the clock cycle length of the candidate architecture are necessary to fully evaluate it. These can be obtained by generating a suitable hardware model.

This section presents a pair of tools that generate a simulator and a hardware model given a target architecture described in ISDL. The simulator generator tool is called GENSIM and creates simulators called XSIM simulators. The hardware model generation tool is called HGEN. We would like to note that the hardware model generation tool was implemented by Pietro Russo[33] and it is only described here for completeness since it is based on the infrastructure created for the XSIM simulators.

3.1.1 Requirements and Features

In order to be able to provide the required performance measurements for architecture exploration, the simulators should be cycle accurate and bit-true. Additionally, they should be fast so that realistic data samples can be simulated for the complete application. In order to allow extraction of useful utilization statistics the simulators should provide at least a method of obtaining dynamic execution counts for all instructions. In order to be useful in verifying correctness, the simulators should provide user-friendly interfaces, full debugging support, and support for regression testing.

Accordingly, the XSIM simulators provide the following features:

- Cycle-accurate and bit-true by construction. The XSIM simulators model any

timing and cycle costs that can be modeled in ISDL. This includes all features visible to the instruction set. The XSIM simulators do not model the effects of architecture features that are not visible to the instruction set, such as the effect of caches. The effects of caches can be derived after the simulation from an execution trace.

- Bit-true: The XSIM simulators use the same representations as the hardware, and correctly account for any timing effects described in ISDL.

- Fast: The prototype simulators can currently simulate between 1.1 M and 4.5 M operations per second on a Sun Ultra 10/333. Disassembly is performed off-line to improve speed. This allows real examples to be used when evaluating the architecture. The same methodology can be used to generate compiled-code simulators that can yield much higher performance.

- Execution traces: The simulators have the ability to dump an execution trace either to a file or directly to a processing program. These execution traces can be used, together with static information in the program files, to determine the utilization of architectural features and to measure performance.

- Easy to use interface and full debugging support: The XSIM simulators provide both a graphical user interface and a command line interface. The command line interface can be used for automated simulation runs, and batch-file processing. The graphical user interface can be used for step-by-step debugging of hand-coded examples, as well as for manual operation of the simulator.

- Full debugging support: The XSIM simulators provide breakpoints, state monitors, checkpoints, batch files for regression testing support, and the ability to attach simulator commands to particular instructions.

  - *Breakpoints:* stop execution of the program when a given instruction address is reached.
  - *Monitors:* notify the user when any user-defined part of the state is modified.
  - *Attached commands:* Allow the user to attach simulator commands to instruction addresses. The commands get executed as if they were typed by the user when the particular instruction is executed.
  - *Checkpoints:* At any stage in the simulation the simulator can dump the entire state into a file and re-load it at a later time. This is useful for detecting bugs that occur very late into a simulation run.

- Regression testing support: XSIM simulators can process batch-files (a list of commands placed in a file). They can also maintain a log of a user session and dump the log into a file, which can then be used as a batch file to re-create the results of the session.

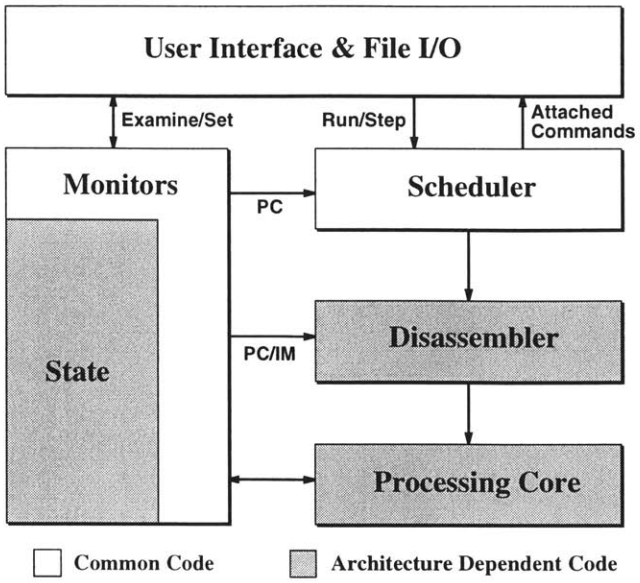

Figure 3-1: **Internal Structure of the XSIM Simulator**

- Off-line disassembly to improve performance: The disassembly of the binary program happens when the program is first loaded, and is stored internally. This improves simulator performance since it takes disassembly off the critical path in simulation.

These features make it possible to use the XSIM simulators for detailed evaluation of candidate architectures. At the same time, they make the simulators easy to use both manually as well as automatically.

In order for the hardware models to be useful, they should be optimized for silicon area and cycle length. Additionally, both the simulator and the hardware model generators should generate their output in a reasonable amount of time.

3.1.2 Simulator Structure

Figure 3-1 shows the structure of an XSIM simulator. All the simulators generated by the GENSIM system share this structure. The simulators consist of six parts:

1. **User Interface and File I/O:** This part implements the command line as well as the graphical interface. The command line interface is written in C. The graphical interface is written in Tcl/Tk and issues commands to the command line interface. It also implements the interfaces to the operating system and file-system.

2. **Scheduler:** The scheduler is responsible for sequencing the instructions during execution, managing most of the debugging support, producing the execution address traces, and dispatching attached commands back to the user interface for processing.

3. **State Monitors:** These detect when any user-defined portion of the state changes, and print a message to that effect. All accesses to the state are routed through the state monitor hooks.

4. **State:** This is a set of data structures that emulate the state of the target architecture.

5. **Disassembler:** The simulator loads the same binary file that would be used to program the ROM on the final hardware[1]. Therefore, the program must be disassembled in order to determine which operations are instantiated in each input instruction. The simulator contains a built-in disassembler which disassembles the program off-line at load time. It then stores the disassembled form internally. The processing core uses this disassembled form to simulate instructions.

6. **Processing Core:** Each operation and ISDL non-terminal option has an RTL action (and an RTL side-effect) associated with it. These translate to a set of routines that emulate those actions. The processing core consists of the collection of these routines.

The sequence of events during simulation starts with a user command to load the binary program file. The user interface loads the appropriate file and then dispatches to the disassembler which disassembles the whole program off-line. During disassembly it detects the operations and non-terminal options instantiated in the instructions, and converts all token instantiations to constants. It then associates with each instruction identifiers that uniquely identify the processing core routines that correspond to the instantiated operations and non-terminal options. It also associates with each instruction the specific token constants. Execution proper begins with a user command to run a program or step through the next (few) instruction(s). This causes control to pass to the scheduler which accesses the PC to find out which instruction to execute. It then checks to see if there is a breakpoint associated with that instruction, and if so passes control back to the user interface. If not, it checks for attached commands and executes them (if any). Then it looks up the identifiers of the processing core routines that were attached to this instruction by the disassembler, and executes the appropriate routines in order. After execution of the processing core routines is complete, it updates the execution trace. It then either passes control to the user interface (if the user command was a `step` command) or fetches the PC again to find the next instruction to execute.

3.1.3 Relationship Between Simulator and Hardware Model

Note that the simulator and hardware model for a particular architecture are very closely related. In particular, we decided to generate hardware models in synthesizable

[1]This ensures that the simulator receives the same information as the hardware, thus making simulator-based testing more robust.

Verilog: these models are themselves simulators. The reason is that both describe the behavior of the architecture in as much level as ISDL will allow, but they are used for different purposes. The simulator simply emulates the behavior, while the hardware model describes an implementation of the same behavior.

This result has a very profound impact on the generation of the hardware model: the model can be generated using the same methodologies that are used to generate the simulator. In particular, the disassembler performs exactly the same function as the decode logic performs in hardware. Similarly, the state portion of the simulator performs the same function as the storage elements allocated in hardware, and the processing core routines implement the same function as the data-paths do in hardware. Therefore, to a first approximation, the hardware model is simply a syntax-translated version of the simulator, minus the user interface, sequencer, and state monitors.

The main difference is the approach to the resource sharing problem (see Section 3.3.2). In the simulator, we do not need to worry about the possibility of sharing resources since the naive implementation of ignoring resource sharing simply results in duplicate pieces of code. Other than increasing the size of the executable and reducing the effectiveness of caches during simulation, this is not a major obstacle. In the hardware model however, ignoring resource sharing results in duplicate data-paths which artificially increase silicon area and power consumption. Therefore, we have to take into account resource sharing. Section 3.3.2 explains how this is done.

3.2 The GENSIM Simulator Generator

This section presents the implementation of a tool called the GENSIM system, that automatically generates an Instruction Level Simulator given an ISDL description of a candidate architecture. This simulator (called an XSIM simulator) can then be used to execute a program in order to measure performance, verify correctness and evaluate the suitability of the architecture.

Figure 3-1 shows the structure of an XSIM simulator. All of the simulator code is written in C, with the exception of the graphical user interface which is written in Tcl/Tk. The user interface, state monitors, and scheduler code is common to all architectures and is implemented as a library. The state data structures, disassembler, and processing core routines are specific to each architecture and are generated as C source code from the ISDL description. The C source can then be compiled and linked with the common library to create an executable program for the simulator. This executable is specific to an architecture, but can load different programs for the same architecture (unlike compiled code simulators).

The following three sections describe how we generate the state, disassembler and processing core of an XSIM simulator.

3.2.1 Storage Generation

Generating the state data structures is the most straightforward step in the simulator generator. For each *primary* (i.e., non-alias) storage element defined in the ISDL description, a data structure is created containing the following:

1. A block of memory sufficient to hold all the data to be stored in the element. For uniformity reasons this block is always treated as a two-dimensional array with the exact depth declared in ISDL and padded to an integer of 32 bits in the width dimension.
2. The name of the storage element as defined in the ISDL description.
3. The type of the storage element as defined in the ISDL description.
4. A flag that denotes whether the element is addressed or not.
5. The width of the two-dimensional array in multiples of 32 bits.
6. The width of the element in bits.
7. An array of monitor flags that are set when the user requests a monitor on a particular location of this storage element.

The only elements of this data structure used by the processing core is the memory block and the element type. All the other elements are used by the user interface and debugging features of the simulator to relate the actual data structure to what the user sees in the panels of the graphical interface.

To handle aliases the simulator generator creates read and write functions. The read functions read the parts of the state that make up an alias, and concatenate them into a contiguous value which they return. The write functions take a value and split it up into the parts of the state that make up the alias. These functions are used exclusively when dealing with aliases of storage elements.

If the debugging support of the simulator is turned on, all accesses to state are automatically routed through the state monitors[2]. Tests for the monitors happen during the write-back phase (see Section 3.2.3).

3.2.2 Disassembler Generation

The bitfield assignments of ISDL provide the *assembly function.* This is a function that given the operations instantiated in the instruction, the options instantiated in any non-terminals, and the actual parameters to the operation, will produce the binary instruction word. More specifically, this function can be modeled as a circuit, where the operation, non-terminal options, and actual operation parameters (such as constants) are inputs to the circuit and the binary instruction word is the output.

[2]The debugging support can be turned off. This helps improve speed since the overhead of checks for breakpoints and monitors can then by eliminated.

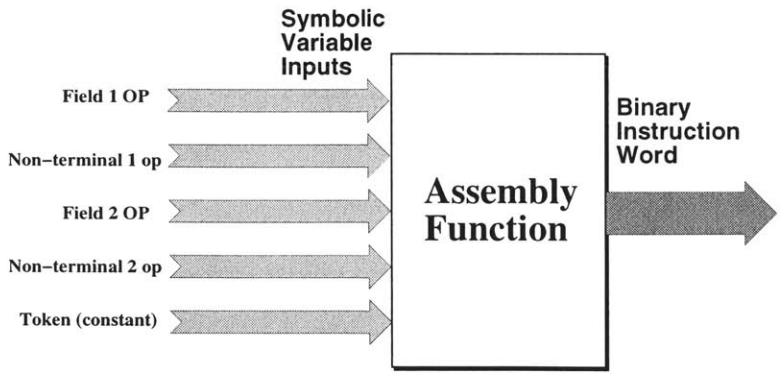

Figure 3-2: **Model of the Assembly Function in** ISDL

In this model, the variables corresponding to which operation is instantiated from each field and which non-terminal option is instantiated in each non-terminal are represented as multi-bit symbolic variables representing a unique identifier. Figure 3-2 shows this model pictorially.

In order to produce a disassembler, we need to reverse this function to derive the *decode function.* This is a function that given an instance of the binary instruction word, produces the symbolic variables that yielded it in the first place. We define a *decodeable* assembly function to be one such that for every valid combination of input (i.e., every valid instruction in the instruction set), the output of the assembly function is unique. This means that if the assembly function is decodeable, we should be able to reverse the assembly function for every valid instruction word[3].

In order to avoid consistency and redundancy issues, we decided not to include the decode function in the machine description. This means that the decode function has to be derived from the assembly function (which is provided in the machine description) in order to derive the disassembler. It turns out that by far the hardest problem in simulator generation is the generation of a disassembler. The general version of the problem is NP-complete (see Theorem 3.2.1) and the number of variables in the input is very large (typically on the order of hundreds or thousands of bits) so a simple search is intractable. Therefore, a number of heuristics are used, and these complicate the algorithms substantially. In addition, approximate answers cannot be used - an exact answer must be determined or the search for a solution aborted.

The Binary Decision Diagram (BDD) Model

Our first attempt to solve the disassembler generation problem was based on Binary Decision Diagrams (BDDs)[4]. The reasoning was that the assembly function can be modeled as a circuit so it can naturally be described by a Reduced Ordered BDD (ROBDD). The ROBDD effectively represents an algebraic description of the assembly function circuit. The circuit can then be algebraically manipulated in order to

[3]In the worst case, we could perform an exhaustive search over the valid inputs of the assembly function until we produce the right instruction word.

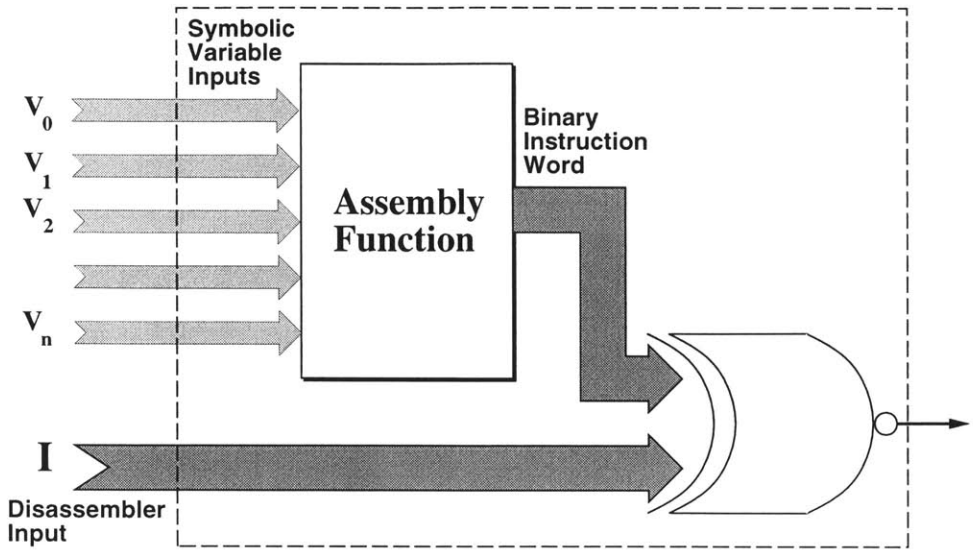

Figure 3-3: **Assembly Function Modified for Reversal**

reverse it.

In particular, the bitfield assignment expressions form trees of combinational logic with the inputs to the assembly function as their variables. The output of each expression either forms the return value of a non-terminal, or is assigned directly to the subfields of the instruction word. If the outputs of an expression tree form the return value of a non-terminal, then they are connected directly wherever a bitfield assignment expression makes use of that non-terminal. A new instance of the non-terminal expression is used for each appearance of the non-terminal. The end result is the circuit representing the assembly function. Let us call this circuit A. All complications in the assembly function (such as heavy op-code encoding which appears as subfields in the r-value of a bitfield expression, op-code takeover, etc.) are all represented in the resulting circuit and will be taken into account once the circuit is reversed.

To reverse the circuit, we modify it as shown in Figure 3-3. This circuit takes an as an additional set of inputs the particular instruction word we wish to disassemble. Its output will be TRUE if the assembly function input variables V_x represent the correct disassembly of the instruction word, and FALSE otherwise. Let us call this circuit D. Given this circuit and an instruction I to be disassembled, we can co-factor D with respect to each bit of I by setting the corresponding input to the appropriate value. Let us call the resulting circuit D'. D' will yield true if the assembly function input variables $V_1 ... V_n$ are set to the values that originally produced I. We now need to find a setting for these variables that satisfies D'. We can represent D by a BDD and co-factor it and find a variable setting that satisfies D'.

Let B_D be the BDD for circuit D. Figure 3-4 shows pseudo-code for constructing B_D. This BDD only needs to be created once during initialization of the disassembler. Let B'_D be the BDD that represents D'. This BDD will have a number of the nodes for the variables $V_1 ... V_n$ in it (not necessarily all) arranged in a chain which leads

```
generate_bdd_top()
    clear word
    for each field i
        field_bdd = create_bdd_mux(num_of_ops(i))
        mux_select(field_bdd) = identifier(i)
        for each operation o in i
            for each bitfield assignment statement a in o
                tmp = generate_bdd_expression(rvalue(a))
                word = insert_bdd(subfield(a), tmp, word)
            end
            connect_bdd_mux(field_bdd, index(o), word)
        end
    end
    create_bdd_xnor(word, input)

generate_bdd_expression(e)
    switch (type(e))
        case tok:
            return create_bdd_parameter(token(e))
        case ntl:
            return generate_bdd_nonterminal(non-terminal(e))
        case and:
            tmp1 = generate_bdd_expression(left(e))
            tmp2 = generate_bdd_expression(right(e))
            return create_bdd_and(tmp1, tmp2)
        ...
    end

generate_bdd_nonterminal(n)
    ntl_bdd = create_bdd_mux(num_of_options(n))
    mux_select(ntl_bdd) = identifier(n)
    for each option o in n
        r = return_expression(o)
        tmp = generate_bdd_expression(r)
        connect_bdd_mux(field_bdd, index(o), tmp)
    end
    return ntl_bdd
```

Figure 3-4: **Generating the Decode BDD B_D**

```
disassemble_with_bdd(b, I)
    set all input variables V_X to "don't care"
    tmp = b
    for each bit b in I
        tmp = cofactor_bdd(tmp, b)
    end

    for each node n in tmp
        if (branch(zero, n) is FALSE)
            set variable(n) to one
        else
            set variable(n) to zero
    end
```

Figure 3-5: **Disassembly Using Decode BDD B_D**

to the TRUE node at the bottom of the BDD. All other branches will lead to the FALSE node. By traversing this B'_D towards the TRUE node and examining the values required for the variable at each node, we obtain a subset of the assembly input variable bits (V_x) and the values they must be set to in order to produce I. Note that traversing the co-factored BDDs can be performed in linear time since all nodes are arranged in a chain and any deviation from the correct path instantly leads to the zero node (which means no back-tracking is necessary). B'_D has to be created every time an instruction is disassembled. Figure 3-5 describes this algorithm in pseudo-code.

This formulation is very elegant and takes care of all the complications that our assembly function is allowed to contain. Since B_D takes into account all of these complications, and B'_D represents an exact account of which assembly function input variables were set to what values in order to produce the particular word, we know that we will obtain the correct values. Note that input variables to the assembly function that were not used to produce this word (such as parameters for operations not instantiated in this word) will disappear from the BDD when it is co-factored. Similarly, operation identifiers for fields whose op-codes were taken over will disappear when the BDD is co-factored.

Given the elegance and cleanliness of the above methodology, it was natural to attempt to implement it. However, very soon we run into problems. BDDs are notoriously sensitive to the ordering of the input variables and tend to explode to an exponential size if the ordering is not just perfect. Despite several attempts to produce ordering algorithms that would contain the problem, the BDDs became prohibitively large and often consumed all the memory available on the machine before terminating. With the benefit of hindsight this is hardly surprising: we know that the general version of the disassembly problem is NP-complete and therefore an exponential run-

time is to be expected. However, traversing the co-factored BDDs is linear with respect to the number of variables, as is co-factoring. It follows that the expected exponential step would have to be the construction phase of B_D and that the resulting BDD would most likely be exponential in size.

The Signature Model

The main reason for the failure of the BDD model was that no heuristics could be applied in an intelligent fashion to reduce the size of the search. The problem was reduced to a different domain and transferring knowledge of the structure of general decode logic circuits to this domain is non-trivial. A domain closer to the actual problem of generating decode circuits was necessary so that intelligent heuristics could be applied.

We produced a model of such a domain, called the *Signature* model. In this model, we annotate every bitfield assignment expression with a signature. This is a symbolic representation of the values of the bits of the expression. A signature is an array of bit symbols, with the same width as the bitfield expression. Each bit symbol can be one of:

- "Unknown" entries (represented by "z") imply that the assembly function for this operation or non-terminal option does not set the corresponding bit.
- "Don't care" entries (represented by "x") imply that the assembly function for this operation sets this bit to one in some cases and to zero in others.
- The constant "0" or "1" implies that the assembly function for this operation sets the corresponding bit to the given constant for all cases.
- A parameter symbol (represented by "s") implies that the assembly function for the operation sets the corresponding bit to a function of the value of one of the parameters.

Signatures of sub-expressions in a bitfield assignment expression can be composed to derive the signatures to the top-level expression. Table 3.1 shows some of the composition rules for signatures - the rest follow the same pattern. Also note that composition of a parameter symbol with another parameter symbol yields an error. The reasons for this is the *parameter decode axiom* (see axiom 3.2.1).

To determine the signature for the return value of a non-terminal, the signatures for each option of the non-terminal are *merged*. Table 3.2 shows the rules for signature merging. Note that the merging rules form a commutative and associative function.

Before we can generate a disassembler, we need to generate signatures for each operation and non-terminal option. To do this we generate signatures for the r-value of each bitfield assignment in the operation (or non-terminal option), and insert these signatures at the appropriate points inside a signature representing the whole instruction word (or the return value of the non-terminal). Figure 3-6 presents the pseudo-code for this step. Figure 3-7 shows an example of how operations and non-terminal options are annotated with signatures.

Expression Node Type	Left Child	Right Child	Result Symbol
Token	NA		s
Non-Terminal	z		z
	x		x
	0		0
	1		1
	s		s
AND	0	any	0
	any	0	0
	x,1,z	s	s
	s	x,1,z	s
	1	x,1,z	x,1,z
	x,1,z	1	z,1,z
	s	s	ERROR
OR	1	any	1
	any	1	1
	x,0,z	s	s
	s	x,0,z	s
	0	x,0,z	x,0,z
	x,0,z	0	z,0,z
	s	s	ERROR
XOR	x,1,0,z	0	x,1,0,z
	0	x,1,0,z	x,1,0,z
	1	x,1,0,z	x,0,1,z
	x,1,0,z	1	x,0,1,z
	any	s	s
	s	any	s
	s	s	ERROR
"$\gg$"	any	integer	shift symbols right

Table 3.1: **Composition Rules for Signatures**

```
signature_annotate()
    for each field f
        for each operation o in f
            clear word
            for each bitfield assignment a in o
                tmp = generate_signature(rvalue(a))
                insert_signature(word, tmp, subfield(a))
            end
            annotate o with word
        end
    end

generate_signature(e)
    switch (type(e))
        case tok:
            return create_symbol_signature(token(e))
        case ntl:
            return generate_ntl_signature(non-terminal(e))
        case and:
            tmp1 = generate_signature(left(e))
            tmp2 = generate_signature(right(e))
            return compose_signature(and, tmp1, tmp2)
        ...
    end

generate_ntl_signature(n)
    clear ret_signature
    for each option o in n
        r = return_statement(o)
        tmp = generate_signature(r)
        ret_signature = merge_signature(ret_signature, tmp)
    end
    return ret_signature
```

Figure 3-6: **Annotating Operations and Non-terminal Options with Signatures**

Signature 1	Signature 2	Result
x	x	x
x	1	x
x	0	x
x	z	x
x	s	s
1	1	1
1	0	x
1	z	1
1	s	s
0	0	0
0	z	0
0	s	s
z	z	z
z	s	s
s	s	s

Table 3.2: **Merging Rules for Signatures**

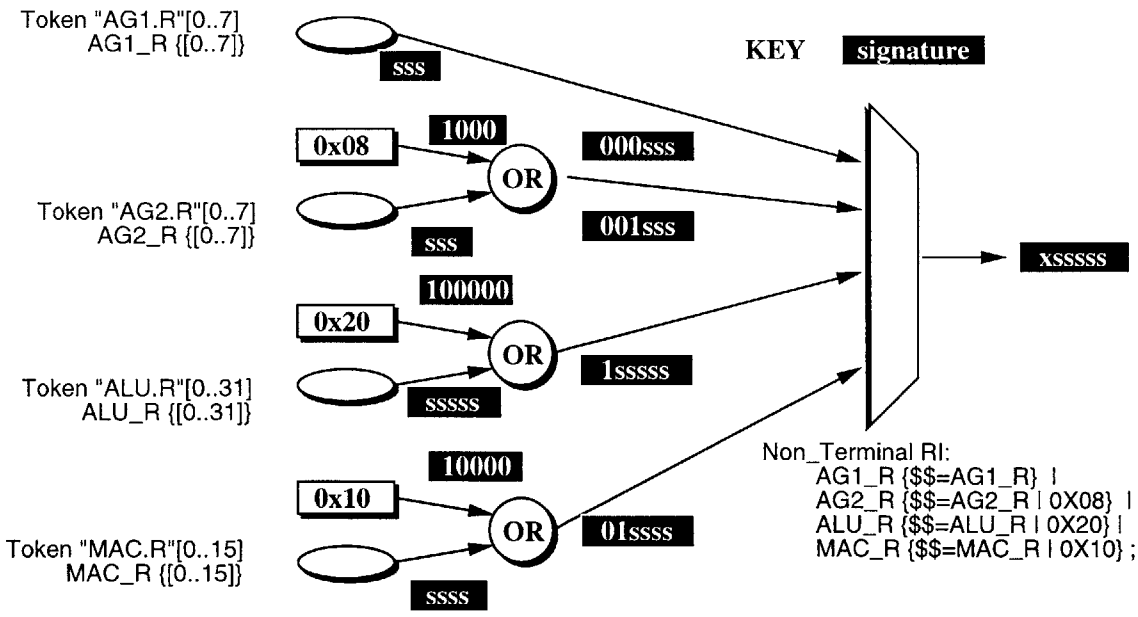

Figure 3-7: **Example of Signature Annotation**

Determining Op-codes

Once we have annotated each operation and non-terminal option with a signature we can proceed to generating a disassembler. We define the *op-code* of an operation or non-terminal option to be the set of bits in its signature that are set to constants (i.e., "1" or "0"), and the values that these bits are set to. If the assembly function of the architecture is decodeable, then the op-codes for each operation must be unique. We say an op-code is unique if its bit-pattern in the instruction cannot be produced by any other combination of operations and parameters. Assuming all op-codes are unique, we can determine if an operation is instantiated in the current instruction by matching its op-code against the instruction word. If we obtain a match, we *know* that the operation was present in the instruction by the definition of uniqueness. Once we know which operations are instantiated in an instruction, then we know which parameters to expect and we can proceed to decode them.

While we could rely on the input to be decodeable and therefore all the op-codes to be unique, it is desirable to perform a check for uniqueness. This would alert the user to invalid inputs and direct him/her to fix the problem. Also note that in the face of op-code takeover and field merging, uniqueness testing becomes a requirement. However, uniqueness testing is NP-complete and therefore exponential in run-time in worst case.

Theorem 3.2.1 *Uniqueness testing for op-codes is NP-complete.*

Proof Consider two signatures `10001zzzz` and `1000xssss`. In order to determine if their op-codes are unique, we need to ensure that bit 5 in the second signature (marked "x") can never result in "1".

- First we need to show that op-code uniqueness is a member of NP. Assume we could non-deterministically obtain the combination of operations and parameter values that would set this bit to 1. Then, starting with these values, we could evaluate every sub-expression and finally the top-level expression yielding the above bit. Note that since ISDL bitfield assignment expressions are arranged as trees, each sub-expression only needs to be evaluated once and we can evaluate all the expressions in time $O(n)$ where n is the size of the bitfield expression that yields the required bit. Therefore, we can verify the non-deterministic choice in polynomial (actually linear) time and thus op-code uniqueness is a member of NP.
- Now we need to show that a known NP-complete problem can be reduced to op-code uniqueness in polynomial time. We choose to reduce 3-SAT to op-code uniqueness. A 3-SAT problem is arranged as a conjunction of a set of terms. Each term is a disjunction of 3 variables either in the normal form or in the inverted form. For each variable in the given 3-SAT problem, we define a non-terminal with two options. The first option returns a value of "0" and the second a value of "1". For each term in the given 3-SAT problem, we combine the return

values of the non-terminals corresponding to the variables in the term using the "|" operator of ISDL. If the corresponding variable in the 3-SAT term was inverted, we invert the return value of the corresponding non-terminal using the ISDL "!" operator. We then combine all terms using the "&" operator of ISDL into an expression which we then use to set bit 5 of the second signature in the example above. We need to construct as many non-terminals as there are variables in the given 3-SAT problem. So the non-terminal construction phase takes $O(m)$ steps where m is the size of the given 3-SAT problem. We also need to construct as many terms as were present in the given 3-SAT problem. Since each term is independent of the others, this once again takes $O(m)$ steps. Each step takes $O(1)$ time. Finally, we need to create an expression in ISDL that contains all the terms constructed. This expression takes $O(m)$ time to construct. Setting the corresponding bit in the second signature above takes $O(1)$ time. Therefore, the 3-SAT problem can be converted to an op-code uniqueness problem in $O(m)$ time. Finally, we attempt to determine if the two op-codes given above are unique. In order for them to be unique, bit 5 of the second signature must be "0". So if we determine that the op-codes are unique then we know that there is no satisfying assignment to the original 3-SAT problem. Note that if we wish to determine whether there *is* a satisfying assignment to the original 3-SAT problem, we try to determine if the constructed op-code is unique with respect to signature `10000zzzz`. Using the above steps, any 3-SAT problem can be reduced to an op-code uniqueness problem in polynomial time in such a form that resolving the op-code uniqueness problem will provide the answer to the original 3-SAT problem.

■

We can determine op-code uniqueness by verifying that *every* pair of op-codes is *pairwise* unique. Two op-codes are pairwise unique iff:

- They contain at least one bit in common and they set it to different values.
- There is a set of bits in the two op-codes that does not overlap and that no combination of operations and parameters other than the current operation can produce.

As expected, the last step above will result in an exponential search. To implement this search we may have to expand non-terminal signatures into the set of signatures for each option in order to test for all possible combinations of parameters. We may also have to explore all possible combinations of operations. We use a number of heuristics to prune the search as much as possible:

- Do not explore operations which never set any of the bits in question. This will also eliminate non-terminal expansion of any non-terminals used by these operations.

- Subtract the overlapping part of the op-codes before testing for bit patterns to increase the probability that operations will be pruned by the heuristic above.
- Memoise the results of non-terminal expansion since these are likely to be used often.

Note the despite the fact that op-code uniqueness (and therefore disassembler generation) is NP-complete, matching of op-codes (and therefore the generated disassembler) proceeds in linear time.

Field Merging

In order to deal with heavy-opcode encoding, ISDL allows subfields to appear in the r-value of bitfield assignment expressions (see Section 2.4.3). This means that the signatures of the operations of the second field are compositions of the operation signature and a symbol representing the subfield. Typically, the resulting signatures do not contain any constants and therefore have no op-code.

In order to resolve the above problem we *merge* the two fields involved by performing a cross-product on their operations. The result is a *merged* field which contains as many operations as the product of the number of operations of each field. Each operation from the first field is combined with each operation from the second field, and the signature of the first operation is composed with the signature of the second operation to create a signature for the merged operation. The signatures of the merged operations are used to perform op-code matching. Once we know which merged operation matched, we can determine which of the original operations were instantiated in the instruction.

Determining Precedences from Op-code Overlaps

In the complex instruction sets common in embedded processors, it is common for an operation $o1$ from field X to take over bits belonging to the op-code of an operation $o2$ in another field Y to store large constants. This is called *op-code takeover*. Since the bits of the op-code of $o2$ have now been set to whatever random value corresponds to the constant, the semantics of op-code takeover state that operations in field Y must be disabled (i.e., that no operation will be performed for field Y). We say that operation $o1$ takes *precedence* over field Y. The disassembler must maintain these semantics.

In order to implement these semantics, we analyze the signatures of all operations and attempt to detect op-code overlaps. Op-code overlap is the case where the op-code of an operation overlaps with anything other than the symbol "z" in the signature of another operation. If there is an op-code overlap, then there is a good chance that there is a precedence relationship between the two operations. A true precedence between $o1$ and $o2$ exists if the op-code of $o2$ partially or completely overlaps with symbols "s" or "x" in the signature of $o1$. If the op-code bits themselves overlap, then obviously only one of the two operations can execute. If the op-codes are unique

then only one operation will match so only that operation will execute. If they are not unique then the assembly function is undecodeable and an error is produced.

Once we have determined the precedences between operations, we arrange them in a DAG where the nodes are the fields and the edges are precedences between operations in the fields. Note that there is one edge in the DAG for *each* precedence relationship between operations so there may be multiple edges between nodes. Also note that if $o1$ in field X takes precedence over field Y and $o3$ in field Y takes precedence over field X, then the precedence relationship cannot be resolved since either operation (or even both) may have been taken over. Therefore, cycles are not allowed in the precedence DAG and they are signaled as errors.

Note that the edges may form chains of nodes where $o1$ in field X takes precedence over field Y and operation $o4$ in field Y takes precedence over field Z. Assume that an instruction matches the op-codes for $o1$ and $o4$. Since $o4$ matches, then one would expect that field Z would be disabled. However, $o4$ is invalid since it belongs to field Y which is disabled by $o1$. So in such a case, field Z is *not* disabled. To take this effect into account the precedence DAG is sorted in topological order. During disassembly, and once the operation matches have been determined, the precedence DAG is traversed in topological order looking for instantiated operations that take precedence over fields. When such operations are found, the corresponding fields are disabled and, in addition, any operations in them are ignored while looking for precedences.

Parameter Decoding

Once the disassembler has determined which operations are instantiated in an instruction and precedences have been taken into account, we know which parameters to expect for each operation. We now have to decode these parameters based on the bits in the instruction word.

Our methodology is based on the following axiom (which we call the *parameter decode axiom*):

Axiom 3.2.1 *Each parameter symbol in a signature is a function of a* single *parameter only.*

What this axiom means is that no bit in the instruction word will be a function of more than one parameter and therefore no back-tracking will be necessary in order to decode the parameters. The reason why signature composition does not allow composition between two parameter symbols is that such an event would violate the parameter decode axiom. All architectures known to us obey the parameter decode axiom. It is easy to see why this axiom would hold for any realistic architecture:

- Encoding parameters in a fashion that violates the parameter decode axiom does not confer any advantages in terms of performance and rarely confers any advantage in terms of code size since most parameters are powers of 2 and will fit exactly in a given number of bits.

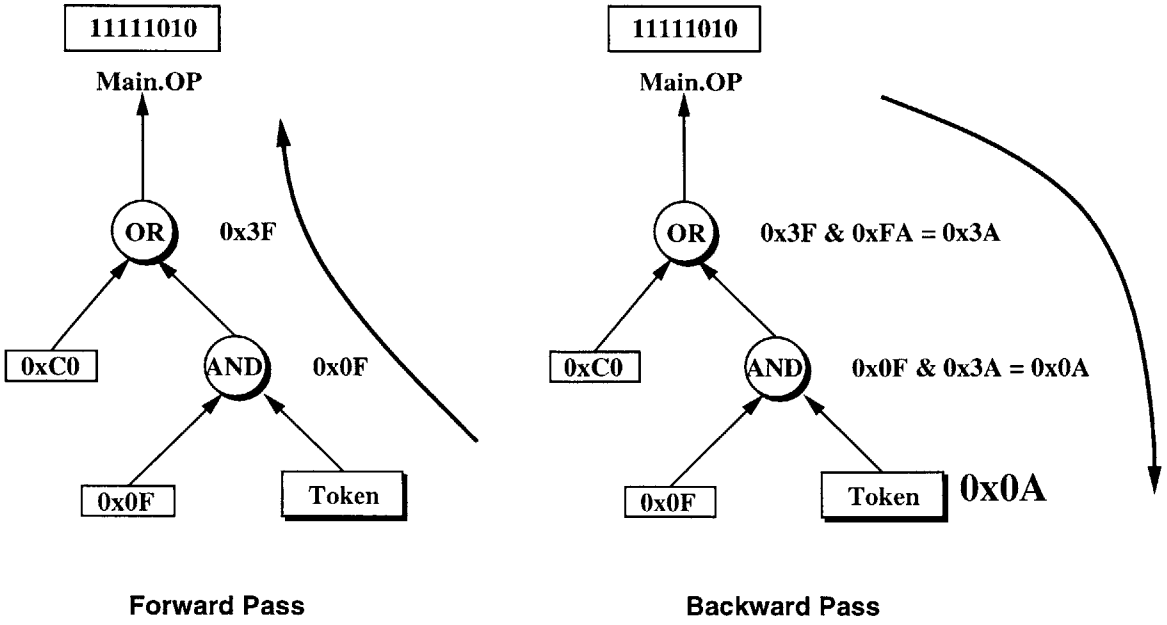

Figure 3-8: **Example of the Forward/Backward Pass Algorithm**

- Back-tracking in the disassembly algorithm translates to very expensive exponential lookup tables in the decode logic.

Given that the axiom above holds, we know that no back-tracking will be necessary to decode the parameters. We can therefore use a simple forward-backward pass algorithm[3] to decode the parameters. In the forward phase, we propagate all constants through the bitfield expressions to reconstruct any parts of the intermediate values in these expressions that do not depend on the parameters. In the backward pass, we propagate the values present in the instruction backwards through the bitfield assignment expressions to arrive at the values of the parameters. Note that because of the parameter decode axiom we only need to perform each pass once. Figure 3-8 illustrates this process with an example.

Note that for non-terminals, the value obtained is the return value of the non-terminal. This has to be recursively disassembled by matching against the op-codes of each non-terminal option and once the option is identified, recursively decoding its parameters.

Overview of Disassembly Process

In the XSIM simulators, disassembly happens during program load time. Once the identifiers for each operation have been generated (i.e., the operations have been identified) and the parameters have been decoded, these are stored for reference during execution. Thus, the cost of disassembly is not incurred during simulation. Even though the run-time of the generated disassembler is linear with respect to the size of the description, it is still a substantial overhead which can be avoided by performing

the disassembly off-line. When the instruction memory is written, the disassembler is called again for this particular location. Thus, the XSIM simulators can handle self-modifying code.

Figure 3-9 shows the algorithm of the overall disassembly process in pseudo-code.

3.2.3 Processing Core Generation

The processing core is merely a collection of routines that correspond to the RTL statements in the actions and side-effects of operation and non-terminal option definitions. These are arranged as a set `switch` statements - one for each field - that, depending on the operation identifier created by the disassembler, will emulate the effects of the corresponding operation. The only complication is maintaining the correct timing and costs semantics.

Maintaining the Costs and Timing Semantics

ISDL semantics state that all RTL statements within every operation in an instruction read their inputs before any operation writes any of its results. Additionally, some operations may not write their results back to storage immediately. They may instead write them at a later time denoted by the value of the `Latency` expression for the operation. To implement these semantics, we divide the execution cycle into two distinct phases:

- **Evaluation Phase:** During this phase we evaluate the costs and timing expressions for each operation. Then all RTL statements read their inputs and evaluate their results. These results are inserted in the *write-back buffer*. The write-back buffer is a set of entries each of which consists of:
  - the value to be written back to storage
  - the storage element and index at which this value will be written
  - a mask showing which bits of the storage element are to be updated
  - a time marker showing when the update is scheduled to happen, in fetch cycles from the current instruction (i.e., the value of the `Latency` expression).
- **Write-back Phase:** During this phase, the write-back buffer is scanned for entries that are scheduled to be written out to storage. The corresponding storage elements are updated and the entries in the write-back buffer cleared and prepared for re-use. During the update of the storage elements, the corresponding memory monitors are checked and messages are printed out if a monitor was placed on the storage element and location by the user. Also, the cycle count is updated.

In addition to the above semantics, we must also maintain the semantics of stall cycles. Whenever a value is inserted into the write-back buffer, it is also inserted into

```
Generate signatures for each operation in each field
Generate signatures for each option in each non-terminal

disassemble(i)
    for each f in description
        disassemble_field(i, f)
    end

disassemble_field(s, f)
    for each operation o in f
        if signature of o matches s
            for each parameter p in o
                case (p)
                    token: reverse s to get token value t
                    non-terminal: reverse s to get return value r
                        disassemble_ntl(r, p)
            end
        return OK
    end
    return ILLEGAL INSTRUCTION

disassemble_ntl(s, n)
    for each option o in n
        if signature of o matches s
            for each parameter p in o
                case (p)
                    token: reverse s to get token value t
                    non-terminal : reverse s to get return value r
                        disassemble_ntl(r, p)
            end
        return OK
    end
    return ILLEGAL INSTRUCTION
```

Figure 3-9: **Disassembly Algorithm**

a stall-cycle reservation table (if the `Stall` cost of the operation was not 0). Each entry in this table contains:

- The storage element and index being written
- The value of the `Stall` expression of the operation.

During the execution phase, if an RTL statement attempts to read a location for which there exists an entry in the stall-cycle reservation table, the corresponding number of stall cycles are noted for the current instruction. During the write-back phase, the maximum stall-cycle cost is computed and added to the cycle-count and the entries in the stall-cycle reservation table are derated accordingly.

Translating the ISDL RTL

Translating the RTL statements is relatively simple. Wherever possible, statements are translated to native C operators. The results of these operators is masked according to the width of the RTL expression to guarantee that the computed values are bit-true. Code is generated to detect special conditions such as overflows (which set the `OVF` flag of ISDL). Non-terminals translate to switch statements that consult the disassembled form of the instruction to figure out which non-terminal option is instantiated. They then evaluate the statements for the appropriate option and return the computed value (if any).

Accesses to storage elements in the r-value of RTL expressions translate to an access to the memory block corresponding to the storage element. A test is inserted to check the stall-cycle reservation table for possible stalls.

Accesses to storage elements appearing as l-values in RTL expressions translate to code that inserts the computed value into the write-back buffer and fills in the buffer entry with the appropriate mask, storage element, and index information.

3.2.4 Restrictions

While the simulator generator supports a wide variety of architectures and architectural features, it still does not support all of the architectural design space that ISDL supports. This section describes the restrictions on what the current simulator generator can handle as input. The restrictions are divided into two classes:

1. Restrictions that are imposed by the algorithms used.
2. Restrictions that are imposed by the current implementation of these algorithms and could easily be lifted by additional implementation effort.

The following sections describe these restrictions in detail. In addition, they state whether each restriction is a hard restriction that cannot be avoided by restructuring the ISDL input, or whether it is a soft restriction that can be circumvented by using a different description style in ISDL. Hard restrictions limit the portion of the architecture design space that can be explored by GENSIM while soft restrictions do not.

Algorithm Restrictions

1. The current algorithms expect the parameter disassembly axiom (see axiom 3.2.1) to be obeyed. The simple forward-backward pass for decoding parameters will fail if this axiom is not obeyed and incorrect results will be generated. This is a hard restriction.

2. The simulator generator does not currently make use of the constraints in the description when generating the disassembler. These constraints should not be necessary to make the function decodeable. Consider the following example:

```
Non_Terminal TEST:
                X {$$ = 0x000;} ... |
                Z {$$ = 0x100;} ... ;

        op_1    TEST {MAIN.OP = 0x1000 | X;} ...
        op_2         {MAIN.OP = 0x1100;} ...

~(op_1 Z)
```

   In this case, the op-codes for `op_1` and `op_2` are distinct but only because of the constraint that prevents the second option of the non-terminal `TEST` from being used with `op_1`. Since the generator never looks at the constraints, it would declare an op-code conflict and fail. This is a soft restriction; the example above could be recoded so that `op_1` takes the parameter `X` directly rather than through the non-terminal.

3. One of the two parameters of the bitfield operator * must be constant. This is so that an exponential search can be avoided upon meeting such an operator. This is a soft restriction as long as the operator does not violate the parameter encoding axiom.

4. The right operand of a bitfield shift operator must be a constant. This is so that an exponential search can be avoided upon meeting such an operator. This is a soft restriction as long as the operator does not violate the parameter encoding axiom.

5. Subfields can only be used in the r-value of a bitfield expression to denote heavy op-code encoding. The simulator generator automatically attempts to merge fields if it detects a subfield used as an r-value. This is a hard restriction.

6. If the options of a non-terminal cannot be distinguished from each other (e.g., they have no op-codes) then they must be equivalent. This is defined as: each option of the non-terminal returns in RTL exactly the same expression it returns in bitfields modulo the keyword `CURRENT` being replaced by the value of the Program Counter. This is a soft restriction.

Implementation Restrictions

1. Dependent non-terminals (i.e., non-terminals that set the instruction word directly rather than through their return value) cannot be used as parameters for other non-terminal options. This is a soft restriction.

2. The input must be arranged in such a fashion so that non-terminals should not be necessary in order to determine the order in which precedences are evaluated. Non-terminals *are* taken into account in evaluating precedences once these have been ordered. This is a soft restriction.

3. Either all or none of the options of a non-terminal should have return values. This is a soft restriction.

4. If a non-terminal has return values for its options then the options should be decodeable simply be looking at these return values. This is a soft restriction.

5. Only two fields can be merged into a single field during heavy op-code encoding analysis. This is a soft restriction.

6. If either the action or side-effects return value of a non-terminal is indexed in any part of the description, then all options in the non-terminal should have a return value corresponding to a storage reference of depth 1. This is a soft restriction.

7. No RTL expression can be wider than 64 bits. This is a soft restriction.

8. In the current version of the simulator, the `LIMIT` and `PLIMIT` flags and the `CHANGED` function have not been implemented so they always return FALSE. This is a soft restriction.

9. The "equ" mode for the `RND` function has not yet been implemented - it is currently replaced by "near". This is a soft restriction.

10. The `MUL` and `DIV` functions can only operate in "trn" mode (no "sat"). The `MUL` function also does not set the `OVF` flag. This is a soft restriction.

11. The `*`, `+`, and `-` operators in RTL will not set the `OVF` flag. This is a soft restriction.

12. The `EVAL` and `HALT` functions operate with the debugging features of the simulator disabled. This is a hard restriction.

13. Assignments to local variables in RTL cannot be indexed. Local variables cannot be used as carry/borrow flags in `ADDC` and `SUBC` either. This is a hard restriction.

With the exception of the above restrictions, the simulator generators can handle any valid ISDL input.

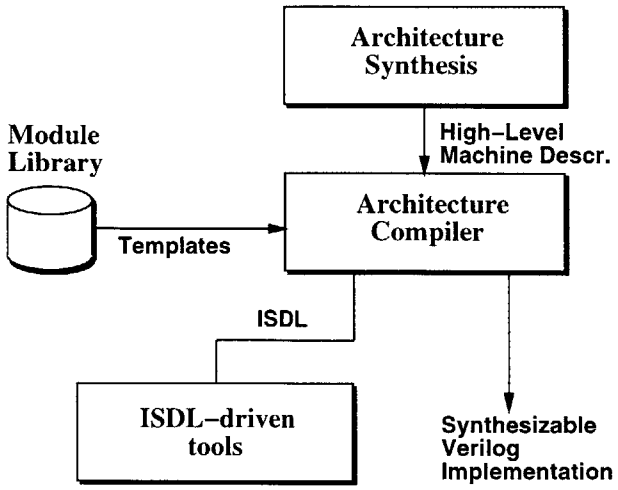

Figure 3-10: **Implementation Based on Module Libraries**

3.3 Hardware Model Generation

In order to fully evaluate an architecture we need to obtain an estimate of its physical costs (e.g., silicon area or power consumption). At the same time, while the XSIM simulator provides performance measurements in terms of cycles, the length of the cycle is still necessary to obtain an accurate measure of performance. Both the cycle length and the physical costs can be determined by synthesizing a hardware model for the architecture. We consider a description of the architecture in synthesizable Verilog to be a sufficient hardware model. This description can then be used to map to any kind of underlying technology using modern CAD tools (silicon compilers).

We compare two competing approaches to this problem, and provide a detailed description of the one we selected.

3.3.1 Module Library Approach

One approach is to perform the architecture synthesis at a higher level using modules from a library. Such a library may contain different implementations of adders, multipliers, floating point units, ALUs, address generators, and other similar functional units. Each module definition in the library consists of an implementation in the technology of interest (in our case, synthesizable Verilog)[4], an ISDL template that describes the operations available from the functional unit, and a description of the interfaces between this functional unit and the rest of the processor.

Figure 3-10 shows the design flow for such a process. Once the candidate architecture has been selected, it is described in terms of its constituent library modules,

[4]Note, however, that library-based synthesis can make use of hand-optimized layouts which may result in much more efficient implementations in terms of area and power consumption.

their interconnection, and possibly glue logic. This description is then fed to an *architecture compiler* which examines the modules in the library as well as the architecture description, and produces a complete implementation in synthesizable Verilog, plus a complete description in ISDL[5]. The synthesizable Verilog can then be used to map to the desired silicon technology, while the ISDL description can be used to drive all of the ISDL-based evaluation tools.

The main advantage of the library-based approach is that it can make use of large, carefully optimized modules. It thus results in very efficient implementations, both in terms of silicon area and in terms of power consumption. A related advantage is that it can map to the desired implementation (e.g., custom silicon layout) *directly*. This is mainly due to the fact that the module definitions can contain *any* type of implementation definition, and they are large enough that they (as opposed to the interconnect) make up most of the implementation. Finally, since the modules are not derived from their ISDL descriptions, the approach is not subject to the resource sharing inefficiencies described in Section 3.3.2.

The following are a list of the major disadvantages of the library-based approach:

- It requires two descriptions: one for the high-level architecture description that is provided as input to the architecture compiler, and the ISDL description that drives all the evaluation tools. These two descriptions must be kept consistent, and only the architecture description can be changed since the ISDL is generated from the architecture description.

- The granularity with which the design space can be explored is very coarse. This is because the only freedom allowed to the designer is which modules are used, how many, and how to connect them. If you want to examine the benefits of adding a complex instruction to one of the functional units, you have to provide a complete library definition for that unit with the new instruction added. This makes it both harder to find a good architecture, as well as less likely that such an architecture will be obtained.

- Because of the above, local changes to the instruction set are no longer possible. Therefore, it is no longer possible to design instruction sets - the designer is forced to design implementations. For applications were code size is just as important as the size of the processor (such as the target set of applications)[6], it is preferable to be able to design instruction sets.

- If the designer is forced to design architectures rather than instruction sets, it is harder to evaluate the effects of changes in the architecture since there is a layer of abstraction between the implementation and both the code and the simulator statistics. It is therefore harder to create improvements to the architecture.

[5]This description is created from the templates in the libraries, as well as information about the interconnect.

[6]Remember that both the program ROM and the actual processor are implemented on a single chip. Code size will consume silicon area and power, just as the processor will.

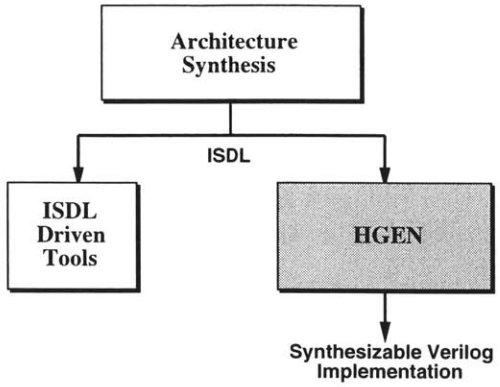

Figure 3-11: **Hardware Synthesis from** ISDL

3.3.2 Hardware Synthesis from ISDL

Figure 3-11 shows our methodology for hardware synthesis. In this methodology, the architecture synthesis system produces instruction sets instead of implementations. The output of the architecture synthesis system is an ISDL description, possibly with some implementation-specific details (such as timing information) missing. This ISDL description is used to drive the ISDL-based evaluation tools, as well as an ISDL-to-hardware compiler (called HGEN). The output of the HGEN compiler is a synthesizable Verilog model which can then be used to create a hardware implementation in any kind of underlying technology. If any implementation-specific information was missing from the original ISDL description, the HGEN compiler will provide it at this time.

The main disadvantage of this approach is that results in less efficient implementations (both in terms of silicon area and power). The primary cause of the inefficiency is the resource sharing problem which is described in Section 3.3.2. Additionally, the ISDL-based approach has to rely on silicon compilers, and therefore it inherits their inefficiencies as well.

The method does, however, have a significant number of advantages:

- There is only a single description to work with. All interfaces are uniform, and no consistency problems exist.
- The granularity with which the design space can be explored is very fine. It is as small as the constituent expressions that are allowed in RTL. This makes it much more likely that a good architecture will be obtained and makes it easier to obtain such an architecture.
- Local changes to the instruction set are possible. In fact, any change in ISDL is reflected in the implementation.
- The design is closer to both the program and the simulator statistics, thus making it easier to detect improvements to the architecture.

- The method abstracts the instruction set design from the actual implementation. This effectively decouples the program ROM from the processor. The instruction set determines the size of the program ROM while the actual implementation of the instruction set (the architecture) determines the size of the processor. This allows an additional degree of freedom (that of instruction set implementation) which makes it even more likely to find good solutions to the design problem.

We feel that direct synthesis from ISDL has compelling advantages, including the fact that it will benefit more from improvements in other CAD tools. Also, the resource sharing problem can be solved using a combinatorial optimization strategy.

The Resource Sharing Problem

The scope of an ISDL operation definition is independent of the scope of any other operation definition. This makes it non-trivial to deduce when hardware resources may be shared by multiple operations.

For example, consider a `move` operation that is implemented using a bus, and `load` and `store` operations that are mutually exclusive with the `move`. The `move` operation resides in a different field than the `load` and `store` operations. A naive scheme would generate additional data-paths to handle the `load` and `store` operations even though it is possible to implement these with the same bus that implements the `move`.

Determining and Synthesizing Shared Resources

We have formulated a way of solving the resource sharing problem to allow ISDL-based hardware synthesis to be used efficiently. First we break up the RTL expressions for all operation definitions into a number of nodes, each of which can be mapped to a circuit. This collective set of nodes (let us say n nodes in total) is numbered with unique numbers from 1 to n. Then we create an $n \times n$ matrix A, with entries that are 1 or 0. A_{ij} is 1 if the nodes can be shared (i.e., they would never operate at the same time), and 0 if they cannot (because they have to operate in parallel). To determine the entries in the matrix we can use the following set of criteria:

1. Nodes that are part of the same RTL statement will have to operate in parallel so they cannot be shared.

2. Nodes that perform completely different tasks (for example a shift and a binary AND operation) cannot be shared. Pairs where one node is a subset of another (e.g., an `add` is a subset of a `subtract`) *can* be shared assuming that the rest of the rules do not prevent it.

3. Nodes belonging to operations in the same field (or to options in the same non-terminal) will *never* be active at the same time so they can be shared.

4. Nodes that belong to operations in different fields will probably have to operate in parallel (since the operations will probably have to operate in parallel) so they cannot be shared.

```
Label each operation in RTL with an integer

    for each i from 0 to n
        for each j from 0 to n
            A_ij = 0
            if i and j not in same operation
                if i and j functionally equivalent
                    if i and j in operations in same field
                    or constraint between i and j
                        A_ij = 1
        end
    end

Generate maximal cliques for A
Generate hardware for maximal cliques
```

Figure 3-12: **Resource Sharing Algorithm**

In addition to the above, constraints may be able to determine even more nodes that cannot operate in parallel (from Rule 4 above), so more sharing may be available if we take constraints into account. Finally, it may be possible that some nodes can be shared, but we might not want to implement them as such because of other constraints (for example routing and layout concerns, or access to register-file ports).

Once we have the entries in the matrix, we can simply create maximal cliques[7] of the nodes that can be shared. These maximal cliques can be synthesized into circuits and the routing and glue logic can be generated to complete the implementation. Figure 3-12 shows this algorithm in pseudo-code.

Obtaining Structural Information from ISDL

While ISDL is a behavioral language and it contains no explicit structural information, a substantial amount of information about the structure of the underlying architecture can be extracted from various parts of the description.

In particular, the costs and timing information exposes the underlying data-path pipelines to the instruction set. For example, an operation with a `Cycle` cost of 1, a `Stall` cost of 3, and a `Latency` of 1 implies a 4-stage data-path pipeline for the functional unit. Additionally, it implies no bypass logic for this particular operation. Similarly, an operation with a `Cycle` cost of 1, a `Stall` cost of 0, and a `Latency` of 1

[7] A clique is a set of nodes such that for any pair of nodes i and j in the clique, $A_{ij} = 1$. A maximal clique is a clique such that if any node is added to the clique, the resulting set of nodes is no longer a clique.

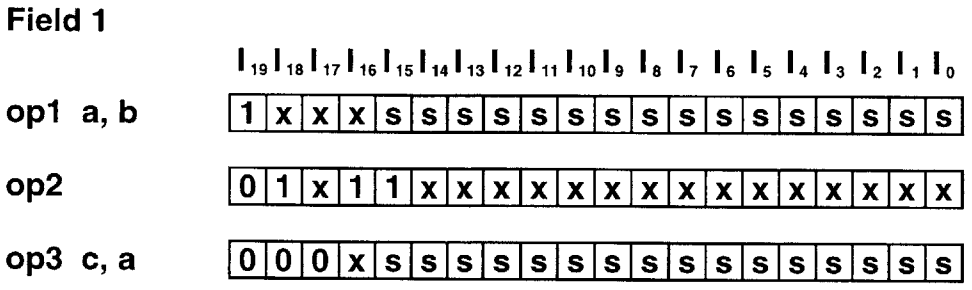

Figure 3-13: **Use of Signatures and Op-Codes for Decode Logic**

implies a similar pipeline with full bypass logic. Finally, an operation with a `Cycle` cost of 1, a `Stall` cost of 0, and a `Latency` of 4 implies a 4-stage pipeline with no protection (stalls or bypass logic) at all.

Similarly, the constraints express hardware restrictions and can therefore be used to deduce the structure of the underlying hardware. Consider the example given earlier of a `move` operation that can move data between various storage elements, and a constraint that shows that this operation cannot occur in parallel with `load` and `store` operations. It is then possible to connect the memory to the same bus as the `move` operation and avoid creating a new set of data paths for the `load` and `store` operations.

3.3.3 Generating Decode Logic

We can generate a complete implementation of the decode logic using the same approach we use to generate the disassembler for the GENSIM system:

For each operation in a field we define a decode line which will be active if the operation is instantiated in the current instruction. We can then derive an equation for each decode line by simply examining the constants in the operation signature. For example, the equation for the operation `op2` in Figure 3-13 is $\bar{I}_{19}.I_{18}.I_{16}.I_{15}$. This results in a a very efficient two-level implementation. Similarly, logic can be generated from the backward pass that reverses parameter encodings (see Section 3.2.2. Finally a set of multiplexers and glue logic completes the decode circuit.

3.4 Previous Work on Simulator and Hardware Model Generation

This section presents work by other researchers related to automatic generation of simulators and hardware models.

3.4.1 LISA

The LISA[38] language was developed as a machine description language specifically designed to support the generation of simulators for generic architectures. It results

in very fast compiled-code simulators, that are cycle-accurate and bit-true. LISA contains a lot of structural information and can model most of the complicated timing effects that are likely to be encountered in embedded applications.

LISA is very effective at generating good simulators. Given the structural content in a LISA description, hardware generation should also be possible although we are unaware of any publications describing such a system. However, LISA is not well suited for generating code-generators and assemblers. If it was used in a system such as ours, a separate language would have to be used for code generation, thus resulting in consistency issues as well as making it harder to generate, describe, and evaluate architectures.

3.4.2 RADL

RADL[34], with its very detailed timing model is particularly well-suited to the generation of simulators and hardware models: it results in very fast ILS simulators and efficient hardware models. However, it suffers from a coupling of the ISA and implementation that can obstruct ISA development. Additionally, it is not well-suited for code generation. Therefore, a system employing RADL may have to employ an additional machine description language in order to support all design evaluation tools.

3.4.3 PLAYDOH

PLAYDOH[23] is a parameterized architecture model used with the TRIMARAN compiler system and the IMPACT project from the University of Illinois. Retargetable simulators exist for the PLAYDOH architecture which take descriptions in HMDES[13]. Also, hardware models can be generated from a parameterized VHDL model. However, note that this is equivalent to the library synthesis process of Section 3.3.1. Granularity in PLAYDOH/HMDES systems is coarse, thus reducing the probability that a good architecture will be found and making it harder to locate such an architecture.

3.4.4 CHESS/nML

The nML machine description language[10] is a high-level machine description language that can be used to support automatically generated tools. It was used in the CHESS[24] system for retargetable code-generation as well as a variety of other tools[9]. nML is very similar to ISDL in that it is a behavioral language based on attributed grammars. The main difference between nML and ISDL is in the way constraints are handled. nML can only describe valid instructions. Therefore, it must work around invalid combinations by using additional rules, to describe interactions between operations. Thus, nML descriptions are longer and less intuitive than ISDL descriptions. It is also unclear how well suited nML would be for hardware generation, since the constraints provide a lot of structural information used to generate

efficient hardware. Finally, it cannot describe some architectures that have complex assembly functions.

3.4.5 MIMOLA

The MIMOLA[27] design system was created as a high-level design environment for hardware, based on the MIMOLA hardware description language[37]. Later on, the system evolved to a hardware-software co-design environment. The MIMOLA system includes a simulation environment. However, the system was designed for development and evaluation of implementations at a much lower level than ISDL. The MIMOLA language is a structural description at a relatively low level, and thus results in unnecessarily long and complex descriptions. At the same time, the lower level of detail results in slower simulators (similar to simulation models written in Verilog). On the other hand, the low-level detail makes it much easier to synthesize hardware from the descriptions.

3.4.6 FLEXWARE

FLEXWARE[30] is a software-firmware system for the development of custom ASIPs and commercial processors. It was developed specifically to support the development of DSP processors and embedded system software. It consists of the code-generator CODESYN and the simulator INSULIN. The FLEXWARE system can be used to rapidly evaluate architectures.

FLEXWARE suffers from the fact that it uses two different machine descriptions for the code-generator and the simulator. This raises consistency issues and makes the work of generating tools for a given architecture harder. It is unclear whether the system is well suited to hardware generation; there are no publications describing attempts to implement such a system.

3.4.7 UPFAST/ADL

UPFAST[35] (University of Pitsburg Flexible Architecture Simulator Tool) is a simulator generator tool for generating micro-architecture simulators. The generated simulators are cycle-accurate to the micro-architecture level and run at about 100-200K/cycles per second. The machine description for the UPFAST system must be provided in ADL (Architecture Description Language). ADL is mainly geared towards RISC and CISC style processors and does not seem to have support for instruction-level parallelism. It also has a substantial structural component which tends to make the descriptions unnecessarily long and complicated (RISC architectures take on the order of 5000 lines of ADL). We are not aware of any systems capable of generating a hardware model from ADL although given the amount of structural information in ADL, it should be relatively easy. Despite the low-level structural information present in ADL, the language does make an effort to distinguish the ISA from the implementation and thus can be used for designing ISAs.

3.4.8 TRS-based Systems

TRAC[22] (Term Rewriting Architecture Compiler) is an architecture synthesis tool based on Term Rewriting Systems (TRS). TRS's are especially well suited to describing high-level behavior. TRAC takes a machine description in a TRS-based language and generates an implementation for it. This implementation could form the hardware model. Additionally, the original TRS description can serve as the simulation environment using appropriate TRS software. TRS's are very flexible and can describe a wide variety of architectures and architectural features. Unfortunately, they are not well-suited to the task of expressing implementation details. Constraints cannot be described naturally in a TRS-based system and assembly functions are missing from TRAC (they are automatically generated but inefficient).

Marinescu and Rinard[26] also describe a system which will generate an implementation given a description in a TRS-based language. Once again, the implementation can serve as the hardware model while the TRS description itself can serve as the simulator. Like the TRAC system, their TRS-based compiler does not provide any support for assembly functions - it automatically generates these in a sub-optimal way. However, it has the ability to support a wide range of architectures and architectural features and can automatically generate pipelined hardware for the candidate architecture.

Chapter 4

Results

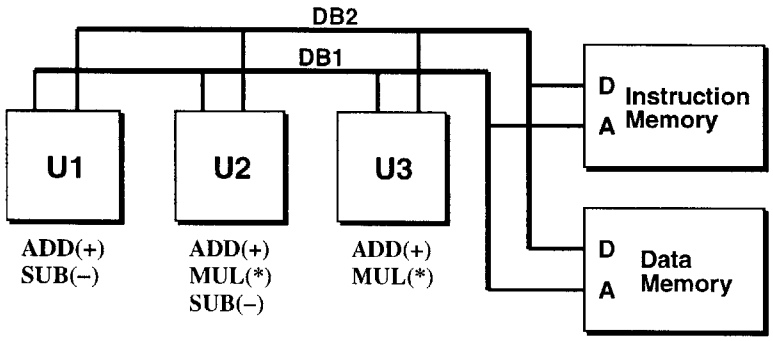

Figure 4-1: **The SPAM VLIW-1 Architecture**

This section presents some experimental results that were obtained using our system and analyzes these results to demonstrate the feasibility of the proposed process and the effectiveness of ISDL and the implemented tools.

4.1 Experimental Results on ISDL Descriptions

A number of architectures were described in ISDL in order to prove the power and flexibility of the language as well as to serve as experiment platforms for the results presented in Sections 4.2 and 4.3. The descriptions of four of these architectures are provided in Appendix C. These four architectures are:

1. **SPAM VLIW-1:** The VLIW-1 architecture is a 5-way VLIW architecture with three data processing units and two transfer buses. In addition, the architecture provides a separate instruction and data memory, either of which may be used to store data. Each data processing unit is very simple and can do a limited number of operations. Each data processing unit also contains an internal register file of 4 registers. The two transfer buses can be used in parallel to execute two register-file-to-register-file transfers in parallel, or can be used together to perform either a data memory transfer or an instruction

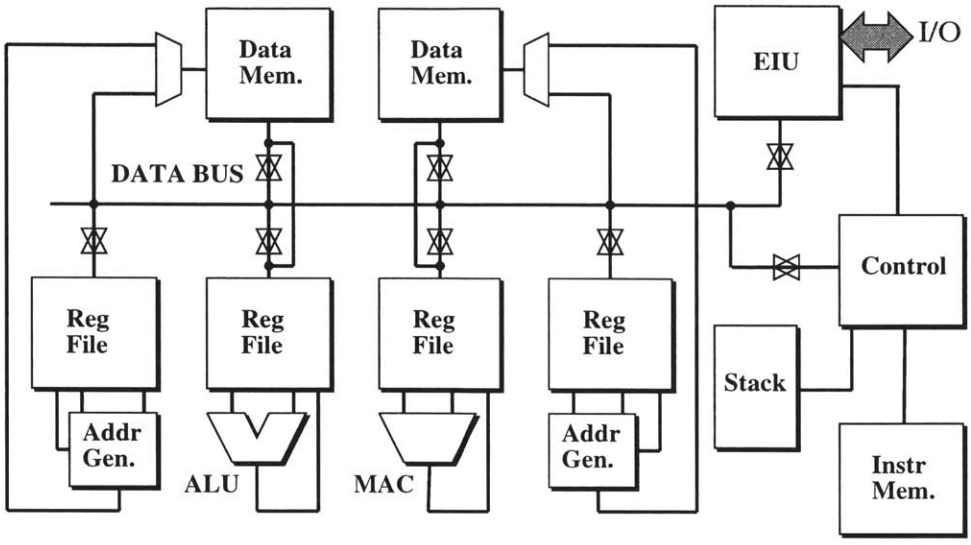

Figure 4-2: **The SPAM VLIW-2 Architecture**

memory transfer. In the case of memory transfers, one bus provides the address to the corresponding memory and the other performs the actual transfer. All data-paths are 16 bits wide and the instruction word is 44 bits wide. A block diagram of this architecture is shown in Figure 4-1.

2. **SPAM VLIW-2:** The VLIW-2 architecture is a 7-way VLIW architecture. It consists of a floating-point data processing unit, an integer data processing unit, a data memory for each unit, an address generator of each data memory, a transfer bus, an external interface unit, and a control unit. The control unit has a separate instruction memory and a hardware stack. The floating point unit contains its own own register file of 16 registers and an internal accumulator register. It is capable of all common floating point operations. The integer data processing unit can perform any arithmetic or logical operations and contains its own register file with 32 registers. Each address generator contains a dedicated register file with 8 registers and can perform simple indexing and incrementing operations. The two data memories can transfer data either over the bus or directly to their respective processing unit. If the data bus is not being used by a data memory, it can be used to transfer data between the registers files of the various or units. All data paths are 32 bits wide and the instruction word is 99 bits wide. A block diagram of this architecture is shown in Figure 4-2.

3. **SPAM RISC:** The RISC architecture is a small architecture implemented in the same spirit as most existing RISC processors (e.g., the Sun Microsystems SPARC processor). It contains a single data processing unit and a register file containing 32 registers. Register 31 is used to store the PC on calls and register 30 is by convention the stack pointer. The architecture supports all common data processing operations. All data-paths are 24-bits wide and the instruction

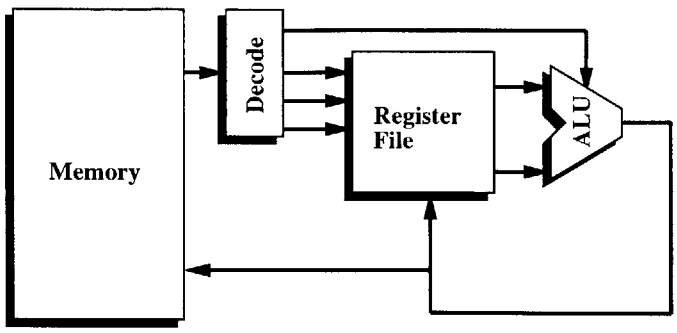

Figure 4-3: **The SPAM RISC Architecture**

Architecture	SPAM VLIW-1	SPAM VLIW-2	SPAM RISC	56000
Class	VLIW	VLIW	RISC	DSP
Lines of ISDL	322	972	329	2286
Man-Hours to Generate	14	50	8	36
Fields	8	8	1	2
Operations	28	81	42	152
Non-Terminals	13	25	5	58
Constraints	6	16	0	43
Reg. Aliases	No	No	No	Yes
Heavy Op-code Encoding	No	Yes	No	No
Op-code Prec.	Yes	Yes	No	Yes

Table 4.1: **Summary of ISDL Descriptions of Base Architectures**

word is 24-bits wide also. A block diagram of this architecture is shown in Figure 4-3.

4. **56000:** This is a description of the Motorola 56000 series DSP. It contains two 56-bit accumulators and two data registers each of which may be used as two 24-bit registers or one 48-bit register. It supports two data memories, each of which is 24-bits wide and which can be combined to a single 48-bit wide memory. It also contains three register files each of which contains 8 registers; these are used to support a rich set of addressing modes. The architecture supports common DSP data processing operations and up to two transfers in parallel with a data operation. The data paths vary in width according to the operation being executed and can be 24, 48, or 56 bits wide. The instruction is 24 bits wide. A block diagram of this architecture is shown in Figure 4-4.

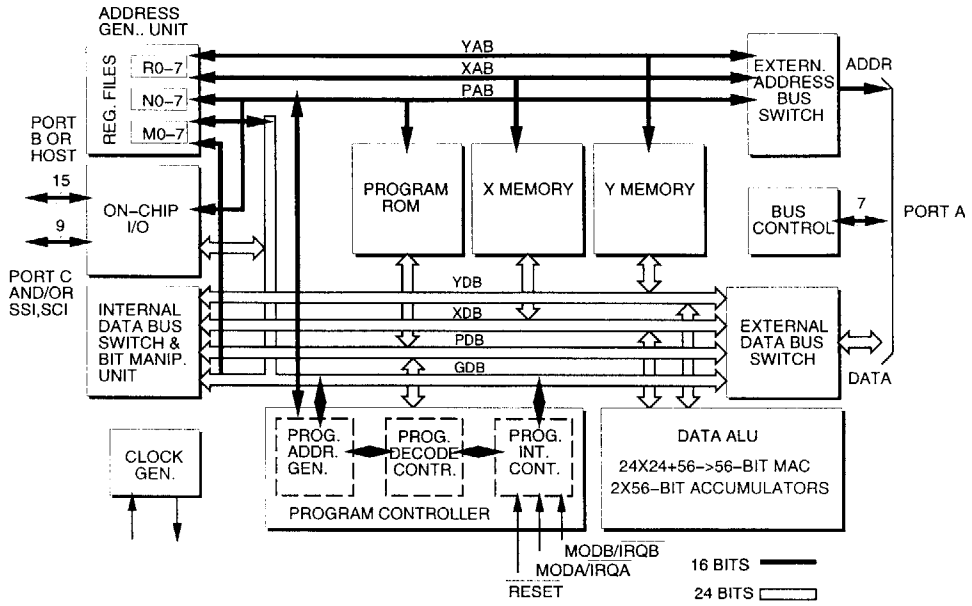

Figure 4-4: **The Motorola 56000 Architecture**

Table 4.1 summarizes the main features of the descriptions. These four architectures represent a broad spectrum ranging from simple RISC style machines to commercial DSP processors, to highly-aggressive VLIW designs. Each architecture listed above was also used to generate several modified designs that were used to obtain the experimental results presented in Sections 4.2 and 4.3. A number of the modified versions of the above architectures (except the 56000 which is already pipelined) were pipelined and provided with a range of timing-related features ranging from unprotected pipelines, to bypass logic, to stalls and delay slot instructions. This shows that ISDL is capable of describing a wide variety of architectures, both in terms of the classes of architectures it will allow and in terms of the features within each architecture.

Note that most architectures resulted in particularly concise descriptions ranging between 300 to 900 lines. The exception is the Motorola 56000 description. This architecture has a very heavily optimized instruction set with a very large number of operations; in fact the encoding of the instruction set appears to have been Huffman-encoded to allow as many operations as possible to be included in the 24-bit word. Additionally, many operations have multiple formats which are expressed as different operation definitions in the ISDL description. The result is a very large number of operations resulting in a large description. Furthermore, the 56000 description makes very heavy use of side-effects statements to set the condition code bits, with multiple such statements occurring in each operation definition. Note, however, that despite the large size of the description, it was developed in about 36 man-hours. This illustrates how easy ISDL descriptions are to develop.

Furthermore, several modified versions of each architecture (except the 56000) were created. Modifications ranged from the addition of a few operations, to the addition of up to 8 times the existing number of operations. Some modifications also involved a complete change in timing model for the associated architecture. Once again, modifications were simple to implement and local to each operation affected. In the cases where new operations created resource conflicts with existing operations, constraints were simply added to the architecture description. Most modifications were completed in less than 1 man-hour. This illustrates that ISDL maintains the orthogonality property of instruction sets (modulo the addition of constraints) and that modifications are easy to implement.

Finally, the descriptions were used to generate functional tools, verifying that ISDL contains sufficient information to generate the required tools.

4.2 Experimental Results on Simulator Generator

This section presents experimental results related to the GENSIM system. The following are the parameters of interest:

- **Cycle-accuracy:** The ability to maintain correct cycle-counts (including stall cycles) and to ensure that results are written back when expected in the case of unprotected pipelines, as described in the costs and timings expressions of the input description.
- **Range of Architectures:** The ability to handle a wide variety of architectures and, in particular, all the architecture classes that form the target of the ARIES system.
- **Support for Architecture Features:** The ability to generate correct behavior for common architectural features such as zero overhead loops, setting of condition codes, heavy op-code encoding, field precedence, etc.
- **Generation Speed:** The amount of time it takes to process an ISDL description and to generate a fully functional simulator from it.
- **Simulation Speed:** The speed at which the generated simulator runs on sample programs.

Cycle-accuracy is guaranteed by construction. However, to verify that the behavior of the generated simulators is correct, a number of architectures and modifications were run on sample programs and the behavior and cycle counts were compared to those expected. The generated simulators do indeed maintain cycle-accuracy to the instruction level (i.e., will maintain any timing semantics that are visible to the instruction set).

The simulator was used to describe a wide range of architectures with a number of common architectural features. The simulator generator successfully generated simulators for all of these, proving that the algorithms are flexible enough to handle

Base	VLIW-1			VLIW-2			SPAM RISC					56000
Version	a	b	c	a	b	c	a	b	c	d	e	a
Class	VLIW			VLIW			RISC					DSP
ILP	x	x	x	x	x	x						x
Zero-Overhead Loops	x	x	x	x	x	x						x
Load/Store	x	x	x	x	x	x	x					
Direct-memory modes								x	x	x	x	x
Complex Instructions			x	x	x	x						x
Condition Codes												x
Hardware Stack				x	x	x						x
Harvard Architecture	x	x	x	x	x	x			x	x	x	x
Address Generators				x	x	x						x
Floating Point				x	x	x						
Stalls										x	x	
Delay Slots											x	
Heavy op-code Encoding				x	x	x						
Field Precedences	x	x	x	x	x	x						x
Register Aliasing												x
Predicated Execution												x

Table 4.2: **Architectures Used for Simulator Generation**

Base	Version	Heavy Op-code Encoding	Precedences	Word (bits)	Generation Time (sec)
SPAM VLIW-1	a		x	44	< 1
	b		x	44	< 1
	c		x	44	< 1
SPAM VLIW-2	a	x	x	99	2.2
	b	x	x	99	2.3
	c	x	x	99	2.6
SPAM RISC	a			24	< 1
	b			24	< 1
	c			24	< 1
	d			24	< 1
	e			24	< 1
56000	a		x	24	181.27

Table 4.3: **Simulator Generation Speed Results**

both a wide range of architectures as well as a wide variety of commonly implemented features. Table 4.2 shows a summary of the architectures for which simulators were generated and lists the architectural features for each one.

Generation speed was also measured for all of the above architectures. Table 4.3 summarizes the results obtained from the various architectures we examined. All run-times were measured on a Sun Microsystems Ultra-10 workstation with a 333 MHz processor, running Solaris 2.6. Most of the time spent on generating the simulator is actually spent determining op-code uniqueness. Note that despite the worst-case exponential run-time, our heuristics ensure that an exponential search is almost never used and therefore the run-times are very reasonable. Both the SPAM VLIW-1 architectures and the SPAM VLIW-2 architectures resort to a non-terminal expansion phase and the resultant search, but the heuristics make sure that the exponent is low (n^2) and each of the fields involved has only a few operations in it. The one exception is the case of the 56000 description. A number of factors conspire to make the exponential search unavoidable:

- The op-codes for operations within the same field are not in a constant place and therefore non-terminal expansion is necessary to resolve almost every single case.
- Since there are only two instruction fields in the description and one of them sometimes uses all the bits in the instruction word, the op-code subtraction heuristics do not perform well.
- Each field has a very large number of operations which means that the base in the exponential equation is really large.

Base	Version	Fetch Cycles ($10^3/sec$)	Operations ($10^6/sec$)
SPAM VLIW-1	a	503	4.0
	b	563	4.5
	c	328	2.6
SPAM VLIW-2	a	292	2.3
	b	297	2.4
	c	452	3.6
SPAM RISC	a	1193	1.2
	b	1241	1.2
	c	1204	1.2
	d	1061	1.1
	e	1169	1.2
56000	NA	46.19†	0.092†

Table 4.4: **Simulation Speed Results**

Despite the above factors, the GENSIM system can generate a simulator in a reasonable time even for the 56000 architecture[1].

We also measured the simulation speed for all of the simulators generated by GENSIM. The results are shown in Table 4.4. All run-times were measured on a Sun Microsystems Ultra-10 workstation with a 333 MHz processor, running Solaris 2.6. Note that simulation speeds for these example architectures range from 1.1 M to 4.5 M operations per second. These simulation speeds are high enough that realistic data samples can be run with the full application, thus increasing the accuracy of the evaluation process. Also note the anomalies in the above figures. While in terms of the fetch rate the five versions of the SPAM RISC were the top performers, in terms of the actual operations per second, these were the worst performers. This gives an indication of the overhead of the scheduler and the write-back mechanism. This overhead is shared over 8 fields in the two VLIW architectures while it is incurred by every operation of the RISC architecture. Also note the abnormally high performance of Version c of the SPAM VLIW-2 architecture. We believe this to be due to a better fit in the super-scalar issue mechanism of the workstation processor because of the reduced number of jumps at the end of the loop. We believe that the abnormally low speed of Version c of the SPAM VLIW-1 architecture is due to the extra computation that needs to happen for the load-and-increment and store-and-increment instructions. Finally, the abnormally low speed of the simulator for the 56000 architecture is an artifact of the compiler used to compile the generated simulators. The simulator for the 56000 was so large that the compiler could not perform optimizations on it[2].

[1]Note that in all the cases listed in Table 4.3, generating the simulator source code took less time than compiling the simulator source code with full optimization flags.

[2]If we enabled compiler optimizations then the compiler would crash due to an internal error.

The code was compiled without optimizations which lead to substantially reduced speed[3].

4.3 Experimental Results on Architecture Exploration

In order to investigate the ease and effectiveness of using ISDL to explore the architecture design space, we coded some applications for three of the four base architectures and evaluated them using our simulator. Using the results from the simulation runs, we modified each architecture repeatedly and evaluated it again. Because of the lack of a full-featured compiler, we had to resort to hand-coded assembly for the applications. This means that the applications are small but the process is representative of what could be obtained had a compiler been available. Additionally, because of the lack of a full-featured hardware model generator, we had to rely on human intuition to investigate the effects of ISDL modifications on the physical costs of each design.

The three base architectures that were used for the architecture exploration experiments were the SPAM VLIW-1, the SPAM VLIW-2 and the SPAM RISC architecture. We did not explore possible improvements to the Motorola 56000 for the following reasons:

- The 56000 makes full use of all the op-codes which means some operations must be removed if others are to be added.
- The 56000 has a Huffman-encoded instruction set which makes it really hard to add new operations. The operation to be removed must have exactly the same binary footprint as the one inserted which means the choice in which operations get removed is restricted.
- The 56000 Instruction Set is already heavily optimized which means opportunities for optimization would be hard to locate without access to a full-featured compiler and a full-featured hardware model generator.

The metrics of interest in all of the experiments was code size and performance.

4.3.1 Experiments Regarding Data Processing

The first set of experiments was concerned with ways of making data processing more efficient. The SPAM VLIW-1 and SPAM VLIW-2 architectures were used as base architectures for this sequence of experiments.

The sequence was initiated by coding an 8-tap FIR and an 8-tap IIR filter on the SPAM VLIW-1 Version a architecture. The description for this architecture is shown in Appendix C.1. The assembly program for the FIR filter is shown in Appendix

[3]The generated code is very inefficient but easy for the compiler to optimize - we count on compiler optimizations to yield the true performance our simulators are capable of.

```
/*
 * A simple FIR filter
 */

/*
    Memory Map
    Writeback loc   IM[1]
    Outer Loop Cnt  IM[2]
    coefficients    IM[3]-IM[10]
    Samples         DM[0]-DM[1023]
    Output          DM[2048]-DM[3072]
*/
...
    /*
     * this is where the coefficients should be - these will be loaded
     * later from a file - unfortunately our assembler does not support
     * dot-notation yet. Let's just create some blank holders
     */
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
...

  { U1_nop;
    U2_mul U2.R0, U2.R1, U2.R2;
    U3_nop;
    DB1_nop;
    DB2_nop;
    DM_NULL;
    IM_NULL;
    C_NULL; }
...
```

Figure 4-5: **Part of the 8-Tap FIR for the SPAM VLIW-1 Version a**

```
Split.Int          U1.OP + U1.RA + U1.RB + U1.RC +
                   U3.OP + U3.RA + U3.RB + U3.RC;

U2_mulc U2_RA, INT, U2_RC
        { U2.OP = 0x1; U2.RA = U2_RA;
          Split.Int = INT; U2.RC = U2_RC; }
        { U2_RC <- MULm(U2_RA, INT); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
```

Figure 4-6: **Modifications to Derive SPAM VLIW-1 Version b**

D.1.1[4]. A portion of that application is shown in Figure 4.3.1. The first thing to note in the code shown in Figure 4.3.1 is that the coefficients are stored in locations 3-10 in the Instruction Memory and are loaded from this memory when needed. If the operation that uses the coefficients could load them as immediate constants instead, the following advantages would obtained:

1. The load operation for the coefficient can be removed completely. This would result in smaller code and faster performance.

2. The coefficients are 16-bit constants but they are stored in 44-bit instruction words thus wasting a substantial amount of space.

3. The updates to the coefficient location can be removed thus improving both code size and performance.

The architecture was modified to include a multiply operation that can take an immediate constant as a parameter. These modifications are shown in Figure 4-6. The application was then re-written to make use of this new operation. In order to do this, the inner loop had to be unrolled thus increasing code size. However, this increase is offset by the reduction in code size stemming from the fact that the coefficients are no longer placed in individual memory locations. Additionally, a substantial performance increase was obtained. Once these modifications were made, the result was simulated and exact cycle-count figures were obtained. These indeed showed a substantial performance increase.

However, there was still room for improvement. Figure 4-7 shows a portion of the inner loop for the modified FIR program. This portion of the program shows one stage of the unrolled loop. The second instruction performs the accumulation step inherent

[4]The implementation of the IIR filter is very similar.

```
{ U1_NULL;
  U2_mulc U2.R0, H1, U2.R1;
  U3_NULL;
  DB1_NULL;
  DB2_NULL;
  DM_ld U2.R0, U1.R0;
  IM_NULL;
  C_NULL; }

{ U1_addc U1.R0, 1;
  U2_add U2.R1, U2.R2, U2.R2;
  U3_nop;
  DB1_nop;
  DB2_nop;
  DM_NULL;
  IM_NULL;
  C_NULL; }
```

Figure 4-7: **Part of the 8-TAP FIR for the SPAM VLIW-1 Version b**

in any FIR implementation and also increments the location from which the current sample is obtained. This instruction can be completely eliminated, resulting in an increase in performance and reduction in code size, if the instruction set contained instructions to:

- perform a multiply-accumulate as a single step,
- auto-increment the sample address when it is loaded.

Both of these are features commonly found in DSP architectures. The SPAM VLIW-1 Version b was modified to include these instructions (see Figure 4-8) resulting in the SPAM VLIW-1 Version c. The FIR application was re-written for the new architecture and simulated, verifying the expected increase in performance and the reduction in code size. Table 4.5 shows a summary of the results obtained using this process for both applications, running on 1024 samples each.

A similar process was followed for the SPAM VLIW-2. The applications were once again an 8-tap FIR filter and an 8-tap IIR filter, both using the floating point unit of the architecture. The description for the base architecture (Version a) is shown in Appendix C.2, and the assembly program for the FIR filter is shown in Appendix D.2.1. A portion of that application is shown in Figure 4-9. Figure 4-9 shows part of the code that loads the coefficients into registers of the floating point unit. These registers can then be used when multiplying the coefficients with the

```
U2_mac U2_RA, INT, U2_RB, U2_RC
        { U2.OP = 0x1; U2.RA = U2_RA; U2.RB = U2_RB;
          Split.Int = INT; U2.RC = U2_RC; }
        { U2_RC <- ADDm(MULm(U2_RA, INT), U2_RB); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }

DM_ldi  REG, LOC
        { DB1.SRC = DMdata; DB1.DEST = REG;
          DB2.SRC = LOC; DB2.DEST = DMaddr; }
        { REG <- DM[LOC]; LOC <- ADDm(LOC, 1); }
        {}
        { Cycle = 1; Size = 1; Stall = 1; }
        { Latency = 1; Usage = 1; }
DM_sti  REG, LOC
        { DB1.SRC = REG; DB1.DEST = DMdata;
          DB2.SRC = LOC; DB2.DEST = DMaddr; }
        { DM[LOC] <- REG; LOC <- ADDm(LOC, 1); }
        {}
        { Cycle = 1; Size = 1; Stall = 0; }
        { Latency = 1; Usage = 1; }
```

Figure 4-8: **Modifications to Derive SPAM VLIW-1 Version c**

Version	Conversion Type	Application	Fetch Cycles	Clock Cycles	Code Size
a	NA	FIR	44840	65338	19
		IIR	39981	58433	24
b	Immediate Mode	FIR	18456	18459	23
		IIR	18456	18459	23
b	Complex Instr. Auto-increment	FIR	11282	17435	17
		IIR	10257	17435	16

Table 4.5: **Results of Architecture Exploration on SPAM VLIW-1**

```
/* start by loading the coefficients in the registers */
main:  {Control_NOP;
        ALU_IDLE;
        MAC_IDLE;
        DB_move H0, MAC.R0;
        AG1_NULL;
        AG2_NULL;
        DM1_idle;
        DM2_idle; }

       {Control_NOP;
        ALU_IDLE;
        MAC_IDLE;
        DB_move H1, MAC.R1;
        AG1_NULL;
        AG2_NULL;
        DM1_idle;
        DM2_idle; }
...
```

Figure 4-9: **Part of the 8-Tap FIR for the SPAM VLIW-2 Version a**

```
Split.DATAs       ALU.OP+ALU.RW+ALU.RA+ALU.RB+ALU.RMEM
                  +MAC.RB+MAC.RW;

MAC_macc      MAC_RA, INT              { MAC.OP = 0x0 ;
                                         MAC.RA = MAC_RA ;
                                         Split.DATAs = INT ; }
                       { ACC <- FADDm(ACC, FMULm(INT, MAC_RA)) ;}
                            {}
                            { Cycle=1; Size=1; Stall=0; }
                            { Latency=1; Usage=1; }
MAC_maccw     MAC_RA, INT, MAC_RW      { MAC.OP = 0x1 ;
                                         MAC.RA = MAC_RA ;
                                         Split.DATAs = INT ;
                                         MAC.RMEM = MAC_RW ; }
                      { ACC <- FADDm(ACC, FMULm(INT, MAC_RA)) ;
                     MAC_RW <- FADDm(ACC, FMULm(INT, MAC_RA)) ; }
                            {}
                            { Cycle=1; Size=1; Stall=0; }
                            { Latency=1; Usage=1; }
```

Figure 4-10: **Modifications to Derive SPAM VLIW-2 Version b**

actual samples. However, the instruction word to do so is 99 bits long but only effectively loads a 32-bit value into a register (because of the lack of parallelism at this stage of the program). If instead, the coefficients could be supplied as immediate constants, then these instructions could be avoided. In addition, the 32-bit immediate data in this architecture is stored in the bitfields normally associated with the two address generators. Since these would be needed when loading the offsets into the address generators themselves and during the multiplication step which updates the sample offset and performs the memory access, it would be better to store 32-bit immediate data somewhere else in the 99-bit instruction. We can observe from the application that when an immediate constant is needed, the `ALU` field and and the `MAC.RB` and `MAC.RW` fields are not used so they could be used to store the constant instead. The architecture was thus modified in two ways:

1. The `MAC_mul` and `MAC_mac` operations were modified so that they take one of their parameters as an immediate 32-bit constant,

2. Immediate 32-bit constants are now held in the `ALU`, `MAC.RB` and `MAC.RW` subfields.

These modifications are shown in Figure 4-10. The application was then re-written to make use of the modifications. The resulting program was simulated to verify

```
/* break out of the loop if you are done */
       {Control_brcz AG2.R3, stop;
        ALU_IDLE;
        MAC_clr;
        DB_IDLE;
        AG1_idle;
        AG2_NULL;
        DM1_idle;
        DM2_idle; }
/* loop back otherwise */
       {Control_jump;
        ALU_IDLE;
        MAC_IDLE;
        DB_NULL;
        AG1_idle;
        AG2_inc AG2.R1, AG2.R1;
        DM1_idle;
        DM2_dir_save_m MAC.R8, AG2.R1; }

stop:   HALT_INST
```

Figure 4-11: **Part of the 8-TAP FIR for the SPAM VLIW-2 Version b**

correctness and to show that there is no degradation in performance.

Further optimizations were sought out in Version b of the architecture. Figure 4-11 shows the end-of-loop code for the modified FIR program. Because this architecture only provides a branch-on-zero instruction, the end-of-loop has to be written using two instructions. We decided to provide a branch-on-not-zero instruction that redirects control flow back to the beginning of the loop thus saving one instruction for every loop, resulting in denser code and better performance. Figure 4-12 shows this modification which resulted in the SPAM VLIW-2 Version c. The FIR application was re-written for the new architecture and simulated, verifying the expected increase in performance and the reduction in code size. Table 4.6 shows a summary of the results obtained using this process for both applications, running on 1024 samples each.

4.3.2 Experiments Regarding Control Flow

While the results of the previous section showed that substantial benefits can be obtained by using an architecture exploration methodology for data-dominated applications, we wanted to investigate if similar results can be obtained for control-

```
Control_brnz RI          { Control.OP = 0xB ;
                           Control.RI = RI ; }
                         { if (RI != 0)
                             { PC <- JR ; }; }
                         {}
                         { Cycle=1; Size=1; Stall=0; }
                         { Latency=1; Usage=1; }
```

Figure 4-12: **Modifications to Derive SPAM VLIW-2 Version c**

Version	Conversion Type	Application	Fetch Cycles	Clock Cycles	Code Size
a	NA	FIR	11280	11280	25
		IIR	11282	11282	27
b	Immediate Mode	FIR	11269	11269	15
		IIR	11270	11270	16
b	brnz	FIR	10236	10236	14
		IIR	10237	10237	15

Table 4.6: **Results of Architecture Exploration on SPAM VLIW-2**

```
int acc(a, c) {
  if (c == 1)
    return a[0];
  return (acc(a, c/2) +
          acc(a + c/2, c - c/2));
}
```

Figure 4-13: **The Array-Accumulate Divide-and-Conquer Algorithm**

flow issues. We decided to investigate one issue in particular; that of supporting calling conventions in hardware. To investigate this we selected a sample divide-and-conquer algorithm (using divide-and-conquer to obtain the sum of all elements in an array), as shown in Figure 4-13.

Because of its recursive nature, this algorithm is a good test for good calling conventions. We used the SPAM RISC Version a architecture as the base architecture for this example. The description for this architecture is shown in Appendix C.3. The complete assembly program for the array accumulate function is shown in Appendix D.3.1. A portion of that code is shown in Figure 4-14.

Note that this code is already optimized in a number of respects. In particular, it only increments the stack pointer at the end of building a frame, it omits the frame pointer, and performs all accesses to the stack as an offset from the un-updated value of the stack pointer. It also only saves registers on the stack when it is absolutely necessary. Figure 4-15 shows the structure of the stack frame to make the program easier to read. This program was simulated on Version a of the RISC architecture to verify correctness and to obtain a reference point for performance.

The first thing to note about the program of Figure 4-14 is that it has to constantly load values from the stack, process them and store them back to the stack. This requires the overhead of load/store instructions which both increase code size and reduce performance. Therefore, the first modification that was implemented was to provide data processing operations which can take their parameters directly off the stack given an offset from the value of the stack pointer, and can store their results in the same fashion. This means that the load and store operations can be removed. Note that since only one value can be loaded from or stored to the memory at a time, the these operations now have to be multi-cycle operations. However, the fetch cycle for the corresponding load or store is still eliminated resulting in performance improvements as well as code size improvements. Figure 4-16 shows two of the new operations as an example.

The array accumulate function was re-implemented using these new addressing modes and simulated verifying both the reduction in code size and the performance improvements. Figure 4-17 shows the new implementation of this function.

By observing the `Cycle` costs for the new data processing operations of Version b of the architecture one will immediately notice that there is a cycle cost associated with the fact that the data is fetched from the same memory which contains the instructions. The data fetch and the instruction fetch compete for the same port in the memory. Thus, it became obvious that a Harvard architecture (i.e., one in which the data is placed in a separate memory with its own port) would provide a substantial performance benefit. Note that describing this in ISDL simply involves changing the costs of the associated operations. This modification was implemented as shown in Figure 4-18. Note that since the instruction set of the processor did not change, the same program as for Version b can be used for Version c as well. The program was simulated on the new architecture verifying the performance gains.

Then we decided to investigate issues involving pipelining of the processor. We decided to simulate a four-stage pipeline with full protection. This means that bypass logic is to be implemented for all data paths and this can include the new address-

```
acc:
   { ldc 2, SP, R1; }
   { subc 1, R1, R2; }
   { brnz R2, LL1; }

   { ldc 1, SP, R0; }
   { ldc 0, R0, R15; }
   { br LL2; }

LL1:
   { stc 4, SP, RET; }
   { ldc 1, SP, R0; }
   { asrc 1, R1, R2; }
   { stc 5, SP, R2; }
   { stc 7, SP, SP; }
   { addc 7, SP, SP; }
   { stc 1, SP, R0; }
   { stc 2, SP, R2; }
   { call acc; }
   { ldc 10, SP, R3; }
   { stc 6, SP, R3; }
   { ldc 1, SP, R0; }
   { ldc 2, SP, R1; }
   { ldc 5, SP, R2; }
   { add R0, R2, R4; }
   { sub R1, R2, R5; }
   { stc 7, SP, SP; }
   { addc 7, SP, SP; }
   { stc 1, SP, R4; }
   { stc 2, SP, R5; }
   { call acc; }
   { ldc 10, SP, R3; }
   { ldc 6, SP, R4; }
   { add R3, R4, R15; }
   { ldc 4, SP, RET; }

LL2:
   { stc 3, SP, R15; }
   { ldc 0, SP, SP; }
   { ret; }
```

Figure 4-14: **Assembly Implementation of the Array-Accumulate Function**

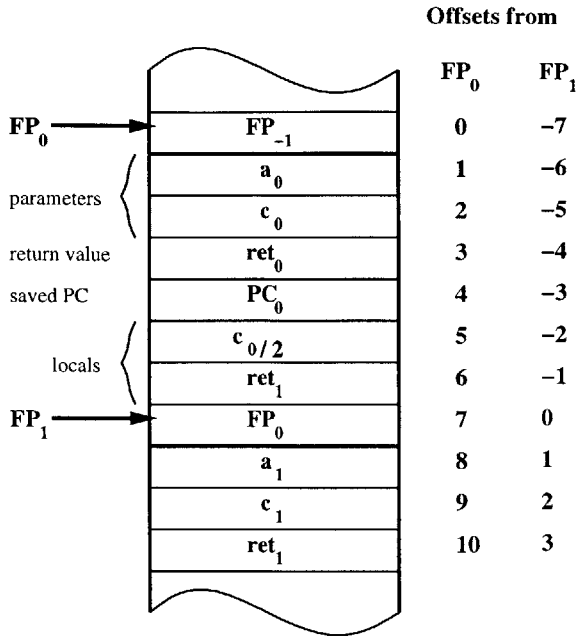

Figure 4-15: **The Stack Frame for the acc Function**

```
...
    add SOFF1, RB, RC
            { W.MODE = 0x4; W.OP = 0x1;
              W.RA = SOFF1; W.RB = RB; W.RC = RC; }
            { RC <- ADDm(IM[SP + SOFF1], RB); }
            {}
            { Cycle = 2; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

    add SOFF1, RB, SOFF3
            { W.MODE = 0x5; W.OP = 0x1;
              W.RA = SOFF1; W.RB = RB; W.RC = SOFF3; }
            { IM[SP + SOFF3] <- ADDm(IM[SP + SOFF1], RB); }
            {}
            { Cycle = 3; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }
...
```

Figure 4-16: **The New Addressing Modes of SPAM RISC Version b**

```
acc:
   { subc 1, 2, R2; }
   { brnz R2, LL1; }

   { ldc 1, SP, R0; }
   { ldc 0, R0, R15; }
   { stc 3, SP, R15; }
   { br LL2; }

LL1:
   { stc 4, SP, RET; }
   { asrc 1, 2, 5; }
   { movs 1, 8; }
   { movs 5, 9; }
   { stc 7, SP, SP; }
   { addc 7, SP, SP; }
   { call acc; }
   { movs 10, 6; }
   { add 1, 5, 8; }
   { sub 2, 5, 9; }
   { stc 7, SP, SP; }
   { addc 7, SP, SP; }
   { call acc; }
   { add 6, 10, 3; }
   { ldc 4, SP, RET; }

LL2:
   { ldc 0, SP, SP; }
   { ret; }
```

Figure 4-17: **Array Accumulate Function for SPAM RISC Version b**

```
...
    add SOFF1, RB, RC
            { W.MODE = 0x4; W.OP = 0x1;
              W.RA = SOFF1; W.RB = RB; W.RC = RC; }
            { RC <- ADDm(DM[SP + SOFF1], RB); }
            {}
            { Cycle = 1; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

    add SOFF1, RB, SOFF3
            { W.MODE = 0x5; W.OP = 0x1;
              W.RA = SOFF1; W.RB = RB; W.RC = SOFF3; }
            { DM[SP + SOFF3] <- ADDm(DM[SP + SOFF1], RB); }
            {}
            { Cycle = 2; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }
...
```

Figure 4-18: **Adjusted Operations of SPAM RISC Version c**

```
...
    brz RC, OFFS
            { W.MODE = 0xC; Split.OFFSs = OFFS; W.RC = RC; }
            { if (RC == 0) { PC <- ADDm(PC, SEXT(OFFS, 15, 24)); }; }
            {}
            { Cycle = (RC == 0)*2 + 1; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

    brnz RC, OFFS
            { W.MODE = 0xD; Split.OFFSs = OFFS; W.RC = RC; }
            { if (RC != 0) { PC <- ADDm(PC, SEXT(OFFS, 15, 24)); }; }
            {}
            { Cycle = (RC != 0)*2 + 1; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }
...
```

Figure 4-19: **Pipeline Flushing in SPAM RISC Version d**

```
...
    brz RC, OFFS
            { W.MODE = 0xC; Split.OFFSs = OFFS; W.RC = RC; }
            { if (RC == 0) { PC <- ADDm(PC, SEXT(OFFS, 15, 24)); }; }
            {}
            { Cycle = 1; Stall = 0; Size = 1; }
            { Latency = 3; Usage = 1; }

    brnz RC, OFFS
            { W.MODE = 0xD; Split.OFFSs = OFFS; W.RC = RC; }
            { if (RC != 0) { PC <- ADDm(PC, SEXT(OFFS, 15, 24)); }; }
            {}
            { Cycle = 1; Stall = 0; Size = 1; }
            { Latency = 3; Usage = 1; }
...
```

Figure 4-20: **Delay Slots in SPAM RISC Version e**

ing modes if the target address is maintained along with the data in the internal data-path. Note however, that we also have to flush the pipeline when control flow operations occur. This is expressed in ISDL by appropriately modifying the `Cycle` cost of any operation that flushes the pipeline. Figure 4-19 shows two examples of conditional branch operations that conditionally flush the pipeline.

Once again, the array accumulate function was simulated on the new architecture without any modifications. As expected, the number of cycles increased due to pipeline flushes but with the expectation that the cycle itself would now be shorter by enough of a margin to offset the cost of extra cycles.

Finally, we decided to investigate the issue of adding delay slot instructions to the architecture in order to reclaim some of the performance lost to pipeline flushes. This involves modifying the `Latency` parameter of control flow operations as shown in Figure 4-20. In this case, the program for the array accumulate function had to be modified to move around control flow operations and include `nop` operations to fill in the delay slots. The new program is shown in Figure 4-21. Note the increase in code size because of the need to insert `nop` operations in some of the delay slots. Simulation of this program showed a marginal performance increase of about 5.4% which may not justify the increased code size (about 13%).

Table 4.7 shows a summary of the experiments on calling conventions.

4.3.3 Overview of Architecture Exploration Results

Table 4.8 presents a summary of the architecture exploration results to give a

```
acc:
   { subc 1, 2, R2; }
   { brnz R2, LL1; }
   { nop; }
   { nop; }

   { ldc 1, SP, R0; }
   { br LL2; }
   { ldc 0, R0, R15; }
   { stc 3, SP, R15; }

LL1:
   { stc 4, SP, RET; }
   { asrc 1, 2, 5; }
   { movs 1, 8; }
   { movs 5, 9; }
   { call acc; }
   { stc 7, SP, SP; }
   { addc 7, SP, SP; }
   { movs 10, 6; }
   { add 1, 5, 8; }
   { sub 2, 5, 9; }
   { call acc; }
   { stc 7, SP, SP; }
   { addc 7, SP, SP; }
   { ldc 4, SP, RET; }
   { add 6, 10, 3; }

LL2:
   { ret; }
   { ldc 0, SP, SP; }
   { nop; }
```

Figure 4-21: **Array Accumulate Function for SPAM RISC Version e**

Version	Conversion Type	Application	Fetch Cycles	Clock Cycles	Code Size
a	NA	Array Acc.	1578	2562	34
b	stack-relative ops	Array Acc.	1070	2171	23
c	Harvard Arch.	Array Acc.	1070	1460	23
d	Pipeline (flush)	Array Acc.	1070	2172	23
e	Delay Slots	Array Acc.	1307	2053	26

Table 4.7: **Results of Architecture Exploration on SPAM RISC**

Architecture	Application	Improvement	
		Code size	Performance
SPAM VLIW-1	FIR	10.5%	253.9%
	IIR	29.2%	235.1%
SPAM VLIW-2	FIR	44.0%	10.2%
	IIR	44.4%	10.2%
SPAM RISC	Array Acc.	32.4%	75.5%

Table 4.8: **Overview of Architecture Exploration Results**

better idea of the benefits that can be gained from the method. The above results show that it is possible to obtain substantial benefits in terms of both code size and performance by performing simple architecture exploration.

Chapter 5

Conclusions

This section summarizes and highlights the main points of the previous sections and presents further work that can be performed to further advance the field.

5.1 Conclusions

The thesis that this work is based on is that architecture exploration can produce architectures that are customized to a particular application while reducing the design cycle over existing methods. In order to support architecture exploration, we claim that it is necessary to provide the following:

- A machine description language that allows instruction sets to be designed.
- A way of generating a number of design evaluation tools automatically from such architecture descriptions, namely:
  - A retargetable compiler that can produce an implementation of the application customized to and optimized for a particular architecture described in an appropriate machine description language.
  - An assembler generator that can generate an assembler given a description of the target architecture in an appropriate machine description language. This assembler can be used to convert the output of the compiler to an executable binary program.
  - A simulator generator that can generate a simulator for an architecture given a description of the architecture in a suitable machine description language. This simulator can execute the assembled program to provide performance and utilization measurements.
  - A hardware model generator that can generate a synthesizable model of the implementation of an architecture given a description of the architecture in a suitable machine description language. This hardware model can be used to derive the physical costs and properties of the architecture and provide an initial implementation.

We further claim that all of the above evaluation tools should be generated from the same machine description in order to avoid consistency issues and the effort of translating from one machine description language to another. The language by which the target architecture is described forms the central component of the system and therefore should be chosen very carefully. We identified the following desirable features that any proposed machine description language should provide:

- Descriptions should be easy to read, write, and modify so that engineers and architects can use the tools without support and so that the final architecture can be further optimized by hand.
- The machine description language should support the re-targeting or generation of all of the above design evaluation tools.
- The language should support as wide a variety of architectures as possible and in particular VLIW architectures which are very efficient for custom applications.
- The language should provide a fine grain of control in order to cover as much of the supported architecture design space as possible.
- The language should separate the description of the architecture instruction set from the actual implementation of the instruction set, in order to leave an additional degree of freedom in finding good architecture candidates.
- Descriptions should contain sufficient information to allow for compiler optimizations and cycle-accurate simulation.

We have presented the ISDL machine description language and have shown that it provides all of the above features. In particular, ISDL has the form of an annotated grammar which is simple to understand. Its use of constraints allows it to describe implementations that result in resource conflicts in a simple and intuitive manner. Its strong semantics make descriptions concise. We have shown that descriptions of simple architectures (such as the SPAM RISC) take little time and effort to develop and that most modifications of these descriptions are local and easy to implement. We have also shown that ISDL supports a wide variety of architectures and architectural features by describing a number of architectures that span classes such as VLIW, DSP and RISC, and contain features as varied as predicated execution, condition codes, Huffman-encoded op-codes, heavy op-code encoding, register aliasing, and complex instructions. We have also shown that constraints help isolate the description of the instruction set from the actual implementation by allowing the description of the instruction sets in a local manner as if operations were completely orthogonal. We have demonstrated the ability of ISDL to support fine-grain modifications by performing both minor and major modifications to the instruction sets of the architectures we described. We have also demonstrated the ability of the language to support the generation of cycle-accurate simulators by generating such simulators automatically and verifying their behavior. Other researchers have demonstrated the ability of ISDL

to support the automatic generation of compilers, assemblers and efficient hardware models.

For the simulator generator tool, we identified the following requirements:

- The simulator generator must support a wide variety of architectures and a wide range of architectural features.
- It must be able to generate simulators for architectures with complex assembly functions without additional support (e.g., requiring inclusion of the decode function in the machine description).
- It must process machine descriptions and produce a simulator in a reasonable amount of time.

We have described the implementation of a tool that automatically generates cycle-accurate simulators from machine descriptions in ISDL. We have shown that this tool, while not flexible enough to deal with all architectures that can possibly be described in ISDL, supports a wide variety of architectures and architectural features in its own right. In particular, we have been able to generate fully functional, cycle-accurate simulators for all the architectures we described. Simulators can be generated very fast for most cases, and in a reasonable time in the worst case. The algorithms embodied in the simulator generator tool are sufficiently intelligent to support even complicated assembly functions without the need to provide the decode function in the machine description.

The generated simulators themselves were required to provide the following:

- Cycle-accuracy and full compliance with the semantics of ISDL.
- High simulation speeds.
- A full-featured user interface.
- A broad range of debugging support features.
- A means of obtaining utilization statistics from each execution.

The simulators that were generated by our tools were indeed shown to be cycle-accurate and bit-true. The simulation speeds range from 1.1 million operations per second to 4.5 million operations per second. At these speeds, the simulators can be used to simulate the full application on a realistic input data sample. The generated simulators also provide both a full-featured graphical user interface and a full-featured command-line interface. They provide full debugging support as well as other features that ease the task of regression testing and supporting long-running simulations. Finally, these simulators can produce execution address traces which can be used to obtain utilization statistics for a particular execution.

Finally, we have demonstrated the feasibility of the architecture exploration concept by performing simple architecture exploration steps assisted by the infrastructure

mentioned above, and demonstrating real and substantial benefits in both code size and performance.

Overall, we have demonstrated the soundness of the thesis by providing an implementation for the infrastructure necessary to perform architecture exploration thus proving architecture exploration to be feasible. We also used this implementation to demonstrate real and substantial benefits in both code-size and performance for particular applications.

5.2 Future Work

The work presented in the previous sections supports the thesis that an architecture exploration system can be created and that it can be effective in creating architectures particularly well suited to a given application. It also presents the implementation of a substantial portion of the infrastructure for such a system. However, a substantial amount of work still remains to be done before the full potential of the methods proposed herein can be unlocked. This section presents some further work that we believe would go along way towards unlocking the full potential of the methodologies we described.

5.2.1 ISDL Version 2.x

While ISDL Version 1 is powerful enough to describe such a wide range of architectures and architectural features, there are a number of additional facilities that could be beneficial. Some of these facilities were considered for Version 1 but originally postponed to Version 2 in order to produce a working language in a reasonable amount of time. Others were discovered only after a range of experiments had been run and additional requirements became clear. This section presents all these possible additions to ISDL in the hope that they will be included in a future version.

Cache Information and Memory Management

ISDL Version 1 does not have the ability to describe what happens in hardware that is not visible to the instruction set and therefore does not take into account the costs and timing related to caches and memory management hardware. Some of this information can be obtained after a simulation by using the execution address trace of the simulator. However, even with that capability, the generated compilers cannot optimize for the presence and structure of caches and memory management systems. Additionally, it would be hard to judge the impact of interactions between the Instruction Set, the implementation of a target application on the Instruction Set and the cache and memory management hardware since the latter has to be investigated in a separate environment. Cache and memory management hardware information should be added to the Optional Information section of an ISDL description.

Multiple-Issue Capabilities

Many architectures contain parallelism which is not explicit in the instruction set. Usually this is in the form of multiple-issue mechanisms where multiple instructions are issued in parallel. The most common implementation of multiple-issue schemes is super-scalar architectures. Since parallelism in such architectures is not explicit in the instruction set, a non-parallelized version of the instruction set can be described and simulated in ISDL Version 1. The exact timing can be derived by post-processing an execution address trace after this is produced by the simulator. However, this still precludes the use of information about the multiple-issue mechanism to optimize compilation.

A large amount of information needs to be provided in order to support multiple-issue execution schemes:

- The number of instructions that can be issued.
- The scheduling policy when some execution units are free and others are occupied.
- The policy for handling data dependencies between instructions.
- The existence of re-order buffers and the policies that govern their operation.
- The presence of speculative execution and the policies that govern its operation.
- The policies concerning branch prediction in the presence of multiple-issue schemes.

While multiple-issue mechanisms are uncommon in embedded systems and are generally unsuitable for environments where the application is known before the architecture is designed, a way of supporting multiple-issue architectures would broaden the scope of ISDL and the related tools and could prove valuable. Such information would be best included in the Optional Information section of an ISDL description.

Exceptions and Interrupts

ISDL Version 1 does not support the notion of exceptions or interrupts. While both are visible to the instruction set, they were omitted from Version 1 due to time constraints.

Both exceptions and interrupts take certain actions that modify the state of the processor when they occur. They can therefore defined in a fashion similar to operation definitions. To define these actions we can create the abstraction of an event that is triggered in the case of a specific exception or interrupt, and performs the above mentioned actions. The following information needs to be provided for each event:

- Input parameters. Even though the semantics of exceptions and interrupts do not allow for parameters, most exceptions perform the same actions with minor modifications in the exception vector they jump to and various mode bits they set. These can be defined as parameters in order to allow the same event definition to be reused for multiple actual exceptions. The same holds for interrupts.

- Event actions. These can be described in the same RTL language that is used to describe operation actions.

- Event side effects. The distinction between event actions and event side-effects is the same as the distinction between operation actions and operation side-effects.

Once the appropriate events have been defined, an exception can be defined by giving the conditions that trigger it (if it applies to all operation instantiations) or by calling an event from an operation definition if it applies only to a single operation (e.g., division by zero). For interrupts, we can define asynchronous signals which trigger the appropriate events with the right parameters.

While exceptions and interrupt definitions are not optional for a full description of an instruction set, in order to maintain backwards compatibility with existing descriptions in ISDL Version 1 it may be appropriate to include them in the Optional Information section of an ISDL description. Backward compatibility can still be maintained if exception and interrupt definitions are placed in the Global Definitions section instead, but at the cost of changing the semantics of the Global Definitions section in a minor way.

State Dependent Constraints

The constraints mechanism provided by ISDL Version 1 is very flexible, however, some constraints cannot be described since they depend on the state of the processor which cannot be accessed by these constraints. In fact, the Motorola 56000 description contains a number of such constraints, mostly having to do with the contents of the `LC` and `LA` registers which control zero-overhead loops.

We propose to expand the constraints mechanism to include relational operations and expressions involving the state of the processor in order to remove this limitation.

Branch Prediction Mechanisms

Branch prediction is a way of avoiding the overhead of pipeline flushes in pipelined architectures by guessing the target of a conditional branch and loading instructions from the target instead of sequentially. Branch prediction mechanisms can be divided in two classes depending on whether the guess is provided by the program itself (hint-driven) or is created dynamically by the hardware at run-time (dynamic). Hint-driven branch prediction is already supported by ISDL Version 1 since it appears in

the instruction set as two otherwise identical operations which differ in their costs depending on whether the branch is taken or not.

Dynamic branch prediction can be described by providing the following information:

- A list of the operations that are subject to dynamic branch prediction.
- A definition of the state that the dynamic branch prediction policy needs to store (such as a flag denoting whether on the last execution of this operation the branch was taken or not).
- An expression which will update the costs of a branch instruction depending on the state of the branch prediction mechanism
- A policy for updating the state for a branch prediction mechanism every time a member of the list of operations subject to this particular policy is executed.

In a completely generic support mechanism, multiple branch prediction policies should be allowed. Dynamic branch prediction definitions should be included in the Optional Information Section of an ISDL description.

Byte-addressed Storage

ISDL Version 1 is word-addressed, meaning that once an address is provided, it will index into the appropriate storage with a stride of 1. However, a number of existing architectures are byte-addressed meaning that the actual index into the storage is the address divided by the width of the storage unit in bytes (a practice which is very common in general purpose processors). In an environment where memory widths may not be an integer multiple of a byte (such as the SPAM VLIW-1 and SPAM VLIW-2 architectures), byte addressing does not make sense. Additionally, byte addressing can be emulated within the RTL definitions of operations and non-terminals. However, in the interest of making it easier to support existing architectures, it may be beneficial to provide support in ISDL to declare certain storage elements as byte-addressed. The natural place for such a definition is in the Storage section of an ISDL description and can be accomplished by the insertion of a single key-word.

Co-Processor Interfaces

ISDL Version 1 does not support fully generic co-processor models. This is because co-processors whose instruction sets are unknown are not used in embedded applications. If the instruction set is already known then it can be included in the instruction set of the main processor as if it was part of the processor. However, fully generic co-processor interfaces are commonly used in general purpose processors and it would be beneficial to support them. Such co-processors would appear as partial operation definitions (i.e., operation definitions with some of the information missing). The natural place for inclusion of such co-processors would be in the Instruction Set section of an ISDL description. Note, however, that co-processors often have interrupts

and exceptions associated with them and these should be included as well in the appropriate section.

External Interface Descriptions

In ISDL Version 1 there is no way of representing peripherals and other interfaces to the external world. This is because such interfaces are not part of the instruction set. However, this makes it impossible to describe certain operations in the instruction set whose sole purpose is to control these peripherals (such as the `RESET` operation in the Motorola 56000 DSP). Additionally, it makes it hard to integrate peripherals into the simulators generated from ISDL since the code that emulates the peripherals has to constantly poll the state of the control registers in the processor. The ability to describe interactions of the instruction set with the peripherals, while being of no use to the compiler, would make it easy to generate simulators that can be easily integrated into a larger simulation environment that simulates the peripherals as well.

In order to allow such interactions, it is necessary is to provide a mechanism for performing function calls to code that is not part of the simulator. In ISDL this would consist of a keyword that denotes such a function call and the definition of a function call stub (to pass parameters to the function being called to simulate the peripherals). This keyword and function stub definitions could be inserted at the appropriate locations in the RTL actions and side-effects of operation definitions (and possibly event definitions for exceptions and interrupts). Thus, real function calls can be generated along with the simulator code that will perform event-driven simulation of the peripherals.

Variable Length Instructions

ISDL Version 1 assumes that the instruction word is of a fixed width[1]. The overwhelming majority of VLIW architectures and *all* unifunctional architectures do indeed have an instruction of fixed width. However, in the interest of reducing code size, some VLIW architectures allow the instruction to be of varying width, by omitting the operations for functional units that would be idle in a specific instruction. There is no way to support such architectures in ISDL Version 1. The least intrusive way of supporting such architectures in ISDL would be to provide an instruction re-write mechanism which would fetch instructions as a stream of the smallest units possible and re-write them in the form of complete VLIW instructions that include `nop` operations for units that are idle. While the existence and nature of such a mechanism is not really optional information, placing it in the Optional Information section of ISDL would be the least intrusive way of including this information. Alternatively, it can be included in the Global Definitions section at the cost of a substantial change in the semantics of this section.

[1] We do not consider the case of additional instruction words being used to store constants as a variable instruction width mechanism.

Formal Arguments for Operations

In ISDL Version 1, parameters to operations are listed as non-terminals or tokens. However, these non-terminals or tokens have to have unique names even though they may have exactly the same semantics. For example, in the SPAM VLIW-1 architecture we had to define three separate non-terminals with exactly the same semantics for `U1_RA`, `U1_RB`, and `U1_RC` in order to to have unique names. A much better method would be to allow operation definitions and non-terminal options to take formal parameters as arguments and attach a type to each of these that would be the single definition of the underlying non-terminal or token. This would remove unnecessary definitions of tokens and non-terminals, reducing the size of descriptions and making them easier to read. However, this would require a substantial change of syntax in the way operations are defined making it hard to maintain backward compatibility.

Separate Timing Information for Side-Effects

ISDL Version 1 dictates that all the actions and side-effects of an operation obey the same timing policy. While almost all architectures we know of conform to this restriction, one can easily imagine architectures in which the side-effects do not share the same timing as the actions (or even the same timing as each other in the case of multiple side-effects). ISDL can be extended to provide additional facilities to describe such cases by allowing the inclusion of timing information in the side-effects RTL itself. This, however, would complicate the ISDL RTL syntax.

Implementation Directives

Currently there is no way to include hints to the hardware model generator in ISDL descriptions. We instead have to rely on algorithms to figure out the intent of the designer or the best way to implement a particular instruction set. In the interest of generating as efficient hardware models as possible, it would be desirable to include implementation directives directly in ISDL. For example, the instruction set architect may know that the best way to implement the instruction set might be to provide a four-port register file and take an extra cycle to load the first parameter, as opposed to building a five-port register file. The exact nature that such directives would take is unclear right now, as is their possible location within as ISDL description. Since any part of an ISDL description other than the the Instruction Word Format section may affect the implementation[2], these directives may also have to be allowed anywhere in the ISDL description.

5.2.2 Automated Architecture Synthesis

While ISDL and the associated tools provide the infrastructure necessary to perform architecture exploration effectively, creating and modifying the architectures still re-

[2]The Instruction Word Format section clearly also affects the implementation - however, this implementation is fixed and requires no further directives.

quires manual effort in the current implementation of the ARIES system. An automated methodology of creating architectures would fully automate the inner loop thus resulting in a tool that requires no manual intervention except for the purposes of tuning the architecture further. The following tasks need to be accomplished in order to fully automate the architecture exploration loop:

1. Generating a good initial architecture.
2. Deriving useful statistics from the simulator, which can be easily extrapolated to locate bottlenecks in the architecture.
3. Discovering ways to automatically remove these bottlenecks by performing suitable architecture modifications.

Note that the last two tasks are very closely coupled and therefore should be considered as parts of a unified approach. All three tasks are explained in further detail below.

The Initial Architecture

Architecture exploration by iterative improvement is equivalent to a gradient descent algorithm in the design space. Gradient descent algorithms are very sensitive to the starting point of the search, therefore a good initial architecture is essential if good solutions are to be obtained.

In order to obtain a good initial architecture, the application code and corresponding profiling information can be analyzed to obtain an estimate of the types of simple operations performed (such as `add`, `mul` etc.) as well as static and dynamic counts of these operations. The next step is to obtain estimates of the silicon area and power consumption of each of hardware implementations of these simple operations. An estimate of the required number of instantiations of each type of operation can be obtained by dividing the required throughput by the execution time of each operation implementation multiplied by the dynamic count of the operation. An estimate of the silicon area and power consumption of the initial architecture can be obtained by multiplying the number of instantiations of each operation by the silicon or power cost of each operation implementation. These estimates can be used to rapidly explore the architecture design space for a promising architecture.

In particular, one way of generating a good initial architecture is to divide the source code into a number of abstract operations (such as `add`, `mul`, `load`, `move`, etc.). Let us call these operations O_1 to O_n. Each of these operations can be implemented in a number of different ways (including a software implementation). Let us denote all the possible implementations of operation O_i by O_{ij} where j varies from 1 to m. An architecture can be considered as a group of operation implementation *instances*, and some interconnect. The following functions can then be defined on operations and their implementations:

- $N(O_{ij})$ denotes the number of instances of operation implementation O_{ij} in a particular architecture. This function effectively describes the architecture.

- $C(O_{ij})$ denotes an estimate of the hardware costs (for example, silicon area) of operation implementation O_{ij}.
- $S(O_{ij})$ denotes an estimate of the encoding (roughly proportional to code size) of operation implementation O_{ij}. Note that this depends on the architecture as well, not only on the operation implementation.
- $T(O_{ij})$ denotes an estimate of the performance cost (execution time) of operation implementation O_{ij}.
- $M(O_{ij})$ denotes the static frequency of operation implementation O_{ij} in the input code.
- $F(O_{ij})$ denotes the dynamic frequency of operation implementation O_{ij} in the input code.

We can now form a set of relations that result in estimates of the performance and hardware cost of the architecture:

$$CodeSize = a \times \sum M(O_{ij}) \times S(O_{ij})$$

$$ProcessorSize = b \times \sum N(O_{ij}) \times C(O_{ij})$$

$$ExecutionTime = c \times \sum \frac{F(O_{ij}) \times T(O_{ij})}{N(O_{ij})}$$

were a, b, and c are suitable heuristically-determined constants. The functions that can be modified are N (the architecture), and M and F (the code implementation functions). We can quickly explore a number of different options and derive estimates for both system size and performance. We can then select the architecture (the N function) that results in the lowest costs and still promises to meet the performance criteria as an initial architecture, and complete the design by creating an ISDL description for it.

Architecture Evaluation

Once a simulator has been generated for a candidate architecture, and the application has been compiled and run on the simulator, we have much more information about how the architecture interacts with the application. However, it is non-trivial to extract information from the simulation trace in a form that can suggest possible improvements.

The information that is readily available from the execution trace (and the compiled program) is the static operation count and the dynamic operation count. From these statistics we can derive the following information by using a set of heuristics:

- Detect under-utilized functional units. These are functional units that fetch `NOP`s more often than the average.

- Detect under-utilized operations. These are operations that have low static or dynamic counts.

- Detect functional unit bottlenecks. These are functional units that have NOP counts substantially below average.

- Detect operation bottlenecks. If an operation appears substantially more often than the rest of the operations in the field, there is probably insufficient resources to execute that operation. If this situation holds for most or all of the functional units that provide this operation type, then it is almost certain that the resources devoted to that operation should be expanded.

- Detect data transfer bottlenecks. This would be reflected by NOPs on functional units while data transfers are taking place.

- Detect operations that could benefit from faster implementations. Operations that appear very often in the dynamic operation count should probably be implemented using faster methods than the rest of the operations, assuming they affect the critical path.

Note that all of the above are heuristics - there is no guarantee that a functional unit with very few NOPs is a true bottleneck, for example. These and other heuristics, however, can be used to suggest possible improvements to the architecture.

Performing Architecture Modifications

Improving an architecture given statistics from the simulator follows two simple rules:

- If a component can be removed while still maintaining performance, then it should be removed.

- If the performance criterion is not being met, the bottleneck should be located and the resources allocated to it expanded.

The interaction of these two rules tends to drive the system to the lowest cost solution that will still meet performance. Note that code size is one of the components that the rules have to contend with. Also note that most of the time there will be multiple options for each rule (i.e., multiple modifications that can be made to either improve performance, or reduce cost).

The first rule is mainly concerned with utilization of components. Components that seem to be underutilized should be removed. Selecting the right option is a complex function of the savings in terms of costs, as well as the performance penalty of the modification. Effectively, each option is assigned a costs saving and a performance penalty and a weighted average of the two is used to rate the options. The option that appears most suitable is then selected and the corresponding modification implemented.

The second rule is mainly concerned with bottlenecks. For each candidate architecture, there will probably be one main bottleneck (and possibly some secondary ones). There may still be multiple options because there may be multiple implementations of the task that is creating the bottleneck. Once again, the options are assigned cost penalties and performance savings, and a weighted average is used to select the best and implement the corresponding modifications.

The main problem is determining the heuristics that will be used to determine the possible options for each rule, and assign the weights to each option.

A few examples of such heuristics are given below:

- If an operation appears often in the static operation count but not proportionally often in the dynamic operation count, then the operation should be considered for heavy encoding that would reduce instruction size at the cost of performance. The savings in cost is equal to the savings in the instruction word size multiplied by the static operation count, minus the expansion in the decode logic. The performance penalty is the additional execution time of the operation, multiplied by the dynamic operation count.

- If a functional unit is underutilized it should be considered for removal. The savings in costs is the area consumed by the implementation of the functional unit, plus the number of instruction word bits required by the functional unit multiplied by the number of instruction words in the program. The performance penalty is the number of operations that have to be mapped to other functional units multiplied by the execution time of each operation on those functional units, multiplied by a heuristically-determined constant that accounts for the fact that some of the operations may fit in `NOP` slots.

- If a functional unit is a bottleneck then the unit should be cloned. The cost penalty is the cost of the unit plus the number of instruction bits for the unit multiplied by the number of instructions in the program (multiplied by a heuristically-determined constant that accounts for the fact that each instruction is now bigger and holds more operations). The performance savings will depend on the secondary bottlenecks.

- If a set of operations tend to appear as a group (for example, an `add` and a `mpy`), then it may be a good idea to combine them together into a single operation (a `mac` operation). This will both reduce code-size and improve performance.

A different but related problem is the accuracy of the measurements. The measurements obtained from the simulator are exact, however, the depend on the compiler having produced code which truly reflects the capabilities of each architecture. Therefore, the real performance figures may differ from those measured because the compiler has been unable to use the architectural features provided in an effective manner. Let us define the *real* performance of an architecture on a given application to be the performance given an optimal implementation of the application on the architecture, and denote it by P^R. Let us defined the measured performance to be

the performance measured for an implementation created by a compiler and denote it by P^M. Let P_A^R and P_B^R denote the *real* performance for architectures A and B respectively, and similarly for P_A^M and P_B^M. Since the proposed method of architecture exploration is gradient decent, it can be shown that the algorithm will work as well as possible if:

$$P_A^M > P_B^M \Rightarrow P_A^R > P_B^R$$

for all architectures A and B and all applications. We call this the *monotonicity criterion*. If the monotonicity criterion holds then no architecture will be considered as better than another architecture by the gradient decent algorithm, unless it really is. This implies that the final architecture selected will be the same despite the fact that the measurements are in fact approximations because of the sub-optimal implementations generated by the compiler. We currently know of no way of *ensuring* that the monotonicity criterion holds. However, the probability that it does hold can be increased if:

- The implementations produced by the compiler are as close to optimal as possible.
- The standard deviation of the probability distribution that the compiler will generate an implementation of a particular performance is as small as possible (i.e., the results of the compiler are as consistent as possible).

Note that the monotonicity criterion and the above criteria also hold for the hardware model generator, which also produces approximate results.

Appendix A

Glossary Of Terms

- **ASIC**: Application Specific Integrated Circuit. An IC custom-designed for one specific application. Usually most custom circuitry in embedded systems is in the form of ASICs.
- **ASIP**: Application Specific Instruction-Set Processor. A processor with an instruction set which is custom-designed for a specific application. Since embedded systems typically have no use for general purpose facilities, they make use of processors with custom instruction sets to reduce the cost of the system and improve performance.
- **Additional Instruction Word**: In some architectures, the instruction word is not long enough to accommodate large constants (such as the destination addresses for jumps and branches) as well as the usual op-codes. In this case, the constant is instead placed in the next instruction word and this word is loaded when needed, but not interpreted as an instruction. This second word is called an additional instruction word.
- **Addressed Storage**: A storage unit that behaves as a group of storage elements, generally of the same width. Individual elements (called locations) may be referred to with an address. Typically, this address is an integer acting as an index into the group of elements.
- **Addressing Modes**: There are various ways for an operation to access its input and output parameters - these are called addressing modes. Most RISC processors provide only a limited number of addressing modes (such as load/store architectures which only offer a register addressing mode). Most DSP and CISC type processors have a wide variety of addressing modes.
- **Ambiguous Instruction Set**: An instruction set with an operation which is multiply-defined and there are at least two of these definitions which will both accept a given instantiation of the operation. For example `Operation 4,6` could correspond to either of the operation definitions below:

```
Non-Terminal ADR:  INT ...  |
```

```
                    NAME ... ;
Field Bad:
    operation INT, ADR ....

    operation ADR, ADR ....
```

- **Architecture Exploration**: A methodology for performing architecture design based on iterative improvement. An initial design is created and evaluated for a particular application. Based on measurements made during the evaluation phase, improvements are made to the architecture, either to improve performance or to reduce cost without sacrificing performance. The process is repeated until no further improvements can be made.

- **Behavioral Language**: A machine description language that uses the behavior of operations in the instruction set to describe the hardware. Behavioral languages generally avoid structural information (such as the structure of the pipeline). They are high level languages so they are generally easier for automatic tools to process, and easier for human engineers to work with.

- **Binary Image**: During the assembly phase, each operation in the instruction set will impart a unique combination of values to a subset of the bits of the instruction word. The set of values of the bits and the subset of the bits, together form the binary image of the operation. This subset of bits can be used to uniquely recognize the particular instance of the operation as being in the corresponding VLIW instruction.

- **Binary Image Overlap**: The binary image of an operation consists of a subset of the bits of the instruction word and a set of unique values for these bits. It is possible for two operations to have binary images where the subsets of bits overlap. This is called binary image overlap and sometimes makes it harder to decode the instruction word.

- **Bitfield Assignment**: The action of setting the appropriate value to the bits in the binary image of an operation.

- **Bitfield Conflict**: In a VLIW instruction it is possible for two operations to attempt to set the same bits in the instruction. This implies that there is a binary image overlap between the two operations. Obviously since the binary image of an operation is the minimum set of bits that must be set to specific values in order to identify an operation, it is not possible for two operations with a binary image overlap to co-exist in the same VLIW instruction. This is called a bitfield conflict. Put simply, it means that two operations in the same instruction are trying to set the same bits to contradictory values and this cannot be allowed.

- **Bitfield**: A contiguous subset of bits in the instruction word.

- **Boolean Clauses**: A set of clauses formulated in Boolean algebra that yield a *true* or *false* answer given a set of *true* or *false* inputs. ISDL uses Boolean clauses as constraints by using operation matches as the inputs.

- **Branch Prediction**: A method of avoiding stalls and delay slots by attempting to predict the outcome of a branch early into the pipeline. The moment a branch instruction is fetched, a piece of logic attempts to guess the outcome of the branch, and hence where the next instruction is going to come from, before it is fetched. Therefore, if the guess is right, no flushing of the pipeline is necessary and no delay slot instructions have to be declared. Pipeline flushes or stalls decrease performance in heavily pipelined machines and delay slots complicate code generation and expose implementation details to the Instruction Set. There are two main methodologies for obtaining the guess: hint-driven in which the branch instruction itself provides the guess and is therefore under the control of the programmer, and automatic in which cached previous outcomes for the particular instruction are used and are therefore probabilistic.

- **CISC**: Complex Instruction Set Computer. Architectures which contain complex and heavily encoded instructions in their instruction sets. Typically they have a variety of complicated addressing modes and instructions that take multiple cycles to complete.

- **Cache Access Pattern**: The sequence of addresses accessed during execution, especially in the context of how this sequence affects any caches present in the system.

- **Code Generator**: A software tool that takes as input a representation of a piece of source code and emits assembly or binary code specific to an architecture to implement that particular piece of source code on that particular architecture. Usually the input is in a compiler intermediate form.

- **Constraint**: A boolean clause with operation matches as inputs, which operate on a current instruction or stream of instructions. If the boolean clause returns a value of *false* then the instruction is in violation of the constraint. This could either mean that the instruction is malformed or improperly scheduled. In either case the hardware cannot guarantee that the instruction stream will execute as expected. If the boolean clause returns a value of *true* then the instruction is not in violation of this particular constraint. It does not imply, however, that it is well-formed and properly scheduled since it might still violate other constraints. Only an instruction that does not violate **any** constraints is well-formed and properly scheduled.

- **Constraints Section**: One of the sections in an ISDL description specifically dedicated to exposing restrictions in the form of operations or instructions, or the scheduling of instructions. These are generally restrictions imposed by the hardware.

- **Control Flow**: Operations that deviate from the sequential fetching of instructions, such as branches, jumps, subroutine calls, returns etc.

- **Control Register**: A register which when written to may have side effects (such as resetting interrupt modes or writing values to an output port), and when read from is not guaranteed to return the last value written to it. Such registers are usually used to control various modes of the processor and peripherals, and to inspect the state of the processor and peripherals. These registers have to be identified so that code generators do not attempt to use them as temporary storage.

- **Costs**: Any instruction has a set of costs associated with it, such as the number of cycles it would take to execute the instruction on the hardware, or the number of additional words it may require. The code generator needs to be aware of such costs in order to be able to make trade-off calculations between different implementations of the same piece of code.

- **Cycle-Accurate Simulation**: A form of simulation that assigns execution times (in cycles) to each instruction in the instruction stream, and where these execution times correspond to the exact number of cycles the same instruction stream would take on the real hardware.

- **Cycle Cost**: The number of cycles it would take for a particular instruction to execute on the hardware.

- **DSP**: Digital Signal Processor. A processor specifically designed and optimized to operate on digital data streams. Such processors usually have complex architectures capable of high numerical throughput and rely on hardware acceleration for a lot of functions (such as multiply-accumulate operations, address generation, butterfly operations for FFT etc.). Embedded systems in data-dominated applications (applications where most of the emphasis is placed on data manipulation rather than control flow operations) typically rely on DSPs for their processing power.

- **Data Register**: A register included in the data path for the explicit purpose of performing data manipulation. Such registers do not have side effects when written to, and they return the last value written to them when read from. They act as simple storage elements.

- **Delay Slots**: In pipelined architectures the outcome of a branch instruction may not be available until a few pipeline stages *after* instruction fetch. In this case, instructions following the branch may have already been loaded into the pipeline even though the branch eventually determines control should have been transferred to a different location in the program. Some architectures specify that such instructions will be executed even if control is transferred to a different location in the program. The number of instructions following a branch that will complete execution when control is transferred to the target of

the branch, are called delay slots. The code generation should take delay slots into account when emitting branch instructions and either emit the branch instruction correspondingly early in the instruction stream (if possible), or fill them with NOOPs otherwise. Since delay slots complicate code generation and expose implementation details to the instruction set, they are often avoided by flushing or stalling the pipeline, or by the use of branch prediction.

- **Design Criteria**: A set of constraints on the implementation of an architecture such as maximum silicon area, maximum power consumption, minimum performance, etc.

- **Embedded System**: A computer system dedicated to a particular application (such as an engine management computer in a car, or a digital filter in a sound processor). Most computers that are not used for general purpose computation are embedded systems.

- **Exception**: A condition that arises when an operation cannot proceed normally, either because of the current state of the processor or because of the values it received as input. Examples of exceptions are division by zero (which cannot proceed because of the input values), or a privileged mode instruction being issued in normal user mode (which cannot proceed because of the current mode of the processor). Typically exceptions will cause a control flow operation to a fixed address where a small program called an exception handler will try to recover from the failure.

- **Exception Handler**: A small piece of code that is executed when a specific exception condition occurs, and which attempts to recover from the exception condition.

- **Field Definition**: A list of operation definitions that are mutually exclusive, grouped together in the Instruction Set Section of an ISDL description. Typically corresponds to the operations that can be performed on a single functional unit in a VLIW processor.

- **Fixed Length Instructions**: The instruction word of a processor is called a fixed length instruction if all instructions available in the instruction set of the processor have the same word length. This does *not* include additional words that may be needed to provide large constants.

- **Flag**: Usually a single-bit piece of state (often provided as a bit in a control register) that denotes a certain condition (e.g. overflow) has occurred, or that denotes a current mode (e.g. interrupts disabled). Flags can be, and sometimes are, longer than a single bit.

- **Format Section**: The section of an ISDL description that describes the structure of the instruction word - i.e. the division into logical units called bitfields.

- **Global Definitions Section**: The section of an ISDL description that defines abstractions (such as tokens, non-terminals, and split functions) that are used in later sections of the description.

- **Hardware/Software Co-design**: A design methodology targeted mainly towards embedded systems that have both a hardware and a software component and attempt to integrate both components on a single chip. Such systems have the property that decisions concerning the hardware component drastically affect the software component and vice-versa. Because of this, the most effective way to design such systems is to provide a common framework for designing and evaluating both components together.

- **Hint Driven Branch Prediction**: A method of branch prediction where the guess as to the outcome of the branch is provided explicitly by the branch instruction used. Thus, the branch prediction mechanism is completely under the programmer's control. It can be achieved by providing duplicate branch instructions with exactly the same action except for the fact that one causes the guess to be that control will be diverted to the new location and the other causes the guess to be that control will remain sequential.

- **ISDL**: Instruction Set Description Language. A behavioral machine description language specifically designed to support a wide variety of architectures (including VLIW) and support the automatic generation of design environment tools. It closely models the structure of the Programmer's Manual.

- **Instruction**: The smallest logical unit that can be fetched by a processor. For VLIW architectures this might correspond to a number of operations, all grouped together into a single unit. For Super-Scalar architectures this may be a small part of what can be decoded and dispatched by the hardware.

- **Instruction Field**: A set of mutually exclusive but related operations in the instruction set of a target architecture. These typically (but not always) correspond to all the operations that a single functional unit in a VLIW architecture can perform.

- **Instruction Level Parallelism**: The ability to execute more than a single operation at any given time when this ability is visible in the instruction set of the target architecture. This, for example, would not encompass the parallelism in a super-scalar architecture, where the instruction set looks the same as that of a unifunctional architecture but operations *are* actually performed in parallel on multiple functional units.

- **Instruction Level Simulator**: A simulator for a target architecture that simulates effects visible in the instruction set but no lower than that. For example, the simulator may give cycle-accurate simulation for the target architecture but not have an explicit model of the pipeline and thus make it impossible to inspect the state of the pipeline at any given time (the pipeline is not generally visible in the instruction set).

- **Instruction Set Section**: A section of an ISDL description that contains the definitions for all the operations available in the instruction set of the target architecture.

- **Interrupt**: A hardware signal that triggers a control flow operation in the processor. Control is diverted away from normal program execution and to a fixed address which may or may not depend on the signal that triggered the control flow operation. A small program called an interrupt handler resides at this address. This program attempts to deal with whatever condition asserted the hardware signal in the first place. Typically interrupts are used by peripherals to signal to the processor that they require attention. Interrupts may also cause mode changes in the processor (such as a switch from normal user mode to privileged mode).

- **Interrupt Handler**: A small piece of code that is executed when a specific interrupt event occurs and which attempts to service the interrupt.

- **Latency**: The number of instructions that need to be fetched before the effects of the current operation become visible.

- **Lex**: A lexical analyzer generator (a tool that receives as input a file containing regular expressions describing the lexical entities of a language and generates as output code that will implement a lexical analyzer)[25]. The lexical analyzer that is used to process ISDL descriptions was generated by Lex. The generated assemblers also use lexical analyzers generated by Lex to process their input files.

- **Lexical Analyzer**: A program that divides an input stream of characters into lexical entities. Together with a parser it can be used to parse input files written in various languages. ISDL descriptions are parsed using a lexical analyzer generated by Lex. The generated assemblers also use lexical analyzers generated by Lex to process their input files.

- **Load/Store Architectures**: A class of architectures in which all operations except loads and stores take their inputs from and write their results to registers.

- **Loop Counter**: A special purpose register in the data path used to keep count of how many iterations a zero overhead loop has performed (or has remaining).

- **Loop Destination Address**: The address of the first instruction in a zero overhead loop (where the loop will branch back to for each iteration until it terminates).

- **Loop Termination Address**: The address of the last instruction in a zero overhead loop (after which the test is performed to see if the loop terminated or if another iteration should be executed).

- **Macro Definitions**: Definitions of text to be expanded into longer and more complicated pieces of text by a preprocessor, before the description is actually processed by the tools. It allows common text patterns to be given shorter names in order to shrink the size of a description.

- **Memory Management Hardware**: Hardware that handles address translation schemes (such as the ones needed for virtual memory).

- **Memory-Mapped I/O**: A group of control registers that appears as memory to the Instruction Set but has the function of communicating with I/O peripherals. Typically, a processor can read from and write to these locations in the same way that it would read from or write to normal memory (sometimes using exactly the same operations). However, writes to memory-mapped I/O locations change the state of I/O peripherals and ports, and reads inspect the values and state on I/O peripherals and ports.

- **Micro-controller**: A processor designed mainly to control other peripherals. Such processors typically have small instruction words and narrow data paths (usually 8-bit instructions and 8-bit data paths) but have a characteristically disproportionate amount of I/O capability (such as three 8-bit bi-directional ports, a serial port and an external memory bus on an 8-bit micro-controller). They appear mostly in control-dominated applications.

- **Multiple Operation Definitions**: Operation definitions within the same field that share a common operation name but different numbers and/or types of arguments.

- **Non-Terminal**: An abstraction that groups syntactically unrelated entities into a logical group. Conceptually, a non-terminal consists of a number of alternatives, any of which may be replace the non-terminal in an actual instance of an operation. These alternatives are called options.

- **Non-Terminal Costs Clause**: Different options in a single non-terminal may result in different costs in an operation that uses this non-terminal. Non-terminals contain a set of cost expressions that allow operations to account for this by providing each option with its own set of cost expressions. This set of expressions is called a costs clause.

- **Non-Terminal Option**: One of the many possible alternatives that a non-terminal groups together into an abstraction. Each option consists of a syntax definition, a return value, an RTL action, an RTL side effect, a costs clause, and a timing clause.

- **Non-Terminal RTL Action**: This is the part of a non-terminal option definition that describes the action of the particular option (usually simply a storage reference). Each option in the non-terminal definition has its own RTL action clause.

- **Non-Terminal RTL Side Effect**: This is the part of a non-terminal option definition that describes the side effects of the particular option (usually simply a storage reference). Each option in the non-terminal definition has its own side effects clause.

- **Non-Terminal Return Value**: Since a non-terminal is a conceptual abstraction of a number of different options, there must be a mechanism of differentiating between options once a non-terminal is instantiated. This is done by providing a return value which is different for each option in the non-terminal. This return value can then be used to perform the bitfield assignment for the non-terminal so it is very common to encode the return values in such a way that they are identical to the binary image of the non-terminal.

- **Non-Terminal Timing Clause**: Different options in a single non-terminal may result in different timing in an operation that uses this non-terminal. Non-terminals contain a set of timing expressions that allow operations to account for this by providing each option with its own set of timing expressions. This set of expressions is called a timing clause.

- **Op-code**: Operation Code. A bitfield whose value typically uniquely identifies the *type* of operation to be performed by a processor or functional unit.

- **Operation**: The smallest unit of data manipulation that can be independently performed by the hardware. In VLIW architectures this is by definition a subpart of the instruction since each VLIW instruction can be considered to be performing a number of independent data manipulations on each of its functional units. In other words, a VLIW instruction consists of multiple operations. In unifunctional architectures, the operation *is* by definition the instruction.

- **Operation Assembly Definition**: The part of an operation definition that provides the assembly syntax for the operation. It is also used to give a name to the operation. It is written in terms of an operation name and a list of parameters which may be tokens or non-terminals.

- **Operation Bitfield Assignment**: The part of an operation definition that contains the description of how to perform all the bitfield assignments for the binary image of the operation. It is written in a restricted expression form based on assignments.

- **Operation Costs**: The part of an operation definition that declares the costs associated with an operation. All costs in ISDL are numerical.

- **Operation Definition**: A set of ISDL clauses which fully defines a single operation within an instruction field.

- **Operation Match**: A regular expression that returns *true* if an operation (or one of a group of operations) is present in an instruction.

- **Operation RTL Action**: A description of the desired effects of the operation on processor state, written in an RTL-type language. If the operation makes use of non-terminals, the RTL action definition of the operation may refer to the RTL action definition of the non-terminals.

- **Operation RTL Side Effect**: A description of the side effects of an operation on visible state, written in an RTL-type language. If the operation makes use of non-terminals, the side effects definition of the operation may refer to the side effects definition of the non-terminals.

- **Operation Timing**: The part of an operation definition that declares the timing parameters associated with an operation. All timing parameters in ISDL are numerical. Timing parameters are mainly concerned with when the effects of an operation become visible in the state of the machine.

- **Optional Architectural Information Section**: A section of an ISDL description that provides information on the target architecture that might not be necessary for useful design tools to be generated, but may result in better tools. Information on the cache system is a good example - this information is not necessary to generate a code-generator or a simulator but both might benefit from the extra information.

- **Orthogonal Operations**: Operations that can be performed in parallel and do not affect each other in any way. In particular, the presence of one operation cannot preclude the presence of the other and the form of one operation cannot restrict the form of the other. In VLIW architectures, all operations would be orthogonal if there were no hardware restrictions. In practice this is rarely the case.

- **Partitioning (Hardware/Software)**: The process of dividing a task into two sub-parts and assigning one part to a custom hardware (i.e. ASIC) implementation and the remainder to a software implementation. The main goal of the process is to select the partition between the two parts so that cost and performance are optimized.

- **Program Counter**: A special purpose register present in every architecture, which points to the current (or next in some cases) instruction in the instruction stream. Typically the contents of this register cannot be explicitly transferred to another location (i.e. read). Furthermore, changing the value of this register diverts control to a new point in the instruction stream and the value automatically gets updated on every instruction fetch. In this respect, it behaves like a control register which can only be written using control flow operations and cannot be read explicitly.

- **RISC**: Reduced Instruction Set Computer. A processor with a simple architecture and a small instruction set. Typically such processors have very few addressing modes (most such processors are load/store architectures). They

often have simple pipeline implementations that are capable of executing each instruction in one clock cycle and have very fast clock speeds. This gives them a high throughput but results in larger code size for the same input source code (note that this may make instruction caches less efficient and thus reduce effective throughput).

- **RTL Functions**: A provision in the version of RTL that ISDL supports, to call functions to perform specialized data manipulation (such as floating point operations etc.). ISDL pre-defines a set of RTL functions; more may be defined and used if the tools that process ISDL are updated to make use of them.

- **Range Operator**: An operator that denotes a range of numbers or characters by listing the beginning value and the ending value of the range.

- **Register File**: An addressed storage unit, typically equipped with multiple ports, through which a functional unit may access multiple registers. The registers are typically of the same width. Register files are typically used to provide the source and destination operands to operations in load/store architectures. They are, more often than not, closely coupled to one or more functional units and can therefore be accessed in a single clock cycle.

- **Register Transfer Language**: RTL. A type of language well suited to describing the behavior of data paths. The main type of statement is an assignment which assigns a new value to a piece of state. The language also contains a number of control features that allow it to specify the behavior of control circuitry as well. ISDL, however, has its own concept of control so it only makes use of a restricted subset of the language to describe the behavior of the data paths on a per-operation basis.

- **Regular Expressions**: An expression syntax specifically designed to describe regular grammars. It makes heavy use of wild-card characters. ISDL regular expressions are augmented with variable matches which require the usage of a stack.

- **Regular Grammar**: A grammar is a set of input strings. A regular grammar is one that can be identified by an FSM.

- **Resource Conflict**: An attempt by two different operations to use the same data path resource (such as a register, data bus, or port on a register file) for different purposes. If an architecture's instruction encoding exposes resource conflicts to the instruction set, then a number of restrictions must be placed on the operations to avoid such conflicts. Operations that result in a resource conflict can never be executed successfully by the hardware.

- **Retargetable Compiler**: A compiler which is capable of emitting output code for new architectures when provided with a description of these architectures.

- **Retargetable Simulator**: A simulator which has the ability to obtain a description of a new architecture and assume the task of simulating that particular architecture.

- **Saturation Arithmetic**: A method of handling overflow (underflow) in arithmetic operations. If the result overflows (underflows), the appropriate flag is set and the largest (smallest) number that can be represented by the architecture is produced as the result.

- **Split Function**: A function that can take a large binary constant and split it into a number of bitfields. These functions are automatically generated from split function definitions in the Global Definitions Section.

- **Stack**: A type of storage unit that behaves like a stack of plates: i.e. multiple values can be stored and the last value stored will be the first one read. It is an addressed storage unit which uses a special purpose register (the stack pointer) to provide the address, and two special purpose operations to read and write the contents. *Pop* reads the contents of the location pointed to by the stack pointer and decrements the stack pointer automatically. *Push* increments the stack pointer first and then writes a value into the location pointed to by the new value of the stack pointer. It is an error to pop a value from an empty stack or to push a value onto a full stack. Typically only the value pointed to by the stack pointer (called the top of the stack) can be accessed even though a stack is a form of addressed storage.

- **Stack Pointer**: The special purpose register that provides the address for a stack.

- **Stall Cost**: The maximum additional number of cycles taken up by stalls, when the next instruction tries to use the results of the current operation. Note that this cost is derated linearly as the distance between the current operation and the instruction which attempts to use the results increases. In other words, if the stall cost is 3, and the instruction using the results of the current operation is the next instruction (distance 1), the actual number of stall cycles is 3. If the distance increases to two (i.e. the instruction using the results is the instruction after the next one) the actual number of stall cycles is 2, and so on until the stall cost falls to 0. Stall cost is one of the ISDL predefined costs.

- **State**: The collective amount of visible storage in a processor. This is equivalent to the storage of the processor, with the distinction that storage usually refers to the actual storage elements available while state usually refers to the values contained in these storage elements. A processor can be modeled in terms of its state and the operations that modify this state.

- **Storage**: The collective amount of storage elements present in a processor and visible to the programmer. This is equivalent to the state of the processor, with the distinction that storage usually refers to the actual storage elements

available while state usually refers to the values contained in these storage elements.

- **Storage Alias**: An alternative name for a subset of the state of a processor. It usually groups together subsets of the storage already defined and treats them as new storage units, with the exception that changes in the subset affect the value read from the newly defined state (the aliased state) and vice versa.

- **Storage Depth**: Addressed storage can be considered as a collection of registers (special purpose or otherwise). The number of such registers that make up an addressed storage unit is referred to as the *depth* of the storage unit. The *depth* of a single register (a single storage element) is 1.

- **Storage Reference**: An expression identifying a subset of the storage available in a processor. It consists of a name, possibly an address or range of addresses (using a range operator), and/or a subset of bits within the named storage unit. The name can be the identifier associated either with a properly defined storage unit or the identifier of a storage alias.

- **Storage Section**: The section of an ISDL description that describes the storage units available to the programmer. It consists of a list of storage unit definitions and, possibly, a number of storage aliases.

- **Storage Type**: The type of a storage unit. ISDL currently defines the following types of storage units: memories (including instruction memories), register files, stacks, registers, control registers, a program counter and memory-mapped I/O locations.

- **Storage Width**: The width in bits of a storage element. For a register (control or otherwise) this is the total number of bits in the register. For an addressed storage unit, this refers to the width of each element in the storage unit (which should be the same for all elements in the storage unit).

- **Structural Language**: A language that describes a target architecture by giving the structure of the architecture (i.e. by describing the functional units and storage elements in the architecture and how these are connected together). Structural languages contain much information which is not relevant to code generation or instruction level simulation, but can be used to generate both the tools as well as an implementation of the architecture.

- **Super-Scalar**: A way of allowing parallelism in the implementation of a processor without exposing this parallelism to the instruction set. Thus, super-scalar architectures typically have instruction sets which are reminiscent of uni-functional processors, but have implementations that contain multiple identical functional units and issue instructions to these units in parallel whenever precedence constraints in the software allow it. In effect, super-scalar architectures do some code-scheduling in hardware while VLIW architectures with multiple identical functional units do all their scheduling in software.

- **Time-Shifted Constraint**: A constraint that expresses a restriction between instructions issued at different times.

- **Timing**: A model describing when the effects of a given operation are visible to other operations. Usually expressed in terms of one or more numerical values.

- **Token**: A token is an ISDL abstraction which groups together one or more lexically related entities (such as the names of registers in a register file).

- **Token Assembly**: The part of a token definition that describes the assembly syntax of the lexically related entities grouped under a token.

- **Token Return Value**: Since tokens group together one or more lexical entities, it is necessary to provide a mechanism that identifies which of the lexical entities was present for a particular instantiation of a token. This is provided by the return value which is a numerical value unique to each of the lexical entities grouped under a token. The return value is often used to generate a binary image for the entity represented by the token, so the encodings of the return values should be chosen to make this process easier.

- **Trap**: An exception generated by the use of an operation specifically designed for this purpose. Just as in the case of normal exceptions, control is diverted to a fixed address where a trap handler attempts to service the exception. Traps may be, and often are, accompanied by changes in the mode of the processor (such as a change from user mode to privileged mode). They are often used to trigger execution of kernel functions in privileged mode, or to inform a piece of privileged code (usually the kernel) of a required service.

- **Trap Handler**: A small piece of code that is executed when a specific trap event occurs, and which attempts to service the trap.

- **Truncation Arithmetic**: A method of handling overflow (underflow) in arithmetic operations. If the result overflows (underflows), the appropriate flag is set and the result is set to as many bits as the destination can hold (i.e. the top few bits are truncated).

- **Unifunctional**: An architecture that can only perform one operation at a time. This does not necessarily mean that it only has one functional unit. It *does* mean that only one functional unit is active at any given time (ignoring program counter increment operations). Unifunctional architectures have no instruction level parallelism.

- **Usage(Timing)**: A timing parameter associated with each operation that declares when the "functional unit" executing the operation may be used again. A usage of 2 means that the "functional unit" (i.e. the field containing this operation) may not be used in the next instruction, and therefore a `NOOP` should be selected from the corresponding field in the next instruction. If any other

operation is selected instead, the hardware will stall the next instruction until the functional unit becomes available once again.

- **VLIW**: Very Long Instruction Word. An architecture with instruction level parallelism, capable of performing multiple operations on multiple functional units at the same time. Each functional unit is typically independent of the others and has its own dedicated bitfields in the instruction word. This typically results in very large word lengths, hence the name.

- **Variable Length Instructions**: Some architectures have instructions that may be of different lengths (depending on the action that the instruction is taking). The instruction word of such an architecture is called a Variable Length Instruction.

- **Variable Match**: An operation match containing a variable binding wild-card. The first time the given variable binding occurs it will match zero or more characters of any type and store these in a variable. Any subsequent times it occurs, it will only match the string that was stored in the variable after the first match.

- **Visible State**: The collective state in a processor that can be explicitly inspected and/or modified by the use of a sequence of processor instructions. Thus, all registers that can be directly written to or read from with operations from the instruction set, are part of the visible state. All registers that cannot (such as temporary registers in certain functional units and the registers making up a pipeline) are not considered part of the visible state.

- **Wild-cards**: Special characters in regular expressions that can match more than a single character and may match one or more copies of such characters. For example "?" will match any single character while "*" will match zero or more copies of any single character.

- **Yacc**: Yet Another Compiler Compiler[25]. A parser generator which takes a description of a context free grammar and generates a parser to parse it. The generated parser in combination with a lexical analyzer can be used to process inputs in a given language (hence the name compiler compiler). ISDL is processed using a parser generated by Yacc. The generated assemblers also use parsers generated by Yacc. Yacc generates SR parsers with single token lookahead and this must be taken into account when generating grammars for them.

- **Zero Overhead Looping**: A methodology of creating efficient software loops by performing all control overhead associated with the loop (such as decrementing a counter variable, checking the variable for zero or some other value, and performing flow control) in hardware. Since data dominated applications often make heavy use of small data manipulation loops, most DSP processors provide one or more forms of zero overhead looping.

Appendix B

BNF Syntax of ISDL

```
NAME     := [a-zA-z][a-zA-Z0-9_]+
INT      := -?[0-9]+
HEX      := 0x[0-9A-F]+ | 0x[0-9a-f]+
FLOAT    := -?[0-9]+\.[0-9]+([eE]-?[0-9]+)?
STRING   := "[^"]*" | '[^']*'
CHAR     := '.'
DIGIT    := [0-9]
SPACE    := [ \t\n]+
RGX_RNG  := \[[^\]]*(\\\])?[^\]]*\]
```

```
<number>            := INT | HEX
<constant>          := INT | HEX | FLOAT
<range_operator>    := '[' <number> '-' <number> ']'
<index_operator>    := '[' <number> ']'
```

```
<ISDL_Description>  := <Format> <Definitions> <Storage> <Instruction>
                       <Constraints> <Optional>
```

```
<Format>            := 'Section' 'Format' <Format_Field_List>

<Format_Field_List> := <Format_Field> |
                       <Format_Field> <Format_Field_List>

<Format_Field>      := NAME '=' <Format_Subfield_List> ';'

<Format_Subfield_List>  := <Format_Subfield> |
```

```
                        <Format_Subfield> ',' <Format_Subfield_List>

<Format_Subfield>    := NAME <index_operator>



<Definitions>        := 'Section' 'Global_Definitions'
                        <Definitions_List>

<Definitions_List>   := <Single_Definition> |
                        <Single_Definition> <Definitions_List>

<Single_Definition>  := <Token_Definition>           |
                        <Non_Terminal_Definition>    |
                        <Split_Function_Definition>

<Token_Definition>   := 'Token' <Token_Syntax> NAME <Token_Return> ';'

<Token_Syntax>       := NAME | NAME <Token_Range_Operator> |
                        STRING | STRING <Token_Range_Operator>

<Token_Range_Operator>  := '[' <number> '..' <number> ']'

<Token_Return>       := '{' '}' | '{' <number> ';' '}'
                        '{' <Token_Range_Operator> ';' '}'


<Non_Terminal_Definition>   := 'Non_Terminal' NAME ':'
                               <Non_Terminal_Option_List> ';'

<Non_Terminal_Option_List>  := <Non_Terminal_Option> |
                 <Non_Terminal_Option> '|' <Non_Terminal_Option_List>

<Non_Terminal_Option>   := <NT_Option_Syntax_List> <NT_Option_Return>
                     <NT_Option_Action> <NT_Option_Side_Effect>
                     <NT_Option_Cost_Mod> <NT_Option_Timing_Mod>

<NT_Option_Syntax_List> := <NT_Option_Syntax_Item> |
                       <NT_Option_Syntax_Item> <NT_Option_Syntax_List>

<NT_Option_Syntax_Item> := NAME | CHAR

<NT_Option_Return>  := '{' <NT_Option_Return_Assignment_List> '}'
```

```
<NT_Option_Return_Assign_List>  := <NT_Option_Return_Assign_Item> |
                                   <NT_Option_Return_Assign_Item>
                                   <NT_Option_Return_Assign_List>

<NT_Option_Return_Assign_Item>  := <NTOR_Assign_Left> '='
                                   <Bitfield_Assign_Expression> ';'

<NTOR_Assign_Left>              := <Bitfield_Assign_Left> |
                                   '$$'

<NT_Option_Action>      := '{' <RTL_Partial> '}'

<NT_Option_Side_Effect> := '{' <RTL_Partial> '}'

<NT_Option_Cost_Mod>    := '{' <CT_Clause> '}'

<NT_Option_Timing_Mod>  := '{' <CT_Clause> '}'

<RTL_Partial>       := <RTL_Statement_List> | <RTL_Expression>

<Split_Function_Definition> := 'Split' '.' NAME
                               <Split_Function_Subfield_List> ';'

<Split_Function_Subfield_List>  := <Subfield_Name> |
                                   <Subfield_Name> '+'
                                   <Split_Function_Subfield_List>

<Subfield_Name> := NAME '.' NAME



<Storage>       := 'Section' 'Storage' <Storage_Definition_List>

<Storage_Definition_List>   := <Storage_Definition> |
                  <Storage_Definition> <Storage_Definition_List>

<Storage_Definition>    := <IMEM_Definition>    |
                           <MEM_Definition>     |
                           <RegFile_Definition> |
                           <Reg_Definition>     |
                           <CReg_Definition>    |
                           <MMIO_Definition>    |
                           <PC_Definition>      |
                           <Stack_Definition>   |
```

```
                          <Alias_Definition>

<IMEM_Definition>       := 'Instruction' 'Memory'
                           NAME '=' <Adressed_Size>

<MEM_Definition>        := 'Memory' NAME '=' <Adressed_Size>

<RegFile_Definition>    := 'RegFile' NAME '=' <Adressed_Size>

<Reg_Definition>        := 'Register' NAME '=' <Register_Size>

<CReg_Definition>       := 'CRegister' NAME '=' <Register_Size>

<MMIO_Definition>       := 'MMIO' NAME '=' <Adressed_Size>

<PC_Definition>         := 'ProgramCounter' NAME '=' <Register_Size>

<Stack_Definition>      := 'Stack' NAME '(' NAME ')' '='
                            <Adressed_Size>

<Alias_Definition>      := 'Alias' NAME <Storage_Reference_List> ';'
                           <Storage_Size>

<Adressed_Size>         := <number> ',' <number>

<Register_Size>         := <number>

<Storage_Size>          := <Adressed_Size> | <Register_Size>

<Storage_Reference_List>    := <Storage_Reference> |
                    <Storage_Reference> ',' <Storage_Reference_List>

<Storage_Reference>     := NAME                              |
                           NAME <IR_operator>                |
                           NAME <IR_operator> <IR_operator>

<IR_operator>           := <index_operator> | <range_operator>



<Instruction>   := 'Section' 'Instruction_Set' <Inst_Field_List>

<Inst_Field_List>       := <Inst_Field> |
                           <Inst_Field> <Inst_Field_List>
```

```
<Inst_Field>    := 'Field' NAME ':' <Operation_Definition_List>

<Operation_Definition_List> := <Operation_Definition> |
                  <Operation_Definition> <Operation_Definition_List>

<Operation_Definition>  := <Operation_Syntax>
                           <Operation_Bitfield_Assign>
                           <Operation_Action>
                           <Operation_Side_Effects>
                           <Operation_Costs> <Operation_Timing>

<Operation_Syntax>      := NAME <Operation_Parameter_List> |
                           NAME

<Operation_Parameter_List>  := NAME |
                               NAME ',' <Operation_Parameter_List>

<Operation_Bitfield_Assign> := '{' <Bitfield_Assign_List> '}'

<Bitfield_Assign_List>  := <Bitfield_Assign> |
                           <Bitfield_Assign> <Bitfield_Assign_List> |

<Bitfield_Assign>       := <Bitfield_Assign_Statement> |
                           <Additional_Word_Clause>

<Bitfield_Assign_Statement> := <Bitfield_Assign_Left> '='
                               <Bitfield_Assign_Expression> ';'

<Bitfield_Assign_Left>  := <Subfield_Name> | 'Split' '.' NAME

<Bitfield_Assign_Expression>    := <number>                          |
                                   NAME                              |
                                   'CURRENT'                         |
                                   Subfield_Name                     |
                                   '(' <Bitfield_Assign_Expression> ')' |
                                   '~' <Bitfield_Assign_Expression>  |
                     <Bitfield_Assign_Expression> <BA_Binary_Operator>
                                   <Bitfield_Assign_Expression>

<BA_Binary_Operator>    := '&' | '|' | '^' | '+' | '*' | '>>' | '<<'

<Additional_Word_Clause>    := 'Additional' '(' <number> ','
                               <Bitfield_Assign_Statement> ')' ';'
```

```
<Operation_Action>          := '{' <RTL_Statement_List> '}'

<Operation_Side_Effects>    := '{' <RTL_Statement_List> '}'

<Operation_Costs>           := '{' <CT_Clause> '}'

<Operation_Timing>          := '{' <CT_Clause> '}'

<RTL_Statement_List>        := <RTL_Statement> |
                               <RTL_Statement> <RTL_Statement_List> |

<RTL_Statement>             := <RTL_Declaration>       |
                               <RTL_Assignment>        |
                               <RTL_If_Clause>         |
                               <RTL_For_Clause>        |
                               <RTL_While_Clause>      |
                               <RTL_Switch_Clause>     |
                               NAME ';'                |
                               <RTL_Function_Call> ';'

<RTL_Declaration>       := 'int' <number> NAME ';'

<RTL_Assignment>        := <RTL_Assign_Left> '<-' <RTL_Expression> ';'

<RTL_If_Clause>         := 'if' '(' <RTL_Expression> ')'
                             '{' <RTL_Statement_List> '}' ';' |
                           'if' '(' <RTL_Expression> ')'
                             '{' <RTL_Statement_List> '}'
                             'else' '{' <RTL_Statement_List> '}' ';'

<RTL_For_Clause>        := 'for' '(' <RTL_Statement> <RTL_Expression>
                  <RTL_Statement> ')' '{' <RTL_Statement_List> '}' ';'

<RTL_While_Clause>      := 'while' '(' <RTL_Expression> ')'
                           '{' <RTL_Statement_List> '}' ';'

<RTL_Switch_Clause>     := 'switch' '(' <RTL_Expression> ')' '{'
                  <Switch_Case_List> <Switch_Optional_Default> '}' ';'

<Switch_Case_List>      := <Switch_Case> |
                           <Switch_Case> <Switch_Case_List>

<Switch_Case>           := 'case' <number> ':' '{'
                                <RTL_Statement_List> '}' ';'
```

```
<Switch_Optional_Default>   := 'default' ':' '{'
                                <RTL_Statement_List> '}'

<RTL_Assign_Left>       := <Storage_Reference> | <Token_Name> |
                           <Non_Terminal_Name> | <Tmp_Name> | 'NULL'

<Token_Name>            := NAME

<Non_Terminal_Name>     := NAME

<Tmp_Name>              := NAME

<RTL_Expression>        := <constant>                                  |
                           <Storage_Reference>                         |
                           <Token_Name>                                |
                           <Non_Terminal_Name>                         |
                           <Tmp_Name>                                  |
                           <RTL_System_Flag>                           |
                           <RTL_Unary_Operator> <RTL_Expression>       |
             <RTL_Expression> <RTL_Binary_Operator> <RTL_Expression>   |
                            <RTL_Function_Call>                        |
                           '(' <RTL_Expression> ')'

<RTL_System_Flag>       := NAME | <RTL_Function_Call>

<RTL_Unary_Operator>    := '~' | '!' | '-'

<RTL_Binary_Operator>   := '+' | '-' | '*' | '/' | '%' | '>>' | '<<' |
                           '==' | '<' | '>' | '<=' | '>='

<RTL_Function_Call>     := NAME '(' <RTL_Parameter_List> ')' |
                           NAME '(' ')'

<RTL_Parameter_List>    := <RTL_Parameter> |
                           <RTL_Parameter> ',' <RTL_Parameter_List>

<RTL_Parameter>         := <RTL_Expression> | STRING

<CT_Clause>             := <CT_Assign_List>

<CT_Assign_List>        := <CT_Assign> |
                           <CT_Assign> <CT_Assign_List> |

<CT_Assign>             := NAME '=' <CT_Expression> ';'
```

```
<CT_Expression>          := <constant>                                    |
                            <Token_Name>                                  |
                            <Non_Terminal_Name>                           |
                            <Field_Name>                                  |
                            <Storage_Reference>                           |
                            <CT_Unary_Operator> <CT_Expression>           |
                 <CT_Expression> <CT_Binary_Operator> <CT_Expression>     |
                            <CT_Function_Call>

<CT_Unary_Operator>      := '-'

<CT_Binary_Operator>     := '+' | '-' | '*' | '/' | '%' | '==' | '<' |
                            '>' | '<=' | '>='

<CT_Function_Call>       := NAME '(' <CT_Parameter_List> ')'
                            NAME '(' ')'

<CT_Parameter_List>      := <CT_Expression> |
                            <CT_Expression> ',' <CT_Parameter_List>



<Constraints>    := 'Section' 'Constraints' <Constraint_List>

<Constraint_List>        := <Constraint> |
                            <Constraint> <Constraint_List> |

<Constraint>     := <Constraint_Expression>

<Constraint_Expression> := '(' <Regular_Expression> ')'                  |
              <Constraint_Unary_Operator> <Constraint_Expression>        |
              '(' <Constraint_Expression> <Constraint_Binary_Operator>
                     <Constraint_Expression> ')'

<Regular_Expression>             := <Widlcard>                           |
                                    <Variable_Match>                     |
                                    <Regex_Constant>                     |
                                    <Regex_Range>                        |
                                 <Regular_Expression> <Regular_Expression>

<Constraint_Unary_Operator>      := '~' | <Time_Shift_Operator>

<Constraint_Binary_Operator>     := '&' | '|'
```

```
<Time_Shift_Operator>          := <index_operator>

<Widlcard>                     := '?' | '+' | '*'

<Variable_Match>               := '@' '[' DIGIT ']'

<Regex_Range>                  := RGX_RNG

<Regex_Constant>               := NAME | <simple_char> | DIGIT |
                                  <escaped_char>

<simple_char>                  := SPACE | '$' | '~' | '`' | ''' |
                                  '{' | '!' | '%' | '-' | '}' |
                                  '>' | '<' | ',' | '.' | ':' |
                                  ';' | '/'

<escaped_char>                 := '\@' | '\^' | '\&' | '\*' | '\+' |
                                  '\?' | '\[' | '\]' | '\|'



<Optional>      := 'Section' 'Optional'
```

Appendix C

Example Descriptions

C.1 The SPAM VLIW-1 Architecture

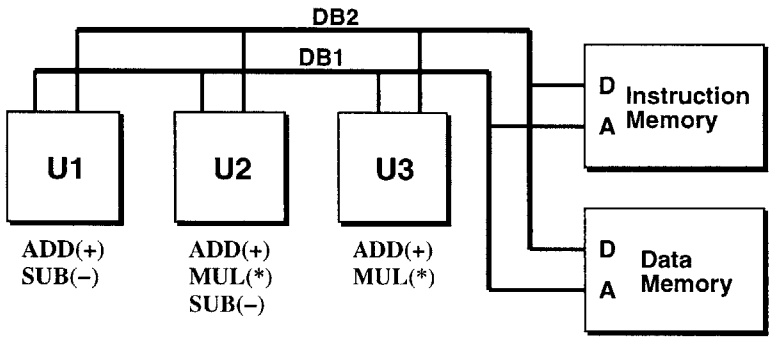

Figure C-1: **The SPAM VLIW-1 Architecture.**

```
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]} {} {} {} ;

Non_Terminal ADDR: INT  { $$ = INT; }  {INT}  {INT}  {} {} |
                   NAME { $$ = NAME; } {NAME} {NAME} {} {} ;

Non_Terminal OFFS: INT  { $$ = INT; }  {INT}  {INT}  {} {} |
                   NAME { $$ = NAME - CURRENT; }
                          {NAME - PC} {NAME - PC} {} {} ;

#define REG SRC
#define LOC DEST

Split.Const        U1.RB+U1.RC;
Split.Addr         DB1.SRC+DB1.DEST+DB2.SRC;
Split.Offs         DB1.SRC+DB1.DEST;
// ----------------------------------------------------------------

Section Storage

//                                        = entries , bits_per_entry
Instruction Memory INST                   = 0x10000 , 0x2C
Memory DM                                 = 0x10000 , 0x10
RegFile U1                                =     0x4 , 0x10
RegFile U2                                =     0x4 , 0x10
RegFile U3                                =     0x4 , 0x10
ProgramCounter PC                         =           0x10
```

```
// ------------------------------------------------------------------

#define DEFINE_NULL_OP  {} { NULLOP(); } {} {} {}

#define ADDm(x,y)     ADD(x,y,16,"trn")
#define SUBm(x,y)     SUB(x,y,16,"trn")
#define MULm(x,y)     MUL(x,y,16,16,"trn")
#define SEXTm(x,y)    SEXT(x,y,16)

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_addc U1_R, INT
                { U1.OP = 0x2; U1.RA = U1_R;
                  Split.Const = INT & 0xF; }
                { U1[U1_R] <- ADDm(U1[U1_R], SEXTm(INT, 4)); }
                {}
                { 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 DM_DATA, DB2.DEST gets code for DM_ADDR
        DM_ld   REG, LOC
                { DB1.SRC = DMdata; DB1.DEST = REG;
                  DB2.SRC = LOC; DB2.DEST = DMaddr; }
                { REG <- DM[LOC]; }
                {}
                { Cycle = 1; Size = 1; Stall = 1; }
                { Latency = 1; Usage = 1; }
// DB1.DEST gets code for DM_DATA, DB2.DEST gets code for DM_ADDR
        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 IM_DATA, DB2.DEST gets code for IM_ADDR
        IM_ld   REG, LOC
                { DB1.SRC = IMdata; DB1.DEST = REG;
                  DB2.SRC = LOC; DB2.DEST = IMaddr; }
                { REG <- INST[LOC]; }
                {}
                { Cycle = 2; Size = 1; Stall = 1; }
                { Latency = 1; Usage = 1; }
// DB1.DEST gets code for IM_DATA, DB2.DEST gets code for IM_ADDR
        IM_st   REG, LOC
                { DB1.SRC = REG; DB1.DEST = IMdata;
                  DB2.SRC = LOC; DB2.DEST = IMaddr; }
                { INST[LOC] <- REG; }
                {}
```

```
                { Cycle = 2; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }

// Control section
// Note that this uses the DB fields to perform control.
// We have available to us all codes where DB2.DEST = 0x10 - 0x1E
// inclusive. Because the above codes are unique we can do anything
// we want with DB2.SRC. We will take over the DB1 bitfields as well
// to store constants

Field Control:
      C_NULL DEFINE_NULL_OP
      C_jump ADDR
                { DB2.DEST = 0x10; Split.Addr = ADDR; }
                { PC <- ADDR; }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
      C_brz SRC, OFFS
                { DB2.DEST = 0x11; Split.Offs = OFFS; DB2.SRC = SRC; }
                { if (SRC == 0) { PC <- PC + SEXTm(OFFS, 10); }; }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
      C_brp SRC, OFFS
                { DB2.DEST = 0x12; Split.Offs = OFFS; DB2.SRC = SRC; }
                { if (SEXT(SRC,16,32) > 0)
                    { PC <- PC + SEXTm(OFFS, 10); }; }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
      C_brn SRC, OFFS
                { DB2.DEST = 0x13; Split.Offs = OFFS; DB2.SRC = SRC; }
                { if (SEXT(SRC,16,32) < 0)
                    { PC <- PC + SEXTm(OFFS, 10); }; }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
      C_brnz SRC, OFFS
                { DB2.DEST = 0x14; Split.Offs = OFFS; DB2.SRC = SRC; }
                { if (SRC != 0) { PC <- PC + SEXTm(OFFS, 10); }; }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
      C_rep SRC, INT
```

```
                { DB2.DEST = 0x15; Split.Offs = INT; DB2.SRC = SRC; }
                { for(SRC <- INT; SRC > 0; SRC <- SRC+1;) {
                    EVAL(PC+1);
                    };
                  }
                {}
                { Cycle = INT; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
      C_halt
                { DB2.DEST = 0x16; }
                { HALT(); }
                {}
                { 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* )

// Can not use buses for a move between register files if a memory
// operation is using the buses
~( ((DB*_move* *,*) | (DB*_nop))& ((DM_* *) | (IM_* *)) )

// Can not do both a DM and IM operation - they use the same buses
~( (DM_* *) & (IM_* *) )

// Can not write to same register from two different operations
~( (DB1_move *,@[1]) & (DB2_move *,@[1]) )

// Can not do any control operations with IM, DM or DB2 operations
~( (C_* *) & (((DB2_move* *) | (DM_* *)) | (IM_* *)))

// Some Control operations cannot be done with DB1 operations
~( (DB1_move* *) & ((((((C_jump *)  |
                        (C_brz *))  |
                        (C_brp *))  |
                        (C_brn *))  |
                        (C_brnz *)) |
                        (C_rep *)))

// ------------------------------------------------------------------
```

```
Section Optional
```

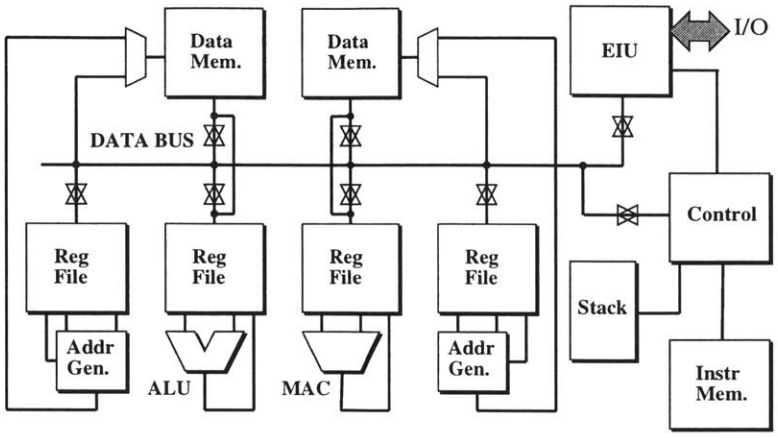

Figure C-2: **The SPAM VLIW-2 Architecture.**

C.2 The SPAM VLIW-2 Architecture

```
// Conventions:

// | = OR
// & = AND
// [x..y] = range from x to y
// , = used between required fields
// @ = variable declaration follows

// \ = ignore special symbol


// ---------------------------------------------------------------
// Number of fields in each VLIW word
// Size and format of each

Section Format

Control = OP[4], RI[6];
AG2     = OP[4], RW[3], RA[3], RB[3], RMEM[3];
AG1     = OP[4], RW[3], RA[3], RB[3], RMEM[3];
ALU     = OP[4], RW[5], RA[5], RB[5], RMEM[5];
MAC     = OP[3], RW[4], RA[4], RB[4], RMEM[4];
DB      = SRC[4], SINK[4];
ALUOP   = OP[2];
MACOP   = OP[2];
DM1A    = OP[1];
DM2A    = OP[1];
```

```
// ------------------------------------------------------------------

Section Global_Definitions

// This ends up in the lex file.
//       assembly         token    value
Token   "AG1.R"[0..7]    AG1_R    { [0..7]; };
Token   "AG2.R"[0..7]    AG2_R    { [0..7]; };
Token   "ALU.R"[0..31]   ALU_R    { [0..31]; };
Token   "MAC.R"[0..15]   MAC_R    { [0..15]; };
Token   EIUCR            EIUCRt   { };
Token   EIUAR            EIUARt   { };
Token   EIUDR            EIUDRt   { };
Token   CTRL             CTRLt    { };
Token   IDLE             IDLE     { };


// This ends up in the yacc file.
//             assembly          token    value
Non_Terminal RI:            AG1_R    {$$ = AG1_R; }
                                      { AG1[AG1_R] } {} {} {} |
                            AG2_R    {$$ = AG2_R|8; }
                                      { AG2[AG2_R] } {} {} {} |
                            ALU_R    {$$ = ALU_R|32; }
                                      { ALU[ALU_R] } {} {} {} |
                            MAC_R    {$$ = MAC_R|16; }
                                      { MAC[MAC_R] } {} {} {} ;
Non_Terminal TI:            INT      {$$ = INT; } { INT }   {} {} {} ;
Non_Terminal AG2_RA:        AG2_R    {$$ = AG2_R;}
                                     {AG2[AG2_R]} {} {} {} ;
Non_Terminal AG2_RB:        AG2_R    {$$ = AG2_R;}
                                     {AG2[AG2_R]} {} {} {} ;
Non_Terminal AG2_RMEM:      AG2_R    {$$ = AG2_R;}
                                     {AG2[AG2_R]} {} {} {} ;
Non_Terminal AG2_RW:        AG2_R    {$$ = AG2_R;}
                                     {AG2[AG2_R]} {} {} {} ;
Non_Terminal AG1_RA:        AG1_R    {$$ = AG1_R;}
                                     {AG1[AG1_R]} {} {} {} ;
Non_Terminal AG1_RB:        AG1_R    {$$ = AG1_R;}
                                     {AG1[AG1_R]} {} {} {} ;
Non_Terminal AG1_RMEM:      AG1_R    {$$ = AG1_R;}
                                     {AG1[AG1_R]} {} {} {} ;
Non_Terminal AG1_RW:        AG1_R    {$$ = AG1_R;}
```

```
                                {AG1[AG1_R]} {} {} {} ;
Non_Terminal ALU_RA:        ALU_R   {$$ = ALU_R;}
                                {ALU[ALU_R]} {} {} {} ;
Non_Terminal ALU_RB:        ALU_R   {$$ = ALU_R;}
                                {ALU[ALU_R]} {} {} {} ;
Non_Terminal ALU_RMEM:      ALU_R   {$$ = ALU_R;}
                                {ALU[ALU_R]} {} {} {} ;
Non_Terminal ALU_RW:        ALU_R   {$$ = ALU_R;}
                                {ALU[ALU_R]} {} {} {} ;
Non_Terminal MAC_RA:        MAC_R   {$$ = MAC_R;}
                                {MAC[MAC_R]} {} {} {} ;
Non_Terminal MAC_RB:        MAC_R   {$$ = MAC_R;}
                                {MAC[MAC_R]} {} {} {} ;
Non_Terminal MAC_RMEM:      MAC_R   {$$ = MAC_R;}
                                {MAC[MAC_R]} {} {} {} ;
Non_Terminal MAC_RW:        MAC_R   {$$ = MAC_R;}
                                {MAC[MAC_R]} {} {} {} ;

Non_Terminal OFFSET:        INT     {$$ = INT; } { INT }   {} {} {} |
                            NAME    {$$ = NAME - CURRENT; }
                                     { NAME - PC } {} {} {} ;

Non_Terminal ADDR:          INT     {$$ = INT; } { INT }   {} {} {} |
                            NAME    {$$ = NAME; } { NAME } {} {} {} ;

Non_Terminal DATA:          INT     {$$ = INT; } { INT } {} {} {} ;
Non_Terminal CONST:         INT     {$$ = INT; } { INT } {} {} {} ;
Non_Terminal SRC:           AG1_R   {DB.SRC = 0x2; AG1.RMEM = AG1_R; }
                                    {AG1[AG1_R]} {} {} {} |
                            AG2_R   {DB.SRC = 0x3; AG2.RMEM = AG2_R; }
                                    {AG2[AG2_R]} {} {} {} |
                            ALU_R   {DB.SRC = 0x4; ALU.RMEM = ALU_R; }
                                    {ALU[ALU_R]} {} {} {} |
                            MAC_R   {DB.SRC = 0x5; MAC.RMEM = MAC_R; }
                                    {MAC[MAC_R]} {} {} {} |
                            DATA    {DB.SRC = 0x6; Split.DATAs = DATA;}
                                    {DATA}  {} {} {} |
                            EIUCRt  {DB.SRC = 0x7;} {EIUCR} {} {} {} |
                            EIUARt  {DB.SRC = 0x8;} {EIUAR} {} {} {} |
                            EIUDRt  {DB.SRC = 0x9;} {EIUDR} {} {} {} |
                            CTRLt   {DB.SRC = 0xA;} {CTRL}  {} {} {} ;

Non_Terminal SRC1:          AG1_R   {DB.SRC = 0x2; AG1.RMEM = AG1_R; }
                                    {AG1[AG1_R]} {} {} {} |
                            AG2_R   {DB.SRC = 0x3; AG2.RMEM = AG2_R; }
```

```
                                {AG2[AG2_R]} {} {} {} |
                        ALU_R   {DB.SRC = 0x4; ALU.RMEM = ALU_R; }
                                {ALU[ALU_R]} {} {} {} |
                        MAC_R   {DB.SRC = 0x5; MAC.RMEM = MAC_R; }
                                {MAC[MAC_R]} {} {} {} |
                        EIUCRt  {DB.SRC = 0x7;} {EIUCR} {} {} {} |
                        EIUARt  {DB.SRC = 0x8;} {EIUAR} {} {} {} |
                        EIUDRt  {DB.SRC = 0x9;} {EIUDR} {} {} {} |
                        CTRLt   {DB.SRC = 0xA;} {CTRL}  {} {} {} ;

Non_Terminal SINK:      AG1_R   {DB.SINK = 0x2; AG1.RMEM = AG1_R; }
                                {AG1[AG1_R]} {} {} {} |
                        AG2_R   {DB.SINK = 0x3; AG2.RMEM = AG2_R; }
                                {AG2[AG2_R]} {} {} {} |
                        ALU_R   {DB.SINK = 0x4; ALU.RMEM = ALU_R; }
                                {ALU[ALU_R]} {} {} {} |
                        MAC_R   {DB.SINK = 0x5; MAC.RMEM = MAC_R; }
                                {MAC[MAC_R]} {} {} {} |
                        EIUCRt  {DB.SINK = 0x6;} {EIUCR} {} {} {} |
                        EIUARt  {DB.SINK = 0x7;} {EIUAR} {} {} {} |
                        EIUDRt  {DB.SINK = 0x8;} {EIUDR} {} {} {} |
                        CTRLt   {DB.SINK = 0x9;} {CTRL}  {} {} {} |
                        IDLE    {DB.SINK = 0xC;} {NULL}  {} {} {} ;


Split.OFFSETs    AG2.OP+AG2.RW+AG2.RA+AG2.RB+AG2.RMEM;
Split.ADDRs      Control.RI+AG2.OP+AG2.RW+AG2.RA+AG2.RB
                 +AG2.RMEM;
Split.DATAs      AG2.OP+AG2.RW+AG2.RA+AG2.RB+AG2.RMEM
                 +AG1.OP+AG1.RW+AG1.RA+AG1.RB+AG1.RMEM;
Split.CONSTs     ALU.RB+ALU.RMEM;

// ---------------------------------------------------------------
// The number of registers for each unit and the topology of
// any of the register files and memories.

Section Storage

//                              = depth , width
Instruction Memory IM           = 0x100000 , 0x63
Memory DM1                      = 0x100000 , 0x20
Memory DM2                      = 0x100000 , 0x20
RegFile ALU                     = 0x20 , 0x20
RegFile MAC                     = 0x10 , 0x20
RegFile AG1                     = 0x8 , 0x20
```

```
RegFile AG2                     = 0x8 , 0x20
Register ACC                    = 0x20
Register SP                     = 0x4           // stack pointer
CRegister EIUCR                 = 0x20
CRegister EIUAR                 = 0x20
CRegister EIUDR                 = 0x20
CRegister CTRL                  = 0x20
CRegister JR                    = 0x20          // jump register
ProgramCounter PC               = 0x20          // program counter
Stack STACK(SP)                 = 0x10 , 0x20

// ---------------------------------------------------------------
// Correspondence between assembly mneumonics, bitfields, and actual
// instructions.

Section Instruction_Set

//                          RTL Descriptions
// NOTES:
//
// 1) There is the possibility that one instruction might have two
// different RTL descriptions based on some piece of state that is
// visible to the compiler. In this case we can use a CASE statement
// selecting over such state to describe which RTL description is
// appropriate for such an instruction. Note that conditional control
// flow instructions make use of this construct.
//
// 2) To describe the flow of control, we describe the values received
// by a specially declared register (declared in the storage section)
// usually called the PC. The value received by this register is the
// address from which the next instruction will be fetched.
//
// 3) NULL is a specially defined register which returns an arbitrary
// value when read and has no effect when written.
//
// here are the definitions of the operators

#define DEFINE_NULL_OP  {} { NULLOP(); } {} {} {}

#define ext3(x)      EXT(x,3,32)
#define ext6(x)      EXT(x,6,32)
#define ext10(x)     EXT(x,10,32)
#define ext22(x)     EXT(x,22,32)
#define sx16(x)      SEXT(x,16,32)
#define sx10(x)      SEXT(x,10,32)
```

```
#define ADDm(x,y)    ADD(x,y,32,"trn")
#define SUBm(x,y)    SUB(x,y,32,"trn")
#define MULm(x,y)    MUL(x,y,32,32,"trn")
#define DIVm(x,y)    DIV(x,y,32,32,32,"trn")
#define IORm(x,y)    OR(x,y,32)
#define ANDm(x,y)    AND(x,y,32)
#define NOTm(x)      NOT(x,32)
#define XORm(x,y)    XOR(x,y,32)
#define LSLm(x,y)    ASL(x,y,0,NULL,32)
#define LSRm(x,y)    ASR(x,y,0,NULL,32)
#define FADDm(x,y)   FADD(x,y,8,24,"sat")
#define FMULm(x,y)   FMUL(x,y,8,24,"sat")

Field Control:
        Control_NULL            DEFINE_NULL_OP
        Control_NOP             { Control.OP = 0x0 ; }
                                { NOP(); }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; }
        Control_call            { Control.OP = 0x1 ; }
                                { STACK[SP] <- PC + 1;
                                  SP <- SP + 1; PC <- JR ; }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        Control_rtn             { Control.OP = 0x2 ; }
                                { SP <- SP - 1; PC <- STACK[SP]; }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        Control_jump            { Control.OP = 0x3 ; }
                                { PC <- JR ; }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        Control_jumpcz RI       { Control.OP = 0x4 ;
                                  Control.RI = RI ; }
                                { if (RI == 0) { PC <- JR ; }; }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        Control_jumpcp RI       { Control.OP = 0x5 ;
                                  Control.RI = RI ; }
                                { if (RI > 0) { PC <- JR ; }; }
```

```
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_rti             { Control.OP = 0x6 ; }
                        { SP <- SP - 1; PC <- STACK[SP]; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_trap TI         { Control.OP = 0x7 ;
                          Control.RI = TI ; }
                        { STACK[SP] <- PC + 1;
                          SP <- SP + 1;
                          PC <-  SUBm(ext6(TI), 0x00000002); }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_rtt             { Control.OP = 0x8 ; }
                        { SP <- SP - 1; PC <- STACK[SP]; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_br OFFSET       { Control.OP = 0x9 ;
                          Split.OFFSETs = OFFSET ; }
                        { PC <-  ADDm(PC, sx16(OFFSET)) ; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_brcz RI,OFFSET  { Control.OP = 0xA ;
                          Control.RI = RI ;
                          Split.OFFSETs = OFFSET ; }
                        { if (RI == 0)
                            { PC <- ADDm(PC, sx16(OFFSET)) ; }
                          else
                            { PC <-  PC + 1 ; } ; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_brcp RI,OFFSET  { Control.OP = 0xB ;
                          Control.RI = RI ;
                          Split.OFFSETs = OFFSET ; }
                        { if (RI > 0)
                            { PC <- ADDm(PC, sx16(OFFSET));};}
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
```

```
        Control_ldjr ADDR       { Control.OP = 0xC ;
                                  Split.ADDRs = ADDR ; }
                                { JR <- ext22(ADDR); }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        Control_ldjrpc          { Control.OP = 0xD ; }
                                { JR <- PC ; }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        Control_HALT            { Control.OP = 0xE ; }
                                { HALT(); }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }

// ////////////////////////////////////////////////////////////////

Field ALUf:
        ALU_NULL                                    DEFINE_NULL_OP
        ALU_add     ALU_RA,ALU_RB,ALU_RW            { ALU.OP = 0x0 ;
                                                      ALU.RA = ALU_RA ;
                                                      ALU.RB = ALU_RB ;
                                                      ALU.RW = ALU_RW ; }
                                  { ALU_RW <-  ADDm(ALU_RA, ALU_RB);}
                                  {}
                                  { Cycle=1; Size=1; Stall=0; }
                                  { Latency=1; Usage=1; }
        ALU_addc    ALU_RA,CONST,ALU_RW             { ALU.OP = 0x1 ;
                                                      ALU.RA = ALU_RA ;
                                                      Split.CONSTs = CONST;
                                                      ALU.RW = ALU_RW ; }
                            { ALU_RW <-  ADDm(ALU_RA, sx10(CONST)); }
                                  {}
                                  { Cycle=1; Size=1; Stall=0; }
                                  { Latency=1; Usage=1; }
        ALU_sub     ALU_RA,ALU_RB,ALU_RW            { ALU.OP = 0x2 ;
                                                      ALU.RA = ALU_RA ;
                                                      ALU.RB = ALU_RB ;
                                                      ALU.RW = ALU_RW ; }
                                  { ALU_RW <-  SUBm(ALU_RA, ALU_RB);}
                                  {}
                                  { Cycle=1; Size=1; Stall=0; }
```

```
                        { Latency=1; Usage=1; }
ALU_subc    ALU_RA,CONST,ALU_RW       { ALU.OP = 0x3 ;
                                        ALU.RA = ALU_RA ;
                                        Split.CONSTs = CONST;
                                        ALU.RW = ALU_RW ; }
                  { ALU_RW <-  SUBm(ALU_RA, sx10(CONST)); }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_mul     ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0x4 ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
                                        ALU.RW = ALU_RW ; }
                        { ALU_RW <-  MULm(ALU_RA, ALU_RB);}
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_mulc    ALU_RA,CONST,ALU_RW       { ALU.OP = 0x5 ;
                                        ALU.RA = ALU_RA ;
                                        Split.CONSTs = CONST;
                                        ALU.RW = ALU_RW ; }
                  { ALU_RW <-  MULm(ALU_RA, sx10(CONST)); }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_div     ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0x6 ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
                                        ALU.RW = ALU_RW ; }
                        { ALU_RW <-  DIVm(ALU_RA, ALU_RB);}
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_and     ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0x7 ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
                                        ALU.RW = ALU_RW ; }
                        { ALU_RW <-  ANDm(ALU_RA, ALU_RB);}
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_or      ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0x8 ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
                                        ALU.RW = ALU_RW ; }
```

```
                          { ALU_RW <-  IORm(ALU_RA, ALU_RB);}
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_not     ALU_RA,ALU_RW                { ALU.OP = 0x9 ;
                                           ALU.RA = ALU_RA ;
                                           ALU.RW = ALU_RW ; }
                                 { ALU_RW <-  NOTm(ALU_RA);}
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_xor     ALU_RA,ALU_RB,ALU_RW         { ALU.OP = 0xA ;
                                           ALU.RA = ALU_RA ;
                                           ALU.RB = ALU_RB ;
                                           ALU.RW = ALU_RW ; }
                          { ALU_RW <-  XORm(ALU_RA, ALU_RB);}
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_pass    ALU_RA,ALU_RW                { ALU.OP = 0xB ;
                                           ALU.RA = ALU_RA ;
                                           ALU.RW = ALU_RW ; }
                                    { ALU_RW <- ALU_RA ; }
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_lsl     ALU_RA,ALU_RB,ALU_RW         { ALU.OP = 0xC ;
                                           ALU.RA = ALU_RA ;
                                           ALU.RB = ALU_RB ;
                                           ALU.RW = ALU_RW ; }
                         { ALU_RW <-  LSLm(ALU_RA, ALU_RB);}
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_lslc    ALU_RA,CONST,ALU_RW          { ALU.OP = 0xD ;
                                           ALU.RA = ALU_RA ;
                                           Split.CONSTs = CONST;
                                           ALU.RW = ALU_RW ; }
                    { ALU_RW <-  LSLm(ALU_RA, ext10(CONST)); }
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_lsr     ALU_RA,ALU_RB,ALU_RW         { ALU.OP = 0xE ;
                                           ALU.RA = ALU_RA ;
                                           ALU.RB = ALU_RB ;
```

```
                                              ALU.RW = ALU_RW ; }
                                  { ALU_RW <-  LSRm(ALU_RA, ALU_RB);}
                                   {}
                                   { Cycle=1; Size=1; Stall=0; }
                                   { Latency=1; Usage=1; }
        ALU_lsrc    ALU_RA,CONST,ALU_RW       { ALU.OP = 0xF ;
                                                ALU.RA = ALU_RA ;
                                                Split.CONSTs = CONST;
                                                ALU.RW = ALU_RW ; }
                        { ALU_RW <-  LSRm(ALU_RA, ext10(CONST)); }
                                   {}
                                   { Cycle=1; Size=1; Stall=0; }
                                   { Latency=1; Usage=1; }

// ////////////////////////////////////////////////////////////////

Field MACf:
        MAC_NULL                                  DEFINE_NULL_OP
        MAC_mac     MAC_RA,MAC_RB                 { MAC.OP = 0x0 ;
                                                    MAC.RA = MAC_RA ;
                                                    MAC.RB = MAC_RB ; }
                     { ACC <- FADDm(ACC, FMULm(MAC_RB, MAC_RA));}
                                   {}
                                   { Cycle=1; Size=1; Stall=0; }
                                   { Latency=1; Usage=1; }
        MAC_macw    MAC_RA,MAC_RB,MAC_RW          { MAC.OP = 0x1 ;
                                                    MAC.RA = MAC_RA ;
                                                    MAC.RB = MAC_RB ;
                                                    MAC.RW = MAC_RW ; }
                     { ACC <- FADDm(ACC, FMULm(MAC_RB, MAC_RA));
                       MAC_RW <- FADDm(ACC, FMULm(MAC_RB, MAC_RA));}
                                   {}
                                   { Cycle=1; Size=1; Stall=0; }
                                   { Latency=1; Usage=1; }
        MAC_add     MAC_RA,MAC_RB                 { MAC.OP = 0x2 ;
                                                    MAC.RA = MAC_RA ;
                                                    MAC.RB = MAC_RB ; }
                                     { ACC <- FADDm(MAC_RA, MAC_RB);}
                                   {}
                                   { Cycle=1; Size=1; Stall=0; }
                                   { Latency=1; Usage=1; }
        MAC_addw    MAC_RA,MAC_RB,MAC_RW          { MAC.OP = 0x3 ;
                                                    MAC.RA = MAC_RA ;
                                                    MAC.RB = MAC_RB ;
```

```
                                                MAC.RW = MAC_RW ; }
                                  { ACC <- FADDm(MAC_RA, MAC_RB);
                                 MAC_RW <- FADDm(MAC_RA, MAC_RB);}
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        MAC_mul     MAC_RA,MAC_RB              { MAC.OP = 0x4 ;
                                                 MAC.RA = MAC_RA ;
                                                 MAC.RB = MAC_RB ; }
                                  { ACC <- FMULm(MAC_RA, MAC_RB);}
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        MAC_mulw    MAC_RA,MAC_RB,MAC_RW       { MAC.OP = 0x5 ;
                                                 MAC.RA = MAC_RA ;
                                                 MAC.RB = MAC_RB ;
                                                 MAC.RW = MAC_RW ; }
                                  { ACC <- FMULm(MAC_RA, MAC_RB);
                                 MAC_RW <- FMULm(MAC_RA, MAC_RB);}
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        MAC_lda     MAC_RA                     { MAC.OP = 0x6 ;
                                                 MAC.RA = MAC_RA ; }
                                { ACC <- MAC_RA ; }
                                {}
                                { cycle=1; size=1; stall=0; }
                                { latency=1; Usage=1; }
        MAC_clr                 { MAC.OP = 0x7 ; }
                                { ACC <- 0x00000000 ; }
                                {}
                                { cycle=1; size=1; stall=0; }
                                { latency=1; Usage=1; }

// ////////////////////////////////////////////////////////////////

Field DB:
        DB_NULL                 DEFINE_NULL_OP
        DB_move     SRC,SINK    { } { SINK <- SRC ; }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }

// ////////////////////////////////////////////////////////////////
```

```
// Address Generator 1 register updates

Field AG1f:
        AG1_NULL        DEFINE_NULL_OP
        AG1_addbc       AG1_RB,CONST,AG1_RW       { AG1.OP = 0x0 ;
                                                    AG1.RB = AG1_RB;
                                                    AG1.RMEM = CONST;
                                                    AG1.RW = AG1_RW;}
                           { AG1_RW <-  ADDm(AG1_RB, ext3(CONST));}
                           {}
                           { Cycle=1; Size=1; Stall=0; }
                           { Latency=1; Usage=1; }
        AG1_addac       AG1_RA,CONST,AG1_RW       { AG1.OP = 0x1 ;
                                                    AG1.RA = AG1_RA;
                                                    AG1.RMEM = CONST;
                                                    AG1.RW = AG1_RW;}
         { AG1_RW <-  ADDm(AG1_RA, ext3(CONST)) ; }
                           {}
                           { Cycle=1; Size=1; Stall=0; }
                           { Latency=1; Usage=1; }
        AG1_inc         AG1_RB,AG1_RW             { AG1.OP = 0x2 ;
                                                    AG1.RB = AG1_RB;
                                                    AG1.RW = AG1_RW;}
         { AG1_RW <-  ADDm(AG1_RB, 0x00000001) ; }
                           {}
                           { Cycle=1; Size=1; Stall=0; }
                           { Latency=1; Usage=1; }
        AG1_add         AG1_RA,AG1_RB,AG1_RW      { AG1.OP = 0x3 ;
                                                    AG1.RA = AG1_RA;
                                                    AG1.RB = AG1_RB;
                                                    AG1.RW = AG1_RW;}
                     { AG1_RW <-  ADDm(AG1_RB,AG1_RA) ; }
                           {}
                           { Cycle=1; Size=1; Stall=0; }
                           { Latency=1; Usage=1; }
        AG1_idle                                  { AG1.OP = 0x4;}
                           {}
                           {}
                           { Cycle=1; Size=1; Stall=0; }
                           { Latency=1; Usage=1; }

// ////////////////////////////////////////////////////////////////
// Address Generator 2 register updates

Field AG2f:
```

```
        AG2_NULL        DEFINE_NULL_OP
        AG2_addbc       AG2_RB,CONST,AG2_RW      { AG2.OP = 0x0;
                                                   AG2.RB = AG2_RB;
                                                   AG2.RMEM = CONST;
                                                   AG2.RW = AG2_RW;}
         { AG2_RW <-  ADDm(AG2_RB, ext3(CONST)) ; }
                         {}
                         { Cycle=1; Size=1; Stall=0; }
                         { Latency=1; Usage=1; }
        AG2_addac       AG2_RA,CONST,AG2_RW      { AG2.OP = 0x1;
                                                   AG2.RA = AG2_RA;
                                                   AG2.RMEM = CONST;
                                                   AG2.RW = AG2_RW;}
         { AG2_RW <-  ADDm(AG2_RA, ext3(CONST)) ; }
                         {}
                         { Cycle=1; Size=1; Stall=0; }
                         { Latency=1; Usage=1; }
        AG2_inc         AG2_RB,AG2_RW            { AG2.OP = 0x2;
                                                   AG2.RB = AG2_RB;
                                                   AG2.RW = AG2_RW;}
         { AG2_RW <-  ADDm(AG2_RB, 0x00000001) ; }
                         {}
                         { Cycle=1; Size=1; Stall=0; }
                         { Latency=1; Usage=1; }
        AG2_add         AG2_RA,AG2_RB,AG2_RW     { AG2.OP = 0x3;
                                                   AG2.RA = AG2_RA;
                                                   AG2.RB = AG2_RB;
                                                   AG2.RW = AG2_RW;}
                    { AG2_RW <-  ADDm(AG2_RB,AG2_RA) ; }
                         {}
                         { Cycle=1; Size=1; Stall=0; }
                         { Latency=1; Usage=1; }
        AG2_idle                                 { AG2.OP = 0x4;}
                         {}
                         {}
                         { Cycle=1; Size=1; Stall=0; }
                         { Latency=1; Usage=1; }

// ////////////////////////////////////////////////////////////////
// Data Memory 1 operations

Field DM1f:
        DM1_NULL            DEFINE_NULL_OP
        DM1_dir_save_i      ALU_RMEM, AG1_RA
                    { ALU.RMEM = ALU_RMEM;
```

```
                ALUOP.OP = 0x0;
                DM1A.OP = 0x0;
                AG1.RA = AG1_RA;
                AG1.OP=AG1.OP+0x0;}
              { DM1[AG1_RA] <- ALU_RMEM; }
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM1_dir_save_io     ALU_RMEM, AG1_RA, AG1_RB
              { ALU.RMEM = ALU_RMEM;
                ALUOP.OP = 0x0;
                DM1A.OP = 0x0;
                AG1.RA = AG1_RA;
                AG1.RB = AG1_RB;
                AG1.OP=AG1.OP+0x5;}
              { DM1[ ADDm(AG1_RA, AG1_RB)] <- ALU_RMEM; }
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM1_dir_save_ic     ALU_RMEM, AG1_RA, CONST
              { ALU.RMEM = ALU_RMEM;
                ALUOP.OP = 0x0;
                DM1A.OP = 0x0;
                AG1.RA = AG1_RA;
                AG1.RMEM = CONST;
                AG1.OP=AG1.OP+0xA;}
              { DM1[ ADDm(AG1_RA, ext3(CONST))] <- ALU_RMEM;}
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM1_dir_save_m      ALU_RMEM, SRC1
              { ALU.RMEM = ALU_RMEM;
                ALUOP.OP = 0x0;
                DM1A.OP = 0x1;
                DB.SINK = 0xA; }
              { DM1[SRC1] <- ALU_RMEM; }
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM1_dir_load_i      ALU_RMEM, AG1_RA
              { ALU.RMEM = ALU_RMEM;
```

```
                ALUOP.OP = 0x1;
                DM1A.OP = 0x0;
                AG1.RA = AG1_RA;
                AG1.OP=AG1.OP+0x0;}
              { ALU_RMEM <- DM1[AG1_RA] ; }
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM1_dir_load_io     ALU_RMEM, AG1_RA, AG1_RB
              { ALU.RMEM = ALU_RMEM;
                ALUOP.OP = 0x1;
                DM1A.OP = 0x0;
                AG1.RA = AG1_RA;
                AG1.RB = AG1_RB;
                AG1.OP=AG1.OP+0x5;}
              { ALU_RMEM <- DM1[ ADDm(AG1_RA, AG1_RB)] ; }
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM1_dir_load_ic     ALU_RMEM, AG1_RA, CONST
              { ALU.RMEM = ALU_RMEM;
                ALUOP.OP = 0x1;
                DM1A.OP = 0x0;
                AG1.RA = AG1_RA;
                AG1.RMEM = CONST;
                AG1.OP=AG1.OP+0xA;}
              { ALU_RMEM <- DM1[ ADDm(AG1_RA, ext3(CONST))];}
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM1_dir_load_m      ALU_RMEM, SRC1
              { ALU.RMEM = ALU_RMEM;
                ALUOP.OP = 0x1;
                DM1A.OP = 0x1;
                DB.SINK = 0xA; }
              { ALU_RMEM <- DM1[SRC1] ; }
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM1_bus_save_i      SRC1, AG1_RA
              { DB.SINK = 0x0;
```

```
              DM1A.OP = 0x0;
              AG1.RA = AG1_RA;
              AG1.OP=AG1.OP+0x0;}
            { DM1[AG1_RA] <- SRC1; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM1_bus_save_io     SRC1, AG1_RA, AG1_RB
            { DB.SINK = 0x0;
              DM1A.OP = 0x0;
              AG1.RA = AG1_RA;
              AG1.RB = AG1_RB;
              AG1.OP=AG1.OP+0x5;}
            { DM1[ ADDm(AG1_RA, AG1_RB)] <- SRC1; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM1_bus_save_ic     SRC1, AG1_RA, CONST
            { DB.SINK = 0x0;
              DM1A.OP = 0x0;
              AG1.RA = AG1_RA;
              AG1.RMEM = CONST;
              AG1.OP=AG1.OP+0xA;}
            { DM1[ ADDm(AG1_RA, ext3(CONST))] <- SRC1; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM1_bus_load_i      SINK, AG1_RA
            { DB.SRC = 0x0;
              DM1A.OP = 0x0;
              AG1.RA = AG1_RA;
              AG1.OP=AG1.OP+0x0;}
            { SINK <- DM1[AG1_RA] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM1_bus_load_io     SINK, AG1_RA, AG1_RB
            { DB.SRC = 0x0;
              DM1A.OP = 0x0;
              AG1.RA = AG1_RA;
              AG1.RB = AG1_RB;
```

```
                  AG1.OP=AG1.OP+0x5;}
                { SINK <- DM1[ ADDm(AG1_RA, AG1_RB)] ; }
                {}
                { Cycle=1; Size=1; Stall=0; }
                { Latency=1; Usage=1; }

        DM1_bus_load_ic     SINK, AG1_RA, CONST
                { DB.SRC = 0x0;
                  DM1A.OP = 0x0;
                  AG1.RA = AG1_RA;
                  AG1.RMEM = CONST;
                  AG1.OP=AG1.OP+0xA;}
                { SINK <- DM1[ ADDm(AG1_RA, ext3(CONST))];}
                {}
                { Cycle=1; Size=1; Stall=0; }
                { Latency=1; Usage=1; }

        DM1_idle    { ALUOP.OP = 0x2;
                  DM1A.OP = 0x1; }
                {}
                {}
                { Cycle=1; Size=1; Stall=0; }
                { Latency=1; Usage=1; }

// ////////////////////////////////////////////////////////////////
// Data Memory 2 operations

Field DM2f:
        DM2_NULL            DEFINE_NULL_OP
        DM2_dir_save_i      MAC_RMEM, AG2_RA
                { MAC.RMEM = MAC_RMEM;
                  MACOP.OP = 0x0;
                  DM2A.OP = 0x0;
                  AG2.RA = AG2_RA;
                  AG2.OP=AG2.OP+0x0;}
                { DM2[AG2_RA] <- MAC_RMEM; }
                {}
                { Cycle=1; Size=1; Stall=0; }
                { Latency=1; Usage=1; }

        DM2_dir_save_io     MAC_RMEM, AG2_RA, AG2_RB
                { MAC.RMEM = MAC_RMEM;
                  MACOP.OP = 0x0;
                  DM2A.OP = 0x0;
                  AG2.RA = AG2_RA;
```

```
              AG2.RB = AG2_RB;
              AG2.OP=AG2.OP+0x5;}
            { DM2[ ADDm(AG2_RA, AG2_RB)] <- MAC_RMEM; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_dir_save_ic     MAC_RMEM, AG2_RA, CONST
            { MAC.RMEM = MAC_RMEM;
              MACOP.OP = 0x0;
              DM2A.OP = 0x0;
              AG2.RA = AG2_RA;
              AG2.RMEM = CONST;
              AG2.OP=AG2.OP+0xA;}
            { DM2[ ADDm(AG2_RA, ext3(CONST))] <- MAC_RMEM;}
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_dir_save_m      MAC_RMEM, SRC1
            { MAC.RMEM = MAC_RMEM;
              MACOP.OP = 0x0;
              DM2A.OP = 0x1;
              DB.SINK = 0xB; }
            { DM2[SRC1] <- MAC_RMEM; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_dir_load_i      MAC_RMEM, AG2_RA
            { MAC.RMEM = MAC_RMEM;
              MACOP.OP = 0x1;
              DM2A.OP = 0x0;
              AG2.RA = AG2_RA;
              AG2.OP=AG2.OP+0x0;}
            { MAC_RMEM <- DM2[AG2_RA] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_dir_load_io     MAC_RMEM, AG2_RA, AG2_RB
            { MAC.RMEM = MAC_RMEM;
              MACOP.OP = 0x1;
              DM2A.OP = 0x0;
              AG2.RA = AG2_RA;
```

```
                AG2.RB = AG2_RB;
                AG2.OP=AG2.OP+0x5;}
              { MAC_RMEM <- DM2[ ADDm(AG2_RA, AG2_RB)];}
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM2_dir_load_ic     MAC_RMEM, AG2_RA, CONST
              { MAC.RMEM = MAC_RMEM;
                MACOP.OP = 0x1;
                DM2A.OP = 0x0;
                AG2.RA = AG2_RA;
                AG2.RMEM = CONST;
                AG2.OP=AG2.OP+0xA;}
              { MAC_RMEM <- DM2[ ADDm(AG2_RA, ext3(CONST))];}
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM2_dir_load_m      MAC_RMEM, SRC1
              { MAC.RMEM = MAC_RMEM;
                MACOP.OP = 0x1;
                DM2A.OP = 0x1;
                DB.SINK = 0xB; }
              { MAC_RMEM <- DM2[SRC1] ; }
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM2_bus_save_i      SRC1, AG2_RA
              { DB.SINK = 0x1;
                DM2A.OP = 0x0;
                AG2.RA = AG2_RA;
                AG2.OP=AG2.OP+0x0;}
              { DM2[AG2_RA] <- SRC1; }
              {}
              { Cycle=1; Size=1; Stall=0; }
              { Latency=1; Usage=1; }

DM2_bus_save_io     SRC1, AG2_RA, AG2_RB
              { DB.SINK = 0x1;
                DM2A.OP = 0x0;
                AG2.RA = AG2_RA;
                AG2.RB = AG2_RB;
                AG2.OP=AG2.OP+0x5;}
```

```
            { DM2[ ADDm(AG2_RA, AG2_RB)] <- SRC1; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_bus_save_ic     SRC1, AG2_RA, CONST
            { DB.SINK = 0x1;
              DM2A.OP = 0x0;
              AG2.RA = AG2_RA;
              AG2.RMEM = CONST;
              AG2.OP=AG2.OP+0xA;}
            { DM2[ ADDm(AG2_RA, ext3(CONST))] <- SRC1;}
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_bus_load_i      SINK, AG2_RA
            { DB.SRC = 0x1;
              DM2A.OP = 0x0;
              AG2.RA = AG2_RA;
              AG2.OP=AG2.OP+0x0;}
            { SINK <- DM2[AG2_RA] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_bus_load_io     SINK, AG2_RA, AG2_RB
            { DB.SRC = 0x1;
              DM2A.OP = 0x0;
              AG2.RA = AG2_RA;
              AG2.RB = AG2_RB;
              AG2.OP=AG2.OP+0x5;}
            { SINK <- DM2[ ADDm(AG2_RA, AG2_RB)] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_bus_load_ic     SINK, AG2_RA, CONST
            { DB.SRC = 0x1;
              DM2A.OP = 0x0;
              AG2.RA = AG2_RA;
              AG2.RMEM = CONST;
              AG2.OP=AG2.OP+0xA;}
            { SINK <- DM2[ ADDm(AG2_RA, ext3(CONST))];}
            {}
```

```
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM2_idle    { MACOP.OP = 0x2;
                      DM2A.OP = 0x1; }
                    {}
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

// ---------------------------------------------------------------
// Any restrictions (in the form of rules) of which instructions go
// together and which do not.

Section Constraints

// Can't do constant addressing memory mode with AG transfers on bus
// bitfield conflict on AG*.RMEM
~((DM1_*ic *) & ((DB_move *,AG1.R*) | (DB_move AG1.R*,*)))
~((DM2_*ic *) & ((DB_move *,AG2.R*) | (DB_move AG2.R*,*)))

// Can't do register updates with constant in AG with AG bus transfers
// bitfield conflict on AG*.RMEM
~((AG1_add?c *) & ((DB_move *,AG1.R*) | (DB_move AG1.R*,*)))
~((AG2_add?c *) & ((DB_move *,AG2.R*) | (DB_move AG2.R*,*)))

// Can't do ALU operations involving constants with ALU bus transfers
// bitfield conflict on ALU.RMEM
~((ALU_*c *) & ((DB_move *,ALU.R*) | (DB_move ALU.R*,*)))

// SRC and SINK cannot be the same in a data bus transfer
// units connected to the bus only have one port to it each
~(DB_move @[1].*,@[1].*)

// Cannot drive both memories on the bus at the same time.
// Bus conflict
~((DM1_bus_load* *) & (DM2_bus_load* *))

// Also cannot read both memories from the bus at the same time
// bitfiled conflict on DB.SINK
~((DM1_bus_save* *) & (DM2_bus_save* *))

// Cannot do a memory bus transfer and a normal bus transfer at the
// same time
~((DM?_bus* *) & (DB_move *))
```

```
// If you are doing an AG* register update with a constant and a
// memory access with a constant make sure the constants are the
// same - they share the same field AG*.RMEM
(~((DM1_*c *,@[1]) & (AG1_add?c *,@[1],*)) |
  ((DM1_*c *,@[1]) & (AG1_add?c *,@[1],*)) )


// If Instruction Memory (DATA) is the source then can't use AG1
// or AG2 or data memory transfers that use the address generators
// or transfer to an address generator file
// Bit conflict on all the fields of AG1 and AG2
~((DB_move [0-9]+, *) &
  ((((AG?_* *)      |
   (DM?_bus* *))     |
   (DM?_dir_*_i*))   |
   (DB_move [0-9]+, AG*)))

// load jump register and branch instructions can't be done with
// AG2 instructions, DM2 operations that use the AG2 or a transfer
// to or from the AG2 regfile
// Bitfield conflict on the AG2 subfields
~(((((Control_ldjr *)        |
   (Control_br *))           |
   (Control_brcz *))         |
   (Control_brcp *))
&
  (((((AG2_* *)              |
   (DM2_bus* *))             |
   (DM2_dir_*_i*))           |
   (DB_move *,AG2.R*))       |
   (DB_move AG2.R*,*)))

// Can't write to same registers from both execution unit operation
// and  DB or transfer path
// Example: If ALU write back to ALU.R1 then DB_move *,ALU.R1 is
// not allowed.
// conflict at the register storage set lines
~((AG?_add* *,*,@[1]) & (DB_move *,@[1]))
~((AG?_inc *,*,@[1]) & (DB_move *,@[1]))
~((MAC_*w *,*,@[1]) & (DB_move *,@[1]))
~((ALU_* *,*,@[1]) & (DB_move *,@[1]))

// ---------------------------------------------------------------
```

Section Optional

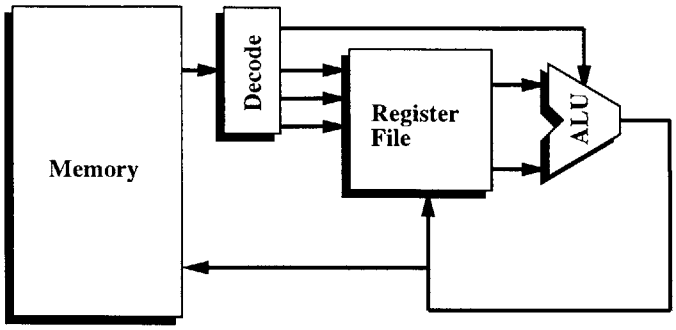

Figure C-3: **The SPAM RISC Architecture.**

C.3 The SPAM RISC Architecture

```
// This is a RISC-style architecture (unifunctional)
// it allows us to explore issues with control flow and calling
// conventions
//
// NOTES:
// 1) RF[31] is used to store the PC on calls
// 2) RF[30] is used as the stack pointer in this modified version

Section Format

// Only one field for the instruction word (unifunctional)

W = MODE[4], OP[5], RA[5], RB[5], RC[5];

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

Section Global_Definitions

//    assembly      token       value
Token "R"[0..31]    Reg         { [0..31]; };

Non_Terminal RA:    Reg  { $$ = Reg; } {RF[Reg]} {} {} {};
Non_Terminal RB:    Reg  { $$ = Reg; } {RF[Reg]} {} {} {};
Non_Terminal RC:    Reg  { $$ = Reg; } {RF[Reg]} {} {} {};

Non_Terminal ADDR:  INT  { $$ = INT; }  {INT}  {} {} {} |
                    NAME { $$ = NAME; } {NAME} {} {} {};

Non_Terminal OFFS:  INT  { $$ = INT; }  {INT}  {} {} {} |
                    NAME { $$ = NAME-CURRENT; } {NAME-PC} {} {} {};
```

```
Split.Const         W.RA+W.RB;
Split.ADDRs         W.OP+W.RA+W.RB+W.RC;
Split.OFFSs         W.OP+W.RA+W.RB;
// ---------------------------------------------------------------

Section Storage

//                                      =    entries , bits_per_entry
Instruction Memory IM                   =  0x1000000 , 0x18
RegFile RF                              =       0x20 , 0x18
ProgramCounter PC                       =            0x18

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

#define ADDm(x,y)    ADD(x,y,24,"trn")
#define SUBm(x,y)    SUB(x,y,24,"trn")

Section Instruction_Set

Field Main:
        // data processing operations - 3-address
        add RA, RB, RC
                { W.MODE = 0x0; W.OP = 0x1; W.RA = RA;
                  W.RB = RB; W.RC = RC; }
                { RC <- ADDm(RA, RB); }
                {}
                { Cycle = 1; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

        sub RA, RB, RC
                { W.MODE = 0x0; W.OP = 0x2; W.RA = RA;
                  W.RB = RB; W.RC = RC; }
                { RC <- SUBm(RA, RB); }
                {}
                { Cycle = 1; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

        and RA, RB, RC
                { W.MODE = 0x0; W.OP = 0x3; W.RA = RA;
                  W.RB = RB; W.RC = RC; }
                { RC <- AND(RA, RB, 24); }
                {}
                { Cycle = 1; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }
```

```
or RA, RB, RC
        { W.MODE = 0x0; W.OP = 0x4; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { RC <- OR(RA, RB, 24); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

xor RA, RB, RC
        { W.MODE = 0x0; W.OP = 0x5; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { RC <- XOR(RA, RB, 24); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

asl RA, RB, RC
        { W.MODE = 0x0; W.OP = 0x6; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { RC <- RA << RB; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

asr RA, RB, RC
        { W.MODE = 0x0; W.OP = 0x7; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { RC <- RA >> RB; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

// data processing operations - 2-address and constant
addc INT, RB, RC
        { W.MODE = 0x0; W.OP = 0x8; W.RA = INT;
          W.RB = RB; W.RC = RC; }
        { RC <- ADDm(RB, INT); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

subc INT, RB, RC
        { W.MODE = 0x0; W.OP = 0x9; W.RA = INT;
          W.RB = RB; W.RC = RC; }
```

```
        { RC <- SUBm(RB, INT); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

aslc INT, RB, RC
        { W.MODE = 0x0; W.OP = 0xA; W.RA = INT;
          W.RB = RB; W.RC = RC; }
        { RC <- RB << INT; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

asrc INT, RB, RC
        { W.MODE = 0x0; W.OP = 0xB; W.RA = INT;
          W.RB = RB; W.RC = RC; }
        { RC <- RB >> INT; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

// data processing operations - 2-address
not RA,  RC
        { W.MODE = 0x0; W.OP = 0x1E; W.RA = RA;
          W.RB = 0x0; W.RC = RC; }
        { RC <- NOT(RA, 24); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

neg RA,  RC
        { W.MODE = 0x0; W.OP = 0x1E; W.RA = RA;
          W.RB = 0x1; W.RC = RC; }
        { RC <- SUBm(0, RA); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

// data processing operations - 1-address
clr RC
        { W.MODE = 0x0; W.OP = 0x1F; W.RA = 0x0;
          W.RB = 0x0; W.RC = RC; }
        { RC <- 0; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
```

```
        { Latency = 1; Usage = 1; }

// relational operations - 2-address
bnt RA,  RC
        { W.MODE = 0x0; W.OP = 0x1E; W.RA = RA;
          W.RB = 0x2; W.RC = RC; }
        { RC <- NOT(RA, 24); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

// relational operations - 3-address
equ RA, RB, RC
        { W.MODE = 0x0; W.OP = 0x0C; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { if (RA == RB) { RC <- 1; } else { RC <- 0; }; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

neq RA, RB, RC
        { W.MODE = 0x0; W.OP = 0x0D; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { if (RA != RB) { RC <- 1; } else { RC <- 0; }; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

gt RA, RB, RC
        { W.MODE = 0x0; W.OP = 0x0E; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { if (RA > RB) { RC <- 1; } else { RC <- 0; }; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

lt RA, RB, RC
        { W.MODE = 0x0; W.OP = 0x0F; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { if (RA < RB) { RC <- 1; } else { RC <- 0; }; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

gte RA, RB, RC
```

```
        { W.MODE = 0x0; W.OP = 0x10; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { if (RA >= RB) { RC <- 1; } else { RC <- 0; }; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

lte RA, RB, RC
        { W.MODE = 0x0; W.OP = 0x11; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { if (RA <= RB) { RC <- 1; } else { RC <- 0; }; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

// Load-store instructions - 3-address
ld RA, RB, RC
        { W.MODE = 0x0; W.OP = 0x12; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { RC <- IM[ADDm(RA, RB)]; }
        {}
        { Cycle = 2; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

st RA, RB, RC
        { W.MODE = 0x0; W.OP = 0x13; W.RA = RA;
          W.RB = RB; W.RC = RC; }
        { IM[ADDm(RA, RB)] <- RC; }
        {}
        { Cycle = 2; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

// Load-store instructions - 2-address and constant
ldc INT, RB, RC
        { W.MODE = 0x0; W.OP = 0x14; W.RA = INT; W.RB = RB;
          W.RC = RC; }
        { RC <- IM[ADDm(INT, RB)]; }
        {}
        { Cycle = 2; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

stc INT, RB, RC
        { W.MODE = 0x0; W.OP = 0x15; W.RA = INT; W.RB = RB;
          W.RC = RC; }
        { IM[ADDm(INT, RB)] <- RC; }
```

```
        {}
        { Cycle = 2; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

// move operations - 1-address and constant
mov RC, INT
        { W.MODE = 0x0; W.OP = 0x16;
          Split.Const = INT; W.RC = RC; }
        { RC <- INT; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

movm RC, INT
        { W.MODE = 0x0; W.OP = 0x17;
          Split.Const = INT; W.RC = RC; }
        { RC <- (INT << 10); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

movh RC, INT
        { W.MODE = 0x0; W.OP = 0x18;
          Split.Const = INT; W.RC = RC; }
        { RC <- (INT << 14); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

// Control operations
jmp ADDR
        { W.MODE = 0x9; Split.ADDRs = ADDR; }
        { PC <- ADDR; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

call ADDR
        { W.MODE = 0xA; Split.ADDRs = ADDR; }
        { RF[31] <- PC + 1; PC <- ADDR; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

ret
```

```
        { W.MODE = 0x7; W.OP = 0; W.RA = 0;
          W.RB = 0; W.RC = 0; }
        { PC <- RF[31]; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

br OFFS
        { W.MODE = 0xB; Split.ADDRs = OFFS; }
        { PC <- ADDm(PC, SEXT(OFFS, 20, 24)); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

brz RC, OFFS
        { W.MODE = 0xC; Split.OFFSs = OFFS; W.RC = RC; }
        { if (RC == 0)
            { PC <- ADDm(PC, SEXT(OFFS, 15, 24)); }; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

brnz RC, OFFS
        { W.MODE = 0xD; Split.OFFSs = OFFS; W.RC = RC; }
        { if (RC != 0)
            { PC <- ADDm(PC, SEXT(OFFS, 15, 24)); }; }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

nop
        { W.MODE = 0x0; W.OP = 0; W.RA = 0;
          W.RB = 0; W.RC = 0; }
        { NOP(); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

halt
        { W.MODE = 0x0; W.OP = 0; W.RA = 0;
          W.RB = 0; W.RC = 1; }
        { HALT(); }
        {}
        { Cycle = 1; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }
```

```
// ------------------------------------------------------------------

Section Constraints
// No constraints - RISCs are VERY simple

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

Section Optional
```

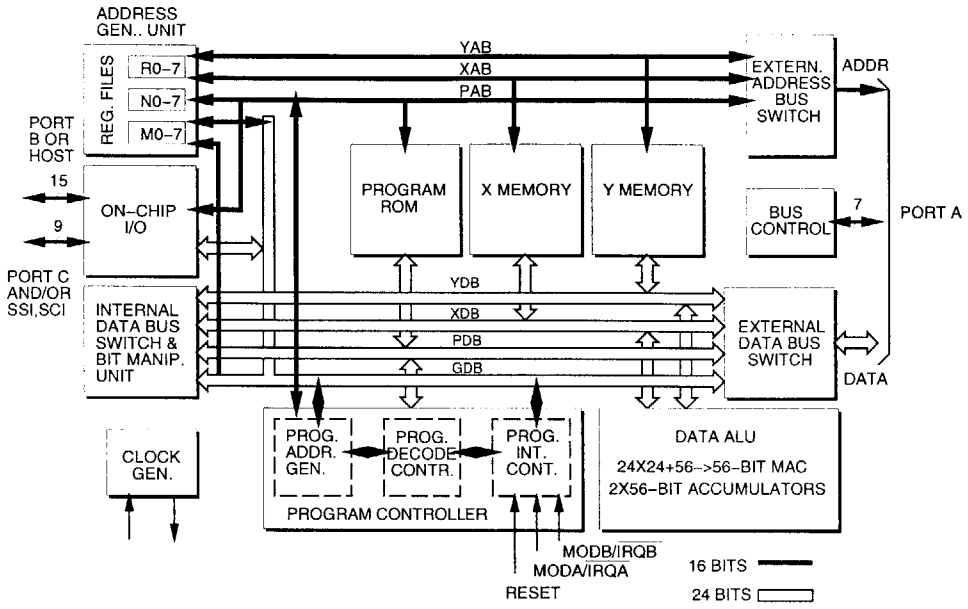

Figure C-4: **The Motorola 56000 DSP engine.**

C.4 The Motorola 56000 DSP

```
// Conventions:

// | = OR
// & = AND
// [x..y] = range from x to y
// , = used between required fields
// @ = variable declaration follows

// \ = ignore special symbol


// ---------------------------------------------------------------
// Number of fields in each VLIW word
// Size and format of each

Section Format

DB      = OP[8], MODE[8];
Main    = OP[8];


// ---------------------------------------------------------------
Section Global_Definitions
```

```
// Some Macro definitions
//
#define RENAME(x,y) Non_Terminal y: x { $$ = x; } { x } { x } { } { }

#define TRUE            1
#define FALSE           0
#define DEFINE_NULL_OP  {} { NULLOP(); } {} {} {}

#define ADDm(x, y)       ADD(x,y,56,"sat")
#define ADDCm(x, y)      ADDC(x,y, 0, CCRS[0], 56,"sat")
#define SUBm(x, y)       SUB(x,y,56,"sat")
#define SUBCm(x, y)      SUBC(x,y, 0, CCRS[0], 56,"sat")
#define MULm(x,y)        MUL(x,y,56,24,"trn")
#define DIVm(x,y)        DIV(x,y,56,24,24,"trn")
#define ANDm(x,y)        AND(x,y,24)
#define XORm(x,y)        XOR(x,y,24)
#define ORm(x,y)         OR(x,y,24)
#define SORm(x,y)        OR(x,y,8)
#define NOTm(x)          NOT(x,24)
#define ANDSm(x,y)       AND(x,y,8)
#define ASLm(x)          ASL(x,1,0,CCRS[0],56)
#define ASRm(x)          ASR(x,1,x[55],CCRS[0],56)
#define LSLm(x)          ASL(x,1,0,CCRS[0],24)
#define LSRm(x)          ASR(x,1,0,CCRS[0],24)
#define ROLm(x)          ASL(x,1,CCRS[0],CCRS[0],24)
#define RORm(x)          ASR(x,1,CCRS[0],CCRS[0],24)
#define ABSm(x)          ABS(x,56)
#define RNDm(x)          RND(x, 32, "near")
// Shorthand names for flags
#define C_F          CCRS[0]
#define V_F          CCRS[1]
#define Z_F          CCRS[2]
#define N_F          CCRS[3]
#define U_F          CCRS[4]
#define E_F          CCRS[5]
#define L_F          CCRS[6]
#define LF_F         MRS[8]
#define T_F          MRS[6]
#define SET_FLAG(c,f)    if (c) { f <- 1;} else {f <- 0;}
#define CSET_FLAG(c,f)   if (c) { f <- 1;}
#define SET_ALL(v,z,n,u,e)                                  \
                        SET_FLAG(v,V_F);                    \
                        SET_FLAG(z,Z_F);                    \
```

```
                        SET_FLAG(n,N_F);                          \
                        SET_FLAG(u,U_F);                          \
                        SET_FLAG(e,E_F);
#define CSET_ALL(v,z,n,u,e)                                        \
                        CSET_FLAG(v,V_F);                          \
                        CSET_FLAG(z,Z_F);                          \
                        CSET_FLAG(n,N_F);                          \
                        CSET_FLAG(u,U_F);                          \
                        CSET_FLAG(e,E_F);
#define CLEAR_ALL(v,z,n,u,e)                                       \
    V_F <- v; Z_F <- z; N_F <- n; U_F <- u; E_F <- e;

#define SIGNED(acc)     (((S_MODE == 0) &                          \
           (((acc >> 47) == 0x0) | ((acc >> 47) == 0x1FF))) |      \
          ((S_MODE == 1) &                                         \
           (((acc >> 48) == 0x0) | ((acc >> 48) == 0xFF))) |       \
          ((S_MODE == 2) &                                         \
           (((acc >> 46) == 0x0) | ((acc >> 46) == 0x3FF))))

#define UNORM(acc)      (((S_MODE == 0) & !(acc[47] ^ acc[46])) | \
                         ((S_MODE == 1) & !(acc[48] ^ acc[47])) | \
                         ((S_MODE == 1) & !(acc[46] ^ acc[45])))

// This ends up in the lex file.
//       assembly        token  type     value
Token   X0              X0_R        {};
Token   X1              X1_R        {};
Token   Y0              Y0_R        {};
Token   Y1              Y1_R        {};
Token   A0              A0_R        {};
Token   A1              A1_R        {};
Token   A2              A2_R        {};
Token   B0              B0_R        {};
Token   B1              B1_R        {};
Token   B2              B2_R        {};
Token   R[0..7]         R_R         { [0..7]; };
Token   N[0..7]         N_R         { [0..7]; };
Token   M[0..7]         M_R         { [0..7]; };
Token   "A10"           A10_D       {};
Token   "B10"           B10_D       {};
Token   XREG            X_D         {};
Token   YREG            Y_D         {};
Token   AREG            A_D         {};
Token   BREG            B_D         {};
Token   LA              LA          {};
```

```
Token   LC              LC          {};
Token   SSH             SSH         {};
Token   SSL             SSL         {};
Token   SP              SP          {};
Token   PC              PC          {};
Token   MR              MR          {};
Token   CCR             CCR         {};
Token   SR              SR          {};
Token   OMR             OMR         {};
Token   SS              SS          {};
Token   AB              AB_D        {};
Token   BA              BA_D        {};
Token   "X:"            XMEM        {};
Token   "Y:"            YMEM        {};
Token   "L:"            LMEM        {};
Token   "P:"            PMEM        {};
Token   "#"             HASH        {};
Token   sia             SIA         {};
Token   lia             LIA         {};
Token   sid             SID         {};
Token   lid             LID         {};
Token   CCC             CCC         { 0x0; };
Token   CGE             CGE         { 0x1; };
Token   CNE             CNE         { 0x2; };
Token   CPL             CPL         { 0x3; };
Token   CNN             CNN         { 0x4; };
Token   CEC             CEC         { 0x5; };
Token   CLC             CLC         { 0x6; };
Token   CGT             CGT         { 0x7; };
Token   CCS             CCS         { 0x8; };
Token   CLT             CLT         { 0x9; };
Token   CEQ             CEQ         { 0xA; };
Token   CMI             CMI         { 0xB; };
Token   CNR             CNR         { 0xC; };
Token   CES             CES         { 0xD; };
Token   CLS             CLS         { 0xE; };
Token   CLE             CLE         { 0xF; };
Token   "-"             SIGN        { 0x1; };

Non_Terminal      C_CODE:
                CCC {$$ = CCC; } { CCRS[0] == 0 } {} {} {} |
                CGE {$$ = CGE; } { (CCRS[3] ^ CCRS[1]) == 0 }
                                  {} {} {} |
                CNE {$$ = CNE; } { CCRS[2] == 0 } {} {} {} |
                CPL {$$ = CPL; } { CCRS[3] == 1 } {} {} {} |
```

```
                CNN {$$ = CNN; } { (CCRS[2] |
                                   ((~CCRS[4])&(~CCRS[5])))
                                   == 0 } {} {} {} |
                CEC {$$ = CEC; } { CCRS[5] == 0 } {} {} {} |
                CLC {$$ = CLC; } { CCRS[6] == 0 } {} {} {} |
                CGT {$$ = CGT; } { (CCRS[2] | (CCRS[3] ^ CCRS[1]))
                                   == 0 } {} {} {} |
                CCS {$$ = CCS; } { CCRS[0] == 1 } {} {} {} |
                CLT {$$ = CLT; } { (CCRS[3] ^ CCRS[1]) == 1 }
                                  {} {} {} |
                CEQ {$$ = CEQ; } { CCRS[2] == 1 } {} {} {} |
                CMI {$$ = CMI; } { CCRS[3] == 1 } {} {} {} |
                CNR {$$ = CNR; } { (CCRS[2] |
                                   ((~CCRS[4])&(~CCRS[5])))
                                   == 1 } {} {} {} |
                CES {$$ = CES; } { CCRS[5] == 1 } {} {} {} |
                CLS {$$ = CLS; } { CCRS[6] == 0 } {} {} {} |
                CLE {$$ = CLE; } { (CCRS[2] | (CCRS[3] ^ CCRS[1]))
                                   == 0 } {} {} {} ;

Non_Terminal      ADDRESS: INT {$$ = INT; } {INT} {INT} {} {} |
                           NAME {$$ = NAME;} {NAME} {NAME} {} {} ;

Non_Terminal      ACC: A_D {$$ = 0; } {AREG} {AREG} {} {} |
                       B_D {$$ = 1; } {BREG} {BREG} {} {} ;

Non_Terminal      ACCS: A_D {$$ = 0; } {A10} {A10} {} {} |
                        B_D {$$ = 1; } {B10} {B10} {} {} ;

Non_Terminal      ACC_1: A_D {$$ = 0; } {A[1]} {A[1]} {} {} |
                         B_D {$$ = 1; } {B[1]} {B[1]} {} {} ;

Non_Terminal      ACC_0: A_D {$$ = 0; } {A[0]} {A[0]} {} {} |
                         B_D {$$ = 1; } {B[0]} {B[0]} {} {} ;

Non_Terminal      X_R:
                    X0_R {$$ = 0;} {X[0]} {X[0]} {} {} |
                    X1_R {$$ = 1;} {X[1]} {X[1]} {} {} ;

Non_Terminal      Y_R:
                    Y0_R {$$ = 0;} {Y[0]} {Y[0]} {} {} |
                    Y1_R {$$ = 1;} {Y[1]} {Y[1]} {} {} ;

Non_Terminal      d:
                    A_D {$$ = 0; } {AREG} {AREG} {} {} |
```

```
                B_D {$$ = 1; } {BREG} {BREG} {} {} ;

Non_Terminal    e:
                X0_R {$$ = 0;} {X[0]} {X[0]} {} {} |
                X1_R {$$ = 1;} {X[1]} {X[1]} {} {} ;

Non_Terminal    f:
                Y0_R {$$ = 0;} {Y[0]} {Y[0]} {} {} |
                Y1_R {$$ = 1;} {Y[1]} {Y[1]} {} {} ;

Non_Terminal    j:
                X_D {$$ = 0;} {XREG} {XREG} {} {} |
                Y_D {$$ = 1;} {YREG} {YREG} {} {} ;

Non_Terminal    DD:
                X_R  {$$ = X_R;} {X_R} {X_R} {} {} |
                Y_R  {$$ = 0x2 | Y_R;} {Y_R} {Y_R} {} {} ;

Non_Terminal    ee:
                X_R  {$$ = X_R;} {X_R} {X_R} {} {} |
                ACC  {$$ = 0x2 | ACC;} {ACC} {ACC} {} {} ;

Non_Terminal    EE:
                MR  {$$ = 0;} {MRS}  {MRS}  {} {} |
                CCR {$$ = 1;} {CCRS} {CCRS} {} {} |
                OMR {$$ = 2;} {OMRS} {OMRS} {} {} ;

Non_Terminal    ff:
                Y_R  {$$ = Y_R;} {Y_R} {Y_R} {} {} |
                ACC  {$$ = 0x2 | ACC;} {ACC} {ACC} {} {} ;

Non_Terminal    JJ:
                X0_R {$$ = 0;} {X[0]} {X[0]} {} {} |
                Y0_R {$$ = 1;} {Y[0]} {Y[0]} {} {} |
                X1_R {$$ = 2;} {X[1]} {X[1]} {} {} |
                Y1_R {$$ = 3;} {Y[1]} {Y[1]} {} {} ;

Non_Terminal    DDD:
                A0_R {$$ = 0x0;} {A[0]} {A[0]} {} {} |
                B0_R {$$ = 0x1;} {B[0]} {B[0]} {} {} |
                A2_R {$$ = 0x2;} {A[2]} {A[2]} {} {} |
                B2_R {$$ = 0x3;} {B[2]} {B[2]} {} {} |
                A1_R {$$ = 0x4;} {A[1]} {A[1]} {} {} |
                B1_R {$$ = 0x5;} {B[1]} {B[1]} {} {} |
                A_D  {$$ = 0x6;} {AREG} {AREG} {} {} |
```

```
                B_D  {$$ = 0x7;} {BREG} {BREG} {} {} ;

Non_Terminal    LLL:
                A10_D {$$ = 0x0;} {A10}   {A10}   {} {} |
                B10_D {$$ = 0x1;} {B10}   {B10}   {} {} |
                X_D   {$$ = 0x2;} {XREG}  {XREG}  {} {} |
                Y_D   {$$ = 0x3;} {YREG}  {YREG}  {} {} |
                A_D   {$$ = 0x4;} {AREG}  {AREG}  {} {} |
                B_D   {$$ = 0x5;} {BREG}  {BREG}  {} {} |
                AB_D  {$$ = 0x6;} {ABREG} {ABREG} {} {} |
                BA_D  {$$ = 0x7;} {BAREG} {BAREG} {} {} ;

Non_Terminal    TTT:
                R_R {$$ = R_R;} {AGU_R[R_R]} {AGU_R[R_R]} {} {} ;

Non_Terminal    ttt:
                R_R {$$ = R_R;} {AGU_R[R_R]} {AGU_R[R_R]} {} {} ;

Non_Terminal    QQQ:
          X0_R ',' X0__R {$$ = 0;} {MULm(X[0], X[0])} {} {} {} |
          Y0_R ',' Y0__R {$$ = 1;} {MULm(Y[0], Y[0])} {} {} {} |
          X1_R ',' X0_R  {$$ = 2;} {MULm(X[1], X[0])} {} {} {} |
          Y1_R ',' Y0_R  {$$ = 3;} {MULm(Y[1], Y[0])} {} {} {} |
          X0_R ',' Y1_R  {$$ = 4;} {MULm(X[0], Y[1])} {} {} {} |
          Y0_R ',' X0_R  {$$ = 5;} {MULm(Y[0], X[0])} {} {} {} |
          X1_R ',' Y0_R  {$$ = 6;} {MULm(X[1], Y[0])} {} {} {} |
          Y1_R ',' X1_R  {$$ = 7;} {MULm(Y[1], X[1])} {} {} {} ;

Non_Terminal    JJJ:
                X_D  {$$ = 2;} {SEXT(XREG, 48, 56)}
                               {SEXT(XREG, 48, 56)} {} {}       |
                Y_D  {$$ = 3;} {SEXT(YREG, 48, 56)}
                               {SEXT(YREG, 48, 56)} {} {}       |
                X0_R {$$ = 4;} {SEXT(X[0], 24, 56) << 24}
                               {SEXT(X[0], 24, 56) << 24} {} {} |
                Y0_R {$$ = 5;} {SEXT(Y[0], 24, 56) << 24}
                               {SEXT(Y[0], 24, 56) << 24} {} {} |
                X1_R {$$ = 6;} {SEXT(X[1], 24, 56) << 24}
                               {SEXT(X[1], 24, 56) << 24} {} {} |
                Y1_R {$$ = 7;} {SEXT(Y[1], 24, 56) << 24}
                               {SEXT(Y[1], 24, 56) << 24} {} {} ;

Non_Terminal    jjj:
                X0_R {$$ = 4;} {X[0]} {X[0]} {} {} |
                Y0_R {$$ = 5;} {Y[0]} {Y[0]} {} {} |
```

```
                X1_R {$$ = 6;} {X[1]} {X[1]} {} {} |
                Y1_R {$$ = 7;} {Y[1]} {Y[1]} {} {} ;

Non_Terminal    GGG:
                SR  {$$ = 0x1;} {SRS}  {SRS}  {} {} |
                OMR {$$ = 0x2;} {OMRS} {OMRS} {} {} |
                SP  {$$ = 0x3;} {SPS}  {SPS}  {} {} |
                SSH {$$ = 0x4;} {SSHS} {SSHS} {} {} |
                SSL {$$ = 0x5;} {SSLS} {SSLS} {} {} |
                LA  {$$ = 0x6;} {LAS}  {LAS}  {} {} |
                LC  {$$ = 0x7;} {LCS}  {LCS}  {} {} ;

Non_Terminal    DDDD:
                DD  {$$ = 0x4 | DD;}  {DD}  {DD}  {} {} |
                DDD {$$ = 0x8 | DDD;} {DDD} {DDD} {} {} ;

Non_Terminal    ccccc:
          M_R {$$ = M_R;}  {AGU_M[M_R]} {AGU_M[M_R]} {} {} |
          SR  {$$ = 0x19;} {SRS}        {SRS}        {} {} |
          OMR {$$ = 0x1A;} {OMRS}       {OMRS}       {} {} |
          SP  {$$ = 0x1B;} {SPS}        {SPS}        {} {} |
          SSH {$$ = 0x1C;} {SSHS}       {SSHS}       {} {} |
          SSL {$$ = 0x1D;} {SSLS}       {SSLS}       {} {} |
          LA  {$$ = 0x1E;} {LAS}        {LAS}        {} {} |
          LC  {$$ = 0x1F;} {LCS}        {LCS}        {} {} ;

Non_Terminal    ddddd:
         DD  {$$ = 0x04 | DD;}  {DD}         {DD}         {} {} |
         DDD {$$ = 0x08 | DDD;} {DDD}        {DDD}        {} {} |
         R_R {$$ = 0x10 | R_R;} {AGU_R[R_R]} {AGU_R[R_R]} {} {} |
         N_R {$$ = 0x18 | N_R;} {AGU_N[N_R]} {AGU_N[N_R]} {} {} ;

Non_Terminal    eeeee:
         DD  {$$ = 0x04 | DD;}  {DD}         {DD}         {} {} |
         DDD {$$ = 0x08 | DDD;} {DDD}        {DDD}        {} {} |
         R_R {$$ = 0x10 | R_R;} {AGU_R[R_R]} {AGU_R[R_R]} {} {} |
         N_R {$$ = 0x18 | N_R;} {AGU_N[N_R]} {AGU_N[N_R]} {} {} ;

Non_Terminal    dddddd:
          DD  {$$ = 0x04 | DD;}  {DD}         {DD}         {} {} |
          DDD {$$ = 0x08 | DDD;} {DDD}        {DDD}        {} {} |
          R_R {$$ = 0x10 | R_R;} {AGU_R[R_R]} {AGU_R[R_R]} {} {} |
          N_R {$$ = 0x18 | N_R;} {AGU_N[N_R]} {AGU_N[N_R]} {} {} |
          M_R {$$ = 0x20 | M_R;} {AGU_M[M_R]} {AGU_M[M_R]} {} {} |
          GGG {$$ = 0x38 | GGG;} {GGG}        {GGG}        {} {} ;
```

```
Non_Terminal      EA:
        '(' R_R ')' '-' N_R     {$$ = 0x00 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] - AGU_N[N_R]; } {} {} |
        '(' R_R ')' '+' N_R     {$$ = 0x08 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] + AGU_N[N_R]; } {} {} |
        '(' R_R ')' '-'         {$$ = 0x10 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] - 1; } {} {} |
        '(' R_R ')' '+'         {$$ = 0x18 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] + 1; } {} {} |
        '(' R_R ')'             {$$ = 0x20 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] } {} {} |
        '(' R_R '+' N_R ')'     {$$ = 0x28 | (R_R & 0x7); }
                { AGU_R[R_R] + AGU_N[N_R] }
                { AGU_R[R_R] + AGU_N[N_R] }
                { Cycle=2; Size=0; } {} |
        '-' '(' R_R ')'         {$$ = 0x38 | (R_R & 0x7); }
                { AGU_R[R_R] - 1 }
                { AGU_R[R_R] <- AGU_R[R_R] - 1; }
                { Cycle=2; Size=0;} {} |
        ADDRESS {$$ = 0x30; Additional(0, Split.ADDR=ADDRESS;); }
                {ADDRESS} {ADDRESS} { Cycle = 2; Size = 1; } {} ;

Non_Terminal      ADRM:
        '(' R_R ')' '-' N_R     {$$ = 0x00 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] - AGU_N[N_R]; } {} {} |
        '(' R_R ')' '+' N_R     {$$ = 0x08 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] + AGU_N[N_R]; } {} {} |
        '(' R_R ')' '-'         {$$ = 0x10 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] - 1; } {} {} |
        '(' R_R ')' '+'         {$$ = 0x18 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] + 1; } {} {} |
        '(' R_R ')'             {$$ = 0x20 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] } {} {} |
        '(' R_R '+' N_R ')'     {$$ = 0x28 | (R_R & 0x7); }
```

```
                { AGU_R[R_R] + AGU_N[N_R] }
                { AGU_R[R_R] + AGU_N[N_R] }
                { Cycle=2; Size=0; } {} |
        '-' '(' R_R ')'         {$$ = 0x38 | (R_R & 0x7); }
                { AGU_R[R_R] - 1 }
                { AGU_R[R_R] <- AGU_R[R_R] - 1; }
                { Cycle=2; Size=0;} {} ;

Non_Terminal      SEA:
        '(' R_R ')' '-' N_R     {$$ = 0x00 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] - AGU_N[N_R]; } {} {} |
        '(' R_R ')' '+' N_R     {$$ = 0x08 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] + AGU_N[N_R]; } {} {} |
        '(' R_R ')' '-'         {$$ = 0x10 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] - 1; } {} {} |
        '(' R_R ')' '+'         {$$ = 0x18 | (R_R & 0x7); }
                { AGU_R[R_R] }
                { AGU_R[R_R] <- AGU_R[R_R] + 1; } {} {} ;

Non_Terminal      SEAN:
        '(' R_R ')' '-' N_R     {$$ = 0x00 | (R_R & 0x7); }
                { AGU_R[R_R] - AGU_N[N_R] } {} {} {}            |
        '(' R_R ')' '+' N_R     {$$ = 0x08 | (R_R & 0x7); }
                { AGU_R[R_R] + AGU_N[N_R] } {} {} {}            |
        '(' R_R ')' '-'         {$$ = 0x10 | (R_R & 0x7); }
                { AGU_R[R_R] - 1 } {} {} {}                     |
        '(' R_R ')' '+'         {$$ = 0x18 | (R_R & 0x7); }
                { AGU_R[R_R] + 1 } {} {} {}                     ;

Non_Terminal      BIT5:
            HASH INT {$$ = 0x0000001F & INT;} {INT} {INT} {} {} ;

Non_Terminal      BIT8:
            HASH INT {$$ = 0x000000FF & INT;} {INT} {INT} {} {} ;

Non_Terminal      BIT12:
            HASH INT {$$ = 0x00000FFF & INT;} {INT} {INT} {} {} ;

Non_Terminal      BIT16:
            HASH INT {$$ = 0x0000FFFF & INT;} {INT} {INT} {} {} ;

Non_Terminal      BIT24:
```

```
            HASH INT {$$ = 0x00FFFFFF & INT;} {INT} {INT} {} {} ;

Non_Terminal      ADD6:
            INT {$$ = 0x0000003F & INT;} {INT} {INT} {} {} ;

Non_Terminal      PRT6:
            INT {$$ = 0x0000003F & INT;} {INT} {INT} {} {} ;

Non_Terminal      ADD12:
            INT {$$ = 0x00000FFF & INT;} {INT} {INT} {} {} ;

Non_Terminal      ADD16:
            INT {Additional(0, Split.ADDR = 0x0000FFFF & INT;);}
                        {INT} {INT} {} {} ;

Non_Terminal      XPP:
            XMEM PRT6 {$$ = PRT6;} {XPort[PRT6]} {XPort[PRT6]} {} {} ;

Non_Terminal      YPP:
            YMEM PRT6 {$$ = PRT6;} {YPort[PRT6]} {YPort[PRT6]} {} {} ;

Non_Terminal      XAA:
            XMEM ADD6 {$$ = ADD6;} {XMEMS[ADD6]} {XMEMS[ADD6]} {} {} ;

Non_Terminal      YAA:
            YMEM ADD6 {$$ = ADD6;} {YMEMS[ADD6]} {YMEMS[ADD6]} {} {} ;

Non_Terminal      XEA:
              XMEM EA {$$ = EA;} {XMEMS[EA]} {EA}
                      {Cycle = EA; Size = EA;} {} ;

Non_Terminal      YEA:
              YMEM EA {$$ = EA;} {YMEMS[EA]} {EA}
                      {Cycle = EA; Size = EA;} {} ;

Non_Terminal      XSEA:
              XMEM SEA {$$ = SEA;} {XMEMS[SEA]} {SEA} {} {} ;

Non_Terminal      YSEA:
              YMEM SEA {$$ = SEA;} {YMEMS[SEA]} {SEA} {} {} ;

Non_Terminal      PEA:
              PMEM EA {$$ = EA;} {IM[EA]} {EA}
                      {Cycle = EA; Size = EA;} {} ;
```

```
Non_Terminal      PAA:
            PMEM ADD6 {$$ = ADD6;} {IM[ADD6]} {IM[ADD6]} {} {} ;

Non_Terminal      LEA:
                    LMEM EA {$$ = EA;} {LMEMS[EA]} {EA}
                            {Cycle = EA; Size = EA;} {} ;

Non_Terminal      LAA:
            LMEM ADD6 {$$ = ADD6;} {LMEMS[ADD6]} {LMEMS[ADD6]} {} {} ;

Non_Terminal      BIT_MODE:
            BIT5 ',' XPP    {$$ = 0x8000 | (XPP << 8) | BIT5;}
                             {XPP[BIT5]} {XPP[BIT5]} {} {}        |
            BIT5 ',' YPP    {$$ = 0x8040 | (YPP << 8) | BIT5;}
                             {YPP[BIT5]} {YPP[BIT5]} {} {}        |
            BIT5 ',' XEA
                             {$$ = 0x4000 | (XEA << 8) | BIT5;}
                             {XEA[BIT5]} {XEA}
                             { Cycle = XEA; Size = XEA; } {}      |
            BIT5 ',' YEA
                             {$$ = 0x4040 | (YEA << 8) | BIT5;}
                             {YEA[BIT5]} {YEA}
                             { Cycle = YEA; Size = YEA; } {}      |
            BIT5 ',' XAA    {$$ = 0x0000 | (XAA << 8) | BIT5;}
                             {XAA[BIT5]} {XAA[BIT5]} {} {}        |
            BIT5 ',' YAA    {$$ = 0x0040 | (YAA << 8) | BIT5;}
                             {YAA[BIT5]} {YAA[BIT5]} {} {}        ;

Non_Terminal      BIT_MODE2:
            BIT5 ',' XPP    {$$ = 0x8000 | (XPP << 8) | BIT5;}
                             {XPP[BIT5]} {XPP[BIT5]} {} {}        |
            BIT5 ',' YPP    {$$ = 0x8040 | (YPP << 8) | BIT5;}
                             {YPP[BIT5]} {YPP[BIT5]} {} {}        |
            BIT5 ',' XMEM ADRM
                             {$$ = 0x4000 | (ADRM << 8) | BIT5;}
                             {XMEMS[ADRM][BIT5]} {ADRM} {} {}     |
            BIT5 ',' YMEM ADRM
                             {$$ = 0x4040 | (ADRM << 8) | BIT5;}
                             {YMEMS[ADRM][BIT5]} {ADRM} {} {}     |
            BIT5 ',' XAA    {$$ = 0x0000 | (XAA << 8) | BIT5;}
                             {XAA[BIT5]} {XAA[BIT5]} {} {}        |
            BIT5 ',' YAA    {$$ = 0x0040 | (YAA << 8) | BIT5;}
                             {YAA[BIT5]} {YAA[BIT5]} {} {}        ;

Non_Terminal      MEM_MODE:
```

```
            XMEM ADRM    {$$ = 0x4000 | (ADRM << 8);}
                             {XMEMS[ADRM]} {XMEMS[ADRM]} {} {}  |
            YMEM ADRM    {$$ = 0x4040 | (ADRM << 8);}
                             {YMEMS[ADRM]} {YMEMS[ADRM]} {} {}  |
            XAA     {$$ = 0x0000 | (XAA << 8);}
                             {XAA} {XAA} {} {}                  |
            YAA     {$$ = 0x0040 | (YAA << 8);}
                             {YAA} {YAA} {} {}                  ;

RENAME(X0_R, X0__R);
RENAME(Y0_R, Y0__R);

Split.ADDR      DB.OP+DB.MODE+Main.OP;
Split.DATA      DB.OP+DB.MODE+Main.OP;

// ---------------------------------------------------------------
// The number of registers for each unit and the topology of
// any of the register files.

Section Storage

//                          = depth , width
Instruction Memory IM       = 0x1000 , 0x18
Memory  XMEMS               = 0x1000 , 0x18
Memory  YMEMS               = 0x1000 , 0x18
RegFile AGU_R               = 0x8 , 0x18
RegFile AGU_N               = 0x8 , 0x18
RegFile AGU_M               = 0x8 , 0x18
RegFile X                   = 0x2 , 0x18
RegFile Y                   = 0x2 , 0x18
RegFile A                   = 0x3 , 0x18
RegFile B                   = 0x3 , 0x18
MMIO XPort                  = 0x40 , 0x18
MMIO YPort                  = 0x40 , 0x18
CRegister LAS               = 0x10          // Loop Address
CRegister LCS               = 0x10          // Loop Counter
CRegister MRS               = 0x8
CRegister CCRS              = 0x8
CRegister OMRS              = 0x8
CRegister SPS               = 0x6
ProgramCounter PCS          = 0x10
Stack SSS(SPS)              = 0xf , 0x20

Alias AREG      A[2][0x7 - 0x0],A[1],A[0];          0x38
Alias BREG      B[2][0x7 - 0x0],B[1],B[0];          0x38
```

```
Alias XREG      X[1],X[0];                          0x30
Alias YREG      Y[1],Y[0];                          0x30
Alias SSHS      SSS[SPS][0x1f - 0x10];              0x10
Alias SSLS      SSS[SPS][0x0f - 0x00];              0x10
Alias SRS       MRS,CCRS;                           0x10
Alias A10       A[1],A[0];                          0x30
Alias B10       B[1],B[0];                          0x30
Alias LMEMS     XMEMS[0 - 0xFFF],YMEMS[0 - 0xFFF];  0x1000,0x30
Alias ABREG     A[1],B[1];                          0x30
Alias BAREG     B[1],A[1];                          0x30
Alias S_MODE    MRS[2 - 3];                         0x02

// ------------------------------------------------------------------
// Correspondence between assembly mneumonics, bitfields, and actual
// instructions.

// NOTE: The timing described here assumes there is no wait states
//       and all memory is internal.

Section Instruction_Set

Field DBM:
    MOVE_NULL                       DEFINE_NULL_OP

    MOVE XSEA, ee, YSEA, ff
                    { DB.OP = 0xC0 | ((YSEA >> 3) << 4)
                      | (ee << 2) | ff;
                      DB.MODE = 0x80 | ((0x3 & YSEA)<< 5) | XSEA; }
                { ff <- YSEA; ee <- XSEA; }
                { XSEA; YSEA; CSET_FLAG(LIMIT, L_F); }
                { Cycle = 0; Stall = 0; Size = 0; }
                { Latency = 2; Usage = 1; }

    MOVE XSEA, ee, ff, YSEA
                    { DB.OP = 0x80 | ((YSEA >> 3) << 4)
                      | (ee << 2) | ff;
                      DB.MODE = 0x80 | ((0x3 & YSEA)<< 5) | XSEA; }
                { YSEA <- ff; ee <- XSEA; }
                { XSEA; YSEA; CSET_FLAG(LIMIT, L_F); }
                { Cycle = 0; Stall = 0; Size = 0; }
                { Latency = 2; Usage = 1; }

    MOVE ee, XSEA, YSEA, ff
                    { DB.OP = 0xC0 | ((YSEA >> 3) << 4)
                      | (ee << 2) | ff;
```

```
                DB.MODE = 0x00 | ((0x3 & YSEA)<< 5) | XSEA; }
        { ff <- YSEA; XSEA <- ee; }
        { XSEA; YSEA; CSET_FLAG(LIMIT, L_F); }
        { Cycle = 0; Stall = 0; Size = 0; }
        { Latency = 2; Usage = 1; }

MOVE ee, XSEA, ff, YSEA
            { DB.OP = 0x80 | ((YSEA >> 3) << 4)
              | (ee << 2) | ff;
              DB.MODE = 0x00 | ((0x3 & YSEA)<< 5) | XSEA; }
        { YSEA <- ff; XSEA <- ee; }
        { XSEA; YSEA; CSET_FLAG(LIMIT, L_F); }
        { Cycle = 0; Stall = 0; Size = 0; }
        { Latency = 2; Usage = 1; }

MOVE XEA, ddddd
                          { DB.OP = 0x40 | ((ddddd >> 3) << 4)
                                  | (0x7 & ddddd);
                            DB.MODE = 0xC0 | XEA; }
        { ddddd <- XEA; }
        { XEA; CSET_FLAG(LIMIT, L_F); }
        { Cycle = XEA; Stall = 0; Size = XEA; }
        { Latency = 2; Usage = 1; }

MOVE ddddd, XEA
                          { DB.OP = 0x40 | ((ddddd >> 3) << 4)
                                  | (0x7 & ddddd);
                            DB.MODE = 0x40 | XEA; }
        { XEA <- ddddd; }
        { XEA; CSET_FLAG(LIMIT, L_F); }
        { Cycle = XEA; Stall = 0; Size = XEA; }
        { Latency = 2; Usage = 1; }

MOVE XAA, ddddd
                          { DB.OP = 0x40 | ((ddddd >> 3) << 4)
                                  | (0x7 & ddddd);
                            DB.MODE = 0x80 | XAA; }
        { ddddd <- XAA; }
        { CSET_FLAG(LIMIT, L_F); }
        { Cycle = 0; Stall = 0; Size = 0; }
        { Latency = 2; Usage = 1; }

MOVE ddddd, XAA
                          { DB.OP = 0x40 | ((ddddd >> 3) << 4)
                                  | (0x7 & ddddd);
```

```
                              DB.MODE = 0x00 | XAA; }
            { XAA <- ddddd; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 0; Stall = 0; Size = 1; }
            { Latency = 2; Usage = 1; }

MOVE YEA, ddddd
                            { DB.OP = 0x48 | ((ddddd >> 3) << 4)
                                    | (0x7 & ddddd);
                              DB.MODE = 0xC0 | YEA; }
            { ddddd <- YEA; }
            { YEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = YEA; Stall = 0; Size = YEA; }
            { Latency = 2; Usage = 1; }

MOVE ddddd, YEA
                            { DB.OP = 0x48 | ((ddddd >> 3) << 4)
                                    | (0x7 & ddddd);
                              DB.MODE = 0x40 | YEA; }
            { YEA <- ddddd; }
            { YEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = YEA; Stall = 0; Size = YEA; }
            { Latency = 2; Usage = 1; }

MOVE YAA, ddddd
                            { DB.OP = 0x48 | ((ddddd >> 3) << 4)
                                    | (0x7 & ddddd);
                              DB.MODE = 0x80 | YAA; }
            { ddddd <- YAA; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 0; Stall = 0; Size = 0; }
            { Latency = 2; Usage = 1; }

MOVE ddddd, YAA
                            { DB.OP = 0x48 | ((ddddd >> 3) << 4)
                                    | (0x7 & ddddd);
                              DB.MODE = 0x00 | YAA; }
            { YAA <- ddddd; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 0; Stall = 0; Size = 0; }
            { Latency = 2; Usage = 1; }

MOVE LEA, LLL
                            { DB.OP = 0x40 | ((LLL >> 2) << 4)
                                    | (0x3 & LLL);
```

```
                              DB.MODE = 0xC0 | LEA; }
            { LLL <- LEA; }
            { LEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = LEA; Stall = 0; Size = LEA; }
            { Latency = 1; Usage = 1; }

MOVE LLL, LEA
                            { DB.OP = 0x40 | ((LLL >> 2) << 4)
                                    | (0x3 & LLL);
                              DB.MODE = 0x40 | LEA; }
            { LEA <- LLL; }
            { LEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = LEA; Stall = 0; Size = LEA; }
            { Latency = 1; Usage = 1; }

MOVE LAA, LLL
                            { DB.OP = 0x40 | ((LLL >> 2) << 4)
                                    | (0x3 & LLL);
                              DB.MODE = 0x80 | LAA; }
            { LLL <- LAA; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 0; Stall = 0; Size = 0; }
            { Latency = 1; Usage = 1; }

MOVE LLL, LAA
                            { DB.OP = 0x40 | ((LLL >> 2) << 4)
                                    | (0x3 & LLL);
                              DB.MODE = 0x00 | LAA; }
            { LAA <- LLL; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 0; Stall = 0; Size = 0; }
            { Latency = 1; Usage = 1; }

MOVE BIT8, ddddd
                            { DB.OP = 0x20 | ddddd;
                              DB.MODE = BIT8; }
            { ddddd <- BIT8; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 0; Stall = 0; Size = 0; }
            { Latency = 1; Usage = 1; }

MOVE eeeee, ddddd
                 { DB.OP = 0x20 | (eeeee >> 3);
                   DB.MODE = ((0x7 & eeeee) << 5) | ddddd; }
            { ddddd <- eeeee; }
```

```
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 0; Stall = 0; Size = 0; }
            { Latency = 1; Usage = 1; }

MOVE SEA
                              { DB.OP = 0x20;
                                DB.MODE = 0x40 | SEA; }
            {}
            { SEA; }
            { Cycle = 0; Stall = 0; Size = 0; }
            { Latency = 1; Usage = 1; }

MOVE_NOP
                              { DB.OP = 0x20;
                                DB.MODE = 0x00; }
            { NOP(); }
            {}
            { Cycle = 0; Stall = 0; Size = 0; }
            { Latency = 1; Usage = 1; }

MOVE ACC, X_R, YEA, ff
                  { DB.OP = 0x10 | (X_R << 3) | (ACC << 2) | ff;
                    DB.MODE = 0xC0 | YEA; }
            { X_R <- ACC; ff <- YEA; }
            { YEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = YEA; Stall = 0; Size = YEA; }
            { Latency = 1; Usage = 1; }

MOVE ACC, X_R, ff, YEA
                  { DB.OP = 0x10 | (X_R << 3) | (ACC << 2) | ff;
                    DB.MODE = 0x40 | YEA; }
            { X_R <- ACC; YEA <- ff; }
            { YEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = YEA; Stall = 0; Size = YEA; }
            { Latency = 1; Usage = 1; }

MOVE ACC, X_R, BIT24, ff
                  { DB.OP = 0x10 | (X_R << 3) | (ACC << 2) | ff;
                    DB.MODE = 0xF4;
                    Additional(0, Split.DATA = BIT24;); }
            { X_R <- ACC; ff <- BIT24; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 0; Stall = 0; Size = 0; }
            { Latency = 1; Usage = 1; }
```

```
    MOVE Y0_R, ACC, YEA
                                { DB.OP = 0x08 | ACC;
                                  DB.MODE = 0x80 | YEA; }
                { ACC <- Y[0]; ACC <- YEA; }
                { YEA; CSET_FLAG(LIMIT, L_F); }
                { Cycle = YEA; Stall = 0; Size = YEA; }
                { Latency = 1; Usage = 1; }

    MOVE  XEA, ee, ACC, Y_R
                     { DB.OP = 0x10 | (ee << 2) | (ACC << 1) | Y_R;
                       DB.MODE = 0x80 | XEA; }
                { ee <- XEA; Y_R <- ACC; }
                { XEA; CSET_FLAG(LIMIT, L_F); }
                { Cycle = XEA; Stall = 0; Size = XEA; }
                { Latency = 1; Usage = 1; }

    MOVE  ee, XEA, ACC, Y_R
                     { DB.OP = 0x10 | (ee << 2) | (ACC << 1) | Y_R;
                       DB.MODE = 0x00 | XEA; }
                { XEA <- ee; Y_R <- ACC; }
                { XEA; CSET_FLAG(LIMIT, L_F); }
                { Cycle = XEA; Stall = 0; Size = XEA; }
                { Latency = 1; Usage = 1; }

    MOVE  BIT24, ee, ACC, Y_R
                     { DB.OP = 0x10 | (ee << 2) | (ACC << 1) | Y_R;
                       DB.MODE = 0xB4;
                       Additional(0, Split.DATA = BIT24;); }
                { ee <- BIT24; Y_R <- ACC; }
                { CSET_FLAG(LIMIT, L_F); }
                { Cycle = 0; Stall = 0; Size = 0; }
                { Latency = 1; Usage = 1; }

    MOVE  ACC, XEA, X0_R
                                { DB.OP = 0x08 | ACC;
                                  DB.MODE = 0x00 | XEA; }
                { XEA <- ACC; ACC <- X[0]; }
                { XEA; CSET_FLAG(LIMIT, L_F); }
                { Cycle = XEA; Stall = 0; Size = XEA; }
                { Latency = 1; Usage = 1; }

Field Main:
    Main_NULL                   DEFINE_NULL_OP

    JS C_CODE, ADD12
```

```
                { DB.OP = 0x0F;
                  DB.MODE = (C_CODE << 4) | (ADD12 >> 8);
                  Main.OP = ADD12 & 0xFF; }
              { if (C_CODE)
                { SPS <- SPS +1; SSHS <- PCS; SSLS <- SRS;
                          PCS <- ADD12; }; }
              {}
              { Cycle = 6; Stall = 0; Size = 1; }
              { Latency = 1; Usage = 1; }

J C_CODE, ADD12
                { DB.OP = 0x0E;
                  DB.MODE = (C_CODE << 4) | (ADD12 >> 8);
                  Main.OP = ADD12 & 0xFF; }
              { if (C_CODE) { PCS <- ADD12; }; }
              {}
              { Cycle = 6; Stall = 0; Size = 1; }
              { Latency = 1; Usage = 1; }

JSR ADD12
                { DB.OP = 0x0D;
                  DB.MODE = 0x0F & (ADD12 >> 8);
                  Main.OP = ADD12 & 0xFF; }
              { SPS <- SPS + 1; SSHS <- PCS;
                SSLS <- SRS; PCS <- ADD12; }
              {}
              { Cycle = 6; Stall = 0; Size = 1; }
              { Latency = 1; Usage = 1; }

JS C_CODE, EA
                              { DB.OP = 0x0B;
                                DB.MODE = 0xC0 | EA;
                                Main.OP = 0xA0 | C_CODE; }
              { if (C_CODE) { SPS <- SPS +1; SSHS <- PCS;
                             SSLS <- SRS; PCS <- EA; }; }
              { EA; }
              { Cycle = 6 + EA; Stall = 0; Size = 1 + EA; }
              { Latency = 1; Usage = 1; }

JSR EA
                              { DB.OP = 0x0B;
                                DB.MODE = 0xC0 | EA;
                                Main.OP = 0x80; }
              { SPS <- SPS + 1; SSHS <- PCS; SSLS <- SRS;
                PCS <- EA; }
```

```
            { EA; }
            { Cycle = 6 + EA; Stall = 0; Size = 1 + EA; }
            { Latency = 1; Usage = 1; }

J C_CODE, EA
                              { DB.OP = 0x0A;
                                DB.MODE = 0xC0 | EA;
                                Main.OP = 0xA0 | C_CODE; }
            { if (C_CODE) { PCS <- EA; }; }
            { EA; }
            { Cycle = 6 + EA; Stall = 0; Size = 1 + EA; }
            { Latency = 1; Usage = 1; }

JMP ADD12
                              { DB.OP = 0x0C;
                                DB.MODE = 0x00 | (ADD12 >> 8);
                                Main.OP = 0xFF & ADD12; }
            { PCS <- ADD12; }
            {}
            { Cycle = 6; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

JMP EA
                              { DB.OP = 0x0A;
                                DB.MODE = 0xC0 | EA;
                                Main.OP = 0x80; }
            { PCS <- EA; }
            { EA; }
            { Cycle = 6 + EA; Stall = 0; Size = 1 + EA; }
            { Latency = 1; Usage = 1; }

JSSET BIT_MODE2, ADD16
                  { DB.OP = 0x0B;
                    DB.MODE = BIT_MODE2 >> 8;
                    Main.OP = 0xA0 | (0xFF & BIT_MODE2);
                    Additional(0, Split.ADDR = ADD16;); }
            { if (BIT_MODE2) { SPS <- SPS + 1; SSHS <- PCS;
                               SSLS <- SRS; PCS <- ADD16; }; }
            { BIT_MODE2; }
            { Cycle = 6 + BIT_MODE2; Stall = 0; Size = 2; }
            { Latency = 1; Usage = 1; }

JSCLR BIT_MODE2, ADD16
                  { DB.OP = 0x0B;
                    DB.MODE = BIT_MODE2 >> 8;
```

```
                Main.OP = 0x80 | (0xFF & BIT_MODE2);
                Additional(0, Split.ADDR = ADD16;); }
            { if (~BIT_MODE2) { SPS <- SPS + 1; SSHS <- PCS;
                                SSLS <- SRS; PCS <- ADD16; }; }
            { BIT_MODE2; }
            { Cycle = 6 + BIT_MODE2; Stall = 0; Size = 2; }
            { Latency = 1; Usage = 1; }

JSET BIT_MODE2, ADD16
                { DB.OP = 0x0A;
                  DB.MODE = BIT_MODE2 >> 8;
                  Main.OP = 0xA0 | (0xFF & BIT_MODE2);
                  Additional(0, Split.ADDR = ADD16;); }
            { if (BIT_MODE2) { PCS <- ADD16; }; }
            { BIT_MODE2; }
            { Cycle = 6 + BIT_MODE2; Stall = 0; Size = 2; }
            { Latency = 1; Usage = 1; }

JCLR BIT_MODE2, ADD16
                { DB.OP = 0x0A;
                  DB.MODE = BIT_MODE2 >> 8;
                  Main.OP = 0x80 | (0xFF & BIT_MODE2);
                  Additional(0, Split.ADDR = ADD16;); }
            { if (~BIT_MODE2) { PCS <- ADD16; }; }
            { BIT_MODE2; }
            { Cycle = 6 + BIT_MODE2; Stall = 0; Size = 2; }
            { Latency = 1; Usage = 1; }

JSSET BIT5, dddddd, ADD16
                { DB.OP = 0x0B;
                  DB.MODE = 0xC0 | dddddd;
                  Main.OP = 0x20 | BIT5;
                  Additional(0, Split.ADDR = ADD16;); }
            { if (dddddd[BIT5]) { SPS <- SPS + 1; SSHS <- PCS;
                                  SSLS <- SRS; PCS <- ADD16; }; }
            {}
            { Cycle = 6; Stall = 0; Size = 2; }
            { Latency = 1; Usage = 1; }

JSCLR BIT5, dddddd, ADD16
                { DB.OP = 0x0B;
                  DB.MODE = 0xC0 | dddddd;
                  Main.OP = 0x00 | BIT5;
                  Additional(0, Split.ADDR = ADD16;); }
            { if (~dddddd[BIT5]) { SPS <- SPS + 1; SSHS <- PCS;
```

```
                                    SSLS <- SRS; PCS <- ADD16; }; }
            {}
            { Cycle = 6; Stall = 0; Size = 2; }
            { Latency = 1; Usage = 1; }

JSET BIT5, dddddd, ADD16
                 { DB.OP = 0x0A;
                   DB.MODE = 0xC0 | dddddd;
                   Main.OP = 0x20 | BIT5;
                   Additional(0, Split.ADDR = ADD16;); }
            { if (dddddd[BIT5]) { PCS <- ADD16; }; }
            {}
            { Cycle = 6; Stall = 0; Size = 2; }
            { Latency = 1; Usage = 1; }

JCLR BIT5, dddddd, ADD16
                 { DB.OP = 0x0A;
                   DB.MODE = 0xC0 | dddddd;
                   Main.OP = 0x00 | BIT5;
                   Additional(0, Split.ADDR = ADD16;); }
            { if (~dddddd[BIT5]) { PCS <- ADD16; }; }
            {}
            { Cycle = 6; Stall = 0; Size = 2; }
            { Latency = 1; Usage = 1; }

BTST BIT_MODE
                 { DB.OP = 0x0B;
                   DB.MODE = BIT_MODE >> 8;
                   Main.OP = 0x20 | (0xFF & BIT_MODE); }
            { C_F <- BIT_MODE; }
            {}
            { Cycle = 4+BIT_MODE; Stall = 0; Size = 1+BIT_MODE; }
            { Latency = 1; Usage = 1; }

BCHG BIT_MODE
                 { DB.OP = 0x0B;
                   DB.MODE = BIT_MODE >> 8;
                   Main.OP = 0x00 | (0xFF & BIT_MODE); }
            { C_F <- BIT_MODE; BIT_MODE <- ~BIT_MODE; }
            {}
            { Cycle = 4+BIT_MODE; Stall = 0; Size = 1+BIT_MODE; }
            { Latency = 1; Usage = 1; }

BSET BIT_MODE
                 { DB.OP = 0x0A;
```

```
                DB.MODE = BIT_MODE >> 8;
                Main.OP = 0x20 | (0xFF & BIT_MODE); }
        { C_F <- BIT_MODE; BIT_MODE <- 1; }
        {}
        { Cycle = 4+BIT_MODE; Stall = 0; Size = 1+BIT_MODE; }
        { Latency = 1; Usage = 1; }

BCLR BIT_MODE
              { DB.OP = 0x0A;
                DB.MODE = BIT_MODE >> 8;
                Main.OP = 0x00 | (0xFF & BIT_MODE); }
        { C_F <- BIT_MODE; BIT_MODE <- 0; }
        {}
        { Cycle = 4+BIT_MODE; Stall = 0; Size = 1+BIT_MODE; }
        { Latency = 1; Usage = 1; }

BTST BIT5, dddddd
                           { DB.OP = 0x0B;
                             DB.MODE = 0xC0 | dddddd;
                             Main.OP = 0x60 | BIT5; }
        { C_F <- dddddd[BIT5]; }
        {}
        { Cycle = 4; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

BCHG BIT5, dddddd
                           { DB.OP = 0x0B;
                             DB.MODE = 0xC0 | dddddd;
                             Main.OP = 0x40 | BIT5; }
        { C_F <- dddddd[BIT5]; dddddd[BIT5] <- ~dddddd[BIT5];}
        {}
        { Cycle = 4; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

BSET BIT5, dddddd
                           { DB.OP = 0x0A;
                             DB.MODE = 0xC0 | dddddd;
                             Main.OP = 0x60 | BIT5; }
        { C_F <- dddddd[BIT5]; dddddd[BIT5] <- 1; }
        {}
        { Cycle = 4; Stall = 0; Size = 1; }
        { Latency = 1; Usage = 1; }

BCLR BIT5, dddddd
                           { DB.OP = 0x0A;
```

```
                        DB.MODE = 0xC0 | dddddd;
                        Main.OP = 0x40 | BIT5; }
            { C_F <- dddddd[BIT5]; dddddd[BIT5] <- 0; }
            {}
            { Cycle = 4; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

REP BIT12
                { DB.OP = 0x06;
                  DB.MODE = 0xFF & BIT12;
                  Main.OP = 0xA0 | (0x0F & (BIT12 >> 8)); }
            { int 16 TMP;
              for (TMP <- BIT12; TMP != 0; TMP <- TMP - 1;) {
                        EVAL(PCS+1); }; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

REP dddddd
                       { DB.OP = 0x06;
                         DB.MODE = 0xC0 | dddddd;
                         Main.OP = 0x20; }
            { int 16 TMP;
              for (TMP <- dddddd; TMP != 0; TMP <- TMP - 1;) {
                        EVAL(PCS+1); }; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

REP MEM_MODE
                { DB.OP = 0x06;
                  DB.MODE = 0xFF & (MEM_MODE >> 8);
                  Main.OP = 0x20 | (0xFF & MEM_MODE); }
            { int 16 TMP;
              for (TMP <- MEM_MODE; TMP != 0; TMP <- TMP - 1;) {
                        EVAL(PCS+1); }; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

DO BIT12, BIT16
                { Additional(0, Split.ADDR = 0xFFFF & BIT16;);
                  DB.OP = 0x06;
                  DB.MODE = 0xFF & BIT12;
                  Main.OP = 0x80 | (0x0F & (BIT12 >> 8)); }
```

```
            { SPS <- SPS + 1; SSHS <- LAS; SSLS <- LCS;
              LCS <- BIT12; SPS <- SPS + 1; SSHS <- PCS;
              SSLS <- SRS; LAS <- BIT16;
              LF_F <- 1;
              while (LCS > 1) {
                while (PCS != (LAS - 1)) {
                  EVAL(PCS + 1); PCS <- PCS + 1;
                };
                PCS <- SSHS;
                LCS <- LCS - 1;
              };
              SRS <- 0x0040 & SSLS; SPS <- SPS - 1; LAS <- SSHS;
              LCS <- SSLS; SPS <- SPS - 1;
            }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 6; Stall = 0; Size = 2; }
            { Latency = 1; Usage = 1; }

DO dddddd, BIT16
                 { Additional(0, Split.ADDR = 0xFFFF & BIT16;);
                   DB.OP = 0x06;
                   DB.MODE = 0xC0 | dddddd;
                   Main.OP = 0x00; }
            { SPS <- SPS + 1; SSHS <- LAS; SSLS <- LCS;
              LCS <- dddddd; SPS <- SPS + 1; SSHS <- PCS;
              SSLS <- SRS; LAS <- BIT16;
              LF_F <- 1;
              while (LCS > 1) {
                while (PCS != (LAS - 1)) {
                  EVAL(PCS + 1); PCS <- PCS + 1;
                };
                PCS <- SSHS;
                LCS <- LCS - 1;
              };
              SRS <- 0x0040 & SSLS; SPS <- SPS - 1; LAS <- SSHS;
              LCS <- SSLS; SPS <- SPS - 1; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 6; Stall = 0; Size = 2; }
            { Latency = 1; Usage = 1; }

DO MEM_MODE, BIT16
                 { Additional(0, Split.ADDR = 0xFFFF & BIT16;);
                   DB.OP = 0x06;
                   DB.MODE = 0xFF & (MEM_MODE >> 8);
                   Main.OP = 0x00 | (0xFF & MEM_MODE); }
```

```
            { SPS <- SPS + 1; SSHS <- LAS; SSLS <- LCS;
              LCS <- MEM_MODE; SPS <- SPS + 1; SSHS <- PCS;
              SSLS <- SRS; LAS <- BIT16;
              LF_F <- 1;
              while (LCS > 1) {
                while (PCS != (LAS - 1)) {
                  EVAL(PCS + 1); PCS <- PCS + 1;
                };
                PCS <- SSHS;
                LCS <- LCS - 1;
              };
              SRS <- 0x0040 & SSLS; SPS <- SPS - 1; LAS <- SSHS;
              LCS <- SSLS; SPS <- SPS - 1; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 6; Stall = 0; Size = 2; }
            { Latency = 1; Usage = 1; }

LUA SEAN, DDDD
                          { DB.OP = 0x04;
                            DB.MODE = 0x40 | SEAN;
                            Main.OP = 0x10 | DDDD; }
            { DDDD <- SEAN; }
            {}
            { Cycle = 4; Stall = 0; Size = 1; }
            { Latency = 2; Usage = 1; }

T C_CODE, jjj, ACC, ttt, TTT
                 { DB.OP = 0x03;
                   DB.MODE = (C_CODE << 4) | ttt;
                   Main.OP = (jjj << 4) | (ACC << 3) | TTT; }
            { if (C_CODE) { ACC <- jjj; TTT <- ttt; }; }
            {}
            { Cycle = 2; Stall = 0; Size = 1; }
            { Latency = 2; Usage = 1; }

T C_CODE, B_D, A_D, ttt, TTT
                          { DB.OP = 0x03;
                            DB.MODE = (C_CODE << 4) | ttt;
                            Main.OP = 0x00 | TTT; }
            { if (C_CODE) { AREG <- BREG; TTT <- ttt; }; }
            {}
            { Cycle = 2; Stall = 0; Size = 1; }
            { Latency = 2; Usage = 1; }

T C_CODE, A_D, B_D, ttt, TTT
```

```
                                { DB.OP = 0x03;
                                  DB.MODE = (C_CODE << 4) | ttt;
                                  Main.OP = 0x08 | TTT; }
                { if (C_CODE) { BREG <- AREG; TTT <- ttt; }; }
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 2; Usage = 1; }

    T C_CODE, jjj, ACC
                     { DB.OP = 0x02;
                       DB.MODE = 0xF0 & (C_CODE << 4);
                       Main.OP = 0x78 & ((jjj << 4) | (ACC << 3)); }
                { if (C_CODE) { ACC <- jjj; }; }
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 2; Usage = 1; }

    T C_CODE, B_D, A_D
                                { DB.OP = 0x02;
                                  DB.MODE = 0xF0 & (C_CODE << 4);
                                  Main.OP = 0x00; }
                { if (C_CODE) { AREG <- BREG; }; }
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 2; Usage = 1; }

    T C_CODE, A_D, B_D
                                { DB.OP = 0x02;
                                  DB.MODE = 0xF0 & (C_CODE << 4);
                                  Main.OP = 0x08; }
                { if (C_CODE) { BREG <- AREG; }; }
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 2; Usage = 1; }

/***** describe the RTL - not too useful and rather complex *****/
    NORM TTT, ACC
                                { DB.OP = 0x01;
                                  DB.MODE = 0xD8 | TTT;
                                  Main.OP = 0x15 | (ACC << 3); }
                {}
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }
```

```
/***** describe the RTL - not too useful and rather complex *****/
    DIV JJ, ACC
                    { DB.OP = 0x01;
                      DB.MODE = 0x80;
                      Main.OP = 0x40 | (JJ << 4) | (ACC << 3) ; }
                {}
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

    ORI BIT8, EE
                                { DB.OP = 0x00;
                                  DB.MODE = BIT8;
                                  Main.OP = 0xF8 | EE; }
                { EE <- SORm(BIT8, EE); }
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

    ANDI BIT8, EE
                                { DB.OP = 0x00;
                                  DB.MODE = BIT8;
                                  Main.OP = 0xB8 | EE; }
                { EE <- ANDSm(BIT8, EE); }
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

    ENDDO
                                { DB.OP = 0x00;
                                  DB.MODE = 0x00;
                                  Main.OP = 0x8C; }
                { SRS <- 0x0040 & SSLS; SPS <- SPS - 1; LAS <- SSHS;
                  LCS <- SSLS; SPS <- SPS - 1; }
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

    STOP
                                { DB.OP = 0x00;
                                  DB.MODE = 0x00;
                                  Main.OP = 0x87; }
                { HALT(); }
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
```

```
                { Latency = 1; Usage = 1; }

    WAIT
                                { DB.OP = 0x00;
                                  DB.MODE = 0x00;
                                  Main.OP = 0x86; }
                { HALT(); }
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

/***** cannot really describe without knowledge of periferals *****/
    RESET
                                { DB.OP = 0x00;
                                  DB.MODE = 0x00;
                                  Main.OP = 0x84; }
                {}
                {}
                { Cycle = 4; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

    RTS
                                { DB.OP = 0x00;
                                  DB.MODE = 0x00;
                                  Main.OP = 0x0C; }
                { PCS <- SSHS; SPS <- SPS - 1; }
                {}
                { Cycle = 4; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

/***** cannot describe this without knowledge of interrupts *****/
    SWI
                                { DB.OP = 0x00;
                                  DB.MODE = 0x00;
                                  Main.OP = 0x06; }
                {}
                {}
                { Cycle = 8; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

    RTI
                                { DB.OP = 0x00;
                                  DB.MODE = 0x00;
                                  Main.OP = 0x04; }
                { PCS <- SSHS; SRS <- SSLS; SPS <- SPS - 1; }
```

```
                {}
                { Cycle = 4; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

/***** cannot describe this without knowledge of exceptions *****/
    ILLEGAL
                                { DB.OP = 0x00;
                                  DB.MODE = 0x00;
                                  Main.OP = 0x01; }
                {}
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }


    NOP
                                { DB.OP = 0x00;
                                  DB.MODE = 0x00;
                                  Main.OP = 0x00; }
                { NOP(); }
                {}
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

    MACR QQQ, ACC
                     { Main.OP = 0x83 | (QQQ << 4) | (ACC << 3); }
                { ACC <- RNDm(ADDm(ACC, QQQ)); }
                { CSET_ALL(OVF,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                           SIGNED(ACC))
                  CSET_FLAG(LIMIT, L_F); }
                { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
                { Latency = 1; Usage = 1; }

    MACR SIGN, QQQ, ACC
                     { Main.OP = 0x87 | (QQQ << 4) | (ACC << 3); }
                { ACC <- RNDm(SUBm(ACC, QQQ)); }
                { CSET_ALL(OVF,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                           SIGNED(ACC))
                  CSET_FLAG(LIMIT, L_F); }
                { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
                { Latency = 1; Usage = 1; }

    MAC QQQ, ACC
                     { Main.OP = 0x82 | (QQQ << 4) | (ACC << 3); }
                { ACC <- ADDm(ACC, QQQ); }
```

```
            { CSET_ALL(OVF,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

MAC SIGN, QQQ, ACC
                 { Main.OP = 0x86 | (QQQ << 4) | (ACC << 3); }
            { ACC <- SUBm(ACC, QQQ); }
            { CSET_ALL(OVF,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

MPYR QQQ, ACC
                 { Main.OP = 0x81 | (QQQ << 4) | (ACC << 3); }
            { ACC <- RNDm(QQQ); }
            { CSET_ALL(0,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F);
              V_F <- 0; }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

MPYR SIGN, QQQ, ACC
                 { Main.OP = 0x85 | (QQQ << 4) | (ACC << 3); }
            { ACC <- -RNDm(QQQ); }
            { CSET_ALL(0,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F);
              V_F <- 0; }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

MPY QQQ, ACC
                 { Main.OP = 0x80 | (QQQ << 4) | (ACC << 3); }
            { ACC <- QQQ; }
            { CSET_ALL(0,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F);
              V_F <- 0; }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }
```

```
MPY SIGN, QQQ, ACC
                { Main.OP = 0x84 | (QQQ << 4) | (ACC << 3); }
            { ACC <- -QQQ; }
            { CSET_ALL(0,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F);
              V_F <- 0; }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

CMPM jjj, ACC
                { Main.OP = 0x07 | (jjj << 4) | (ACC << 3); }
            { NULL <-
                SUBCm(ABSm(SEXT(jjj, 24, 56) << 24), ABSm(ACC)); }
            { CSET_ALL(OVF,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

CMPM B_D, A_D
                              { Main.OP = 0x07; }
            { NULL <- SUBCm(ABSm(BREG), ABSm(AREG)); }
            { CSET_ALL(OVF,(AREG == 0),(AREG[55] == 0),UNORM(AREG),
                       SIGNED(AREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

CMPM A_D, B_D
                              { Main.OP = 0x0F; }
            { NULL <- SUBCm(ABSm(AREG), ABSm(BREG)); }
            { CSET_ALL(OVF,(BREG == 0),(BREG[55] == 0),UNORM(BREG),
                       SIGNED(BREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

AND JJ, ACC_1
                { Main.OP = 0x46 | (JJ << 4) | (ACC_1 << 3); }
            { ACC_1 <- AND(ACC_1, JJ, 24); }
            { V_F <- 0; CSET_FLAG(LIMIT, L_F);
              CSET_FLAG(ACC_1[47], N_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }
```

```
CMP jjj, ACC
                { Main.OP = 0x05 | (jjj << 4) | (ACC << 3); }
            { NULL <- SUBCm(SEXT(jjj, 24, 56) << 24, ACC); }
            { CSET_ALL(OVF,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

CMP B_D, A_D
                              { Main.OP = 0x05; }
            { NULL <- SUBCm(BREG, AREG); }
            { CSET_ALL(OVF,(AREG == 0),(AREG[55] == 0),UNORM(AREG),
                       SIGNED(AREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

CMP A_D, B_D
                              { Main.OP = 0x0D; }
            { NULL <- SUBCm(AREG, BREG); }
            { CSET_ALL(OVF,(BREG == 0),(BREG[55] == 0),UNORM(BREG),
                       SIGNED(BREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

SUB JJJ, ACC
                { Main.OP = 0x04 | (JJJ << 4) | (ACC << 3); }
            { ACC  <- SUBm(ACC, JJJ); }
            {}
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

SUB B_D, A_D
                              { Main.OP = 0x04; }
            { AREG  <- SUBCm(AREG, BREG); }
            { CSET_ALL(OVF,(AREG == 0),(AREG[55] == 0),UNORM(AREG),
                       SIGNED(AREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

SUB A_D, B_D
```

```
                        { Main.OP = 0x0C; }
            { BREG  <- SUBCm(BREG, AREG); }
            { CSET_ALL(OVF,(BREG == 0),(BREG[55] == 0),UNORM(BREG),
                       SIGNED(BREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

EOR JJ, ACC_1
                { Main.OP = 0x43 | (JJ << 4) | (ACC_1 << 3); }
            { ACC_1 <- XORm(JJ, ACC_1); }
            { V_F <- 0; CSET_FLAG(LIMIT, L_F);
              CSET_FLAG(ACC_1[47], N_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

OR JJ, ACC_1
                { Main.OP = 0x42 | (JJ << 4) | (ACC_1 << 3); }
            { ACC_1 <- ORm(JJ, ACC_1); }
            { V_F <- 0; CSET_FLAG(LIMIT, L_F);
              CSET_FLAG(ACC_1[47], N_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

TFR jjj, ACC
                { Main.OP = 0x01 | (jjj << 4) | (ACC << 3); }
            { ACC <- jjj; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 2; Usage = 1; }

TFR B_D, A_D
                        { Main.OP = 0x01; }
            { AREG <- BREG; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 2; Usage = 1; }

TFR A_D, B_D
                        { Main.OP = 0x09; }
            { BREG <- AREG; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 2; Usage = 1; }
```

```
ADD JJJ, ACC
                { Main.OP = 0x00 | (JJJ << 4) | (ACC << 3); }
            { ACC  <- ADDCm(ACC, JJJ); }
            { CSET_ALL(OVF, (ACC == 0), (ACC[55] == 0), UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

ADD B_D, A_D
                           { Main.OP = 0x00; }
            { AREG  <- ADDCm(AREG, BREG); }
            { CSET_ALL(OVF,(AREG == 0),(AREG[55] == 0),UNORM(AREG),
                       SIGNED(AREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

ADD A_D, B_D
                           { Main.OP = 0x08; }
            { BREG  <- ADDCm(BREG, AREG); }
            { CSET_ALL(OVF,(BREG == 0),(BREG[55] == 0),UNORM(BREG),
                       SIGNED(BREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

SBC j, ACC
                { Main.OP = 0x25 | (j << 4) | (ACC << 3); }
            { ACC <-
                SUBC(ACC, SEXT(j, 48, 56), C_F, C_F, 56, "trn"); }
            { CSET_ALL(OVF,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

ADC j, ACC
                { Main.OP = 0x21 | (j << 4) | (ACC << 3); }
            { ACC <-
                ADDC(ACC, SEXT(j, 48, 56), C_F, C_F, 56, "trn"); }
            { CSET_ALL(OVF, (ACC == 0), (ACC[55] == 0), UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
```

```
            { Latency = 1; Usage = 1; }

ROL ACCS
                            { Main.OP = 0x37 | (ACCS << 3); }
            { ACCS <- ASL(ACCS, 1, C_F, C_F, 48); }
            { V_F <- 0; CSET_FLAG(LIMIT, L_F);
              CSET_FLAG(ACCS[47], N_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

NEG ACC
                            { Main.OP = 0x36 | (ACC << 3); }
            { ACC <- -ACC; }
            { CSET_ALL(OVF,(ACC == 0),(ACC[55] == 0),UNORM(ACC),
                       SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

LSL ACCS
                            { Main.OP = 0x33 | (ACCS << 3); }
            { ACCS <- ASL(ACCS, 1, 0, C_F, 48); }
            { V_F <- 0; CSET_FLAG(LIMIT, L_F);
              CSET_FLAG(ACCS[47], N_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

ASL ACC
                            { Main.OP = 0x32 | (ACC << 3); }
            { ACC <- ASL(ACC, 1, 0, C_F, 56); }
            { CSET_ALL(CHANGED(ACC[55]),(ACC == 0),(ACC[55]  == 0),
                       UNORM(ACC), SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

ROR ACCS
                            { Main.OP = 0x27 | (ACCS << 3); }
            { ACCS <- ASR(ACCS, 1, C_F, C_F, 48); }
            {}
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

ABS ACC
                            { Main.OP = 0x26 | (ACC << 3); }
```

```
            { ACC <- ABSm(ACC); }
            { CSET_ALL(OVF, (ACC == 0), (ACC[55] == 0),
                       UNORM(ACC), SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

LSR ACCS
                              { Main.OP = 0x23 | (ACCS << 3); }
            { ACCS <- ASR(ACCS, 1, 0, C_F, 48); }
            { V_F <- 0; CSET_FLAG(LIMIT, L_F);
              CSET_FLAG(ACCS[47], N_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

ASR ACC
                              { Main.OP = 0x22 | (ACC << 3); }
            { ACC <- ASR(ACC, 1, 0, C_F, 56); }
            { CSET_ALL(CHANGED(ACC[55]),(ACC == 0),(ACC[55] == 0),
                       UNORM(ACC), SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

NOT ACC_1
                              { Main.OP = 0x17 | (ACC_1 << 3); }
            { ACC_1 <- NOTm(ACC_1); }
            { V_F <- 0; CSET_FLAG(LIMIT, L_F);
              CSET_FLAG(ACC_1[47], N_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

SUBL B_D, A_D
                              { Main.OP = 0x16; }
            { AREG <- SUBCm(MULm(2, AREG), BREG); }
            { CSET_ALL(OVF | CHANGED(AREG[55]), (AREG == 0),
                         (AREG[55] == 0),UNORM(AREG),SIGNED(AREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

SUBL A_D, B_D
                              { Main.OP = 0x1E; }
            { BREG <- SUBCm(MULm(2, BREG), AREG); }
            { CSET_ALL(OVF | CHANGED(BREG[55]), (BREG == 0),
```

```
                    (BREG[55] == 0),UNORM(BREG),SIGNED(BREG))
      CSET_FLAG(LIMIT, L_F); }
    { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
    { Latency = 1; Usage = 1; }

CLR ACC
                          { Main.OP = 0x13 | (ACC << 3); }
    { ACC <- 0; }
    { CSET_FLAG(LIMIT, L_F); E_F <- 0; U_F <- 1; N_F <- 0;
      Z_F <- 1; V_F <- 0; }
    { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
    { Latency = 1; Usage = 1; }

ADDL B_D, A_D
                          { Main.OP = 0x12; }
    { AREG <- ADDCm(MULm(2, AREG), BREG); }
    { CSET_ALL(OVF | CHANGED(AREG[55]), (AREG == 0),
                    (AREG[55] == 0),UNORM(AREG),SIGNED(AREG))
      CSET_FLAG(LIMIT, L_F); }
    { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
    { Latency = 1; Usage = 1; }

ADDL A_D, B_D
                          { Main.OP = 0x1A; }
    { BREG <- ADDCm(MULm(2, BREG), AREG); }
    { CSET_ALL(OVF | CHANGED(BREG[55]), (BREG == 0),
                    (BREG[55] == 0),UNORM(BREG),SIGNED(BREG))
      CSET_FLAG(LIMIT, L_F); }
    { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
    { Latency = 1; Usage = 1; }

RND ACC
                          { Main.OP = 0x11 | (ACC << 3); }
    { switch (S_MODE) {
        case 1: { ACC <- RND(ACC, 32, "down"); };
        case 2: { ACC <- RND(ACC, 32, "up"); };
      }; }
    { CSET_ALL(OVF, (ACC == 0), (ACC[55] == 0),
                    UNORM(ACC), SIGNED(ACC))
      CSET_FLAG(LIMIT, L_F); }
    { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
    { Latency = 1; Usage = 1; }

SUBR B_D, A_D
                          { Main.OP = 0x06; }
```

```
            { AREG <- SUBCm(DIVm(AREG, 2), BREG); }
            { CSET_ALL(OVF, (AREG == 0), (AREG[55] == 0),
                       UNORM(AREG), SIGNED(AREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

SUBR A_D, B_D
                            { Main.OP = 0x0E; }
            { BREG <- SUBCm(DIVm(BREG, 2), AREG); }
            { CSET_ALL(OVF, (BREG == 0), (BREG[55] == 0),
                       UNORM(BREG), SIGNED(BREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

TST ACC
                            { Main.OP = 0x03 | (ACC << 3); }
            { NULL <- SUBm(ACC, 0); }
            { CSET_ALL(0, (ACC == 0), (ACC[55] == 0),
                       UNORM(ACC), SIGNED(ACC))
              CSET_FLAG(LIMIT, L_F);  V_F <- 0; }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

ADDR B_D, A_D
                            { Main.OP = 0x02; }
            { AREG <- ADDCm(DIVm(AREG, 2), BREG); }
            { CSET_ALL(OVF, (AREG == 0), (AREG[55] == 0),
                       UNORM(AREG), SIGNED(AREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

ADDR A_D, B_D
                            { Main.OP = 0x0A; }
            { BREG <- ADDCm(DIVm(BREG, 2), AREG); }
            { CSET_ALL(OVF, (BREG == 0), (BREG[55] == 0),
                       UNORM(BREG), SIGNED(BREG))
              CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2 + DBM; Stall = 0; Size = 1 + DBM; }
            { Latency = 1; Usage = 1; }

MOVEP XEA, XPP
                            { DB.OP = 0x08;
```

```
                              DB.MODE = 0xC0 | XEA;
                              Main.OP = 0xC0 | XPP; }
            { XPP <- XEA; }
            { XEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4 + XEA; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

MOVEP YEA, XPP
                            { DB.OP = 0x08;
                              DB.MODE = 0xC0 | YEA;
                              Main.OP = 0x80 | XPP; }
            { XPP <- YEA; }
            { YEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4 + YEA; Stall = 0; Size = 1 + YEA; }
            { Latency = 1; Usage = 1; }

MOVEP BIT16, XPP
                            { DB.OP = 0x08;
                              DB.MODE = 0xF4;
                              Main.OP = 0x80 | XPP; }
            { XPP <- BIT16; }
            { CSET_FLAG(LIMIT, L_F);}
            { Cycle = 4; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

MOVEP XPP, XEA
                            { DB.OP = 0x08;
                              DB.MODE = 0x80 | XEA;
                              Main.OP = 0xC0 | XPP; }
            { XEA <- XPP; }
            { XEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4 + XEA; Stall = 0; Size = 1 + XEA; }
            { Latency = 1; Usage = 1; }

MOVEP XPP, YEA
                            { DB.OP = 0x08;
                              DB.MODE = 0x80 | YEA;
                              Main.OP = 0x80 | XPP; }
            { YEA <- XPP; }
            { YEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4 + YEA; Stall = 0; Size = 1 + YEA; }
            { Latency = 1; Usage = 1; }

MOVEP XEA, YPP
                            { DB.OP = 0x09;
```

```
                              DB.MODE = 0xC0 | XEA;
                              Main.OP = 0xC0 | YPP; }
            { YPP <- XEA; }
            { XEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4 + XEA; Stall = 0; Size = 1 + XEA; }
            { Latency = 1; Usage = 1; }

MOVEP YEA, YPP
                            { DB.OP = 0x09;
                              DB.MODE = 0xC0 | YEA;
                              Main.OP = 0x80 | YPP; }
            { YPP <- YEA; }
            { YEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4 + YEA; Stall = 0; Size = 1 + YEA; }
            { Latency = 1; Usage = 1; }

MOVEP BIT16, YPP
                            { DB.OP = 0x09;
                              DB.MODE = 0xF4;
                              Main.OP = 0x80 | YPP; }
            { YPP <- BIT16; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

MOVEP YPP, XEA
                            { DB.OP = 0x09;
                              DB.MODE = 0x80 | XEA;
                              Main.OP = 0xC0 | YPP; }
            { XEA <- YPP; }
            { XEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4 + XEA; Stall = 0; Size = 1 + XEA; }
            { Latency = 1; Usage = 1; }

MOVEP YPP, YEA
                            { DB.OP = 0x09;
                              DB.MODE = 0x80 | YEA;
                              Main.OP = 0x80 | YPP; }
            { YEA <- YPP; }
            { YEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4 + YEA; Stall = 0; Size = 1 + YEA; }
            { Latency = 1; Usage = 1; }

MOVEP PEA, XPP
                            { DB.OP = 0x08;
```

```
                              DB.MODE = 0xC0 | PEA;
                              Main.OP = 0x40 | XPP; }
            { XPP <- PEA; }
            { PEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 6 + PEA; Stall = 0; Size = 1 + PEA; }
            { Latency = 1; Usage = 1; }

MOVEP XPP, PEA
                            { DB.OP = 0x08;
                              DB.MODE = 0x40 | PEA;
                              Main.OP = 0x40 | XPP; }
            { PEA <- XPP; }
            { PEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 6 + PEA; Stall = 0; Size = 1 + PEA; }
            { Latency = 1; Usage = 1; }

MOVEP PEA, YPP
                            { DB.OP = 0x09;
                              DB.MODE = 0xC0 | PEA;
                              Main.OP = 0x40 | YPP; }
            { YPP <- PEA; }
            { PEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 6 + PEA; Stall = 0; Size = 1 + PEA; }
            { Latency = 1; Usage = 1; }

MOVEP YPP, PEA
                            { DB.OP = 0x09;
                              DB.MODE = 0x40 | PEA;
                              Main.OP = 0x40 | YPP; }
            { PEA <- YPP; }
            { PEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 6 + PEA; Stall = 0; Size = 1 + PEA; }
            { Latency = 1; Usage = 1; }

MOVEP dddddd, XPP
                            { DB.OP = 0x08;
                              DB.MODE = 0xC0 | dddddd;
                              Main.OP = 0x00 | XPP; }
            { XPP <- dddddd; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

MOVEP XPP, dddddd
                            { DB.OP = 0x08;
```

```
                              DB.MODE = 0x40 | dddddd;
                              Main.OP = 0x00 | XPP; }
            { dddddd <- XPP; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

MOVEP dddddd, YPP
                            { DB.OP = 0x09;
                              DB.MODE = 0xC0 | dddddd;
                              Main.OP = 0x00 | YPP; }
            { YPP <- dddddd; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

MOVEP YPP, dddddd
                            { DB.OP = 0x09;
                              DB.MODE = 0x40 | dddddd;
                              Main.OP = 0x00 | YPP; }
            { dddddd <- YPP; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 4; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

MOVEM dddddd, PEA
                            { DB.OP = 0x07;
                              DB.MODE = 0x40 | PEA;
                              Main.OP = 0x80 | dddddd; }
            { PEA <- dddddd; }
            { PEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 6 + PEA; Stall = 0; Size = 1 + PEA; }
            { Latency = 1; Usage = 1; }

MOVEM PEA, dddddd
                            { DB.OP = 0x07;
                              DB.MODE = 0x80 | PEA;
                              Main.OP = 0x80 | dddddd; }
            { dddddd <- PEA; }
            { PEA; CSET_FLAG(LIMIT, L_F); }
            { Cycle = 6 + PEA; Stall = 0; Size = 1 + PEA; }
            { Latency = 1; Usage = 1; }

MOVEM dddddd, PAA
                            { DB.OP = 0x07;
```

```
                              DB.MODE = 0x00 | PAA;
                              Main.OP = 0x00 | dddddd; }
            { PAA <- dddddd; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 6; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

MOVEM PAA, dddddd
                            { DB.OP = 0x07;
                              DB.MODE = 0x80 | PAA;
                              Main.OP = 0x00 | dddddd; }
            { dddddd <- PAA; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 6; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

/***** If you got this far you need a hobby *****/
MOVEC XAA, ccccc
                            { DB.OP = 0x05;
                              DB.MODE = 0x00 | XAA;
                              Main.OP = 0x20 | ccccc; }
            { ccccc <- XAA; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

MOVEC ccccc, XAA
                            { DB.OP = 0x05;
                              DB.MODE = 0x80 | XAA;
                              Main.OP = 0x20 | ccccc; }
            { XAA <- ccccc; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

MOVEC YAA, ccccc
                            { DB.OP = 0x05;
                              DB.MODE = 0x00 | YAA;
                              Main.OP = 0x60 | ccccc; }
            { ccccc <- YAA; }
            { CSET_FLAG(LIMIT, L_F); }
            { Cycle = 2; Stall = 0; Size = 1; }
            { Latency = 1; Usage = 1; }

MOVEC ccccc, YAA
```

```
                                { DB.OP = 0x05;
                                  DB.MODE = 0x80 | YAA;
                                  Main.OP = 0x60 | ccccc; }
                { YAA <- ccccc; }
                { CSET_FLAG(LIMIT, L_F); }
                { Cycle = 2; Stall = 0; Size = 1; }
                { Latency = 1; Usage = 1; }

// ---------------------------------------------------------------
// Any restrictions (in the form of rules) of which instructions go
// together and which do not.

Section Constraints

// Moves cannot be done at the same time as some operations because
// of a bitfield conflict on the DBM fields:
// Note that the list of instructions could be made shorter using
// wildcards at the expense of convenience though.

~( (MOVE *) &
   ((((((((((((((((((((((((((((((ANDI *)       |
    (BCHG *))        |
    (BCLR *))        |
    (BSET *))        |
    (BTST *))        |
    (DIV *))         |
    (DO *))          |
    (ENDDO))         |
    (ILLEGAL))       |
    (J *))           |
    (JI *))          |
    (JCLR *))        |
    (JMP *))         |
    (JMPI *))        |
    (JS *))          |
    (JSI *))         |
    (JSCLR *))       |
    (JSET *))        |
    (JSR* *))        |
    (JSSET *))       |
    (LUA *))         |
    (MOVEC *))       |
    (MOVEM *))       |
    (MOVEP *))       |
    (NORM *))        |
```

```
    (ORI *))        |
    (REP *))        |
    (RESET))        |
    (RTI))          |
    (RTS))          |
    (STOP))         |
    (SWI))          |
    (T *))          |
    (WAIT)))

// You cannot do things like ADD A, A etc
~(ADD @[1], @[1])
~(CMP @[1], @[1])
~(CMPM @[1], @[1])
~(SUBL @[1], @[1])
~(SUBR @[1], @[1])

// Where you do parallel moves to both X and Y mem banks, make sure
// they use opposite banks in their addressing modes.
~(MOVE *R[0123]*R[0123]*)
~(MOVE *R[4567]*R[4567]*)

// The ANDI IMM, MR instruction cannot be used immediately before an
// ENDDO or RTI instruction and cannot be one of the last three
// instructions in a DO loop (at LA-2, LA-1, or LA)
// ***** we have no way of describing the latter part of this
// constraint for the moment *****
~((ANDI *, MR) & [1]((ENDDO) | (RTI)))

// ANDI IMM, CCR cannot be used immediately before an RTI
~((ANDI *, CCR) & [1](RTI))

// Do is very picky as to what can come before or after it simply
// because of all the control flow that has to be taken care of.

// Instructions that cannot be performed immediately before a DO
// MOVEC to LA, LC, SSH, SSL, or SP
// MOVEM to LA, LC, SSH, SSL, or SP
// MOVEP to LA, LC, SSH, SSL, or SP
// MOVEC from SSH
// MOVEM from SSH
// MOVEP from SSH
~((((((MOVE? *, LA) |
  (MOVE? *, LC))  |
  (MOVE? *, SSH)) |
```

```
  (MOVE? *, SSL)) |
  (MOVE? *, SP)) & [1](DO *))
~((MOVE? SSH, *) & [1](DO *))

// You cannot repeat a DO usign REP
~((REP *) & [1](DO *))

// This is also illegal
~(DO SSH,*)

// The following are also illegal but we have no way of describing the
// fact so far

// The following instructions are not valid when the L_F bit is set
// JSR to LA
// JS C_CODE to LA
// JSCLR to LA
// JSSET to LA

// The following instructions cannot begin at the indicated positions
// near the end of a DO loop
// At LA-2, LA-1, and LA
//      DO
//      MOVEC to LA, LC, SSH, SSL, or SP
//      MOVEM to LA, LC, SSH, SSL, or SP
//      MOVEP to LA, LC, SSH, SSL, or SP
//      MOVEC from SSH
//      MOVEM from SSH
//      MOVEP from SSH
//      ANDI *,MR
//      ORI *,MR
//      Two word instructions which read LC, SP, or SSL
// At LA-1
//      single word instructions (except REP)
//      which read LC, SP, or SSL
//      JCLR
//      JSET
//      Two word JMP
//      Two word J C_CODE
// At LA
//      Any two word instruction
//      J C_CODE, JCLR, JSET, JMP, JS C_CODE, JSR, REP, RESET, RTI,
//      RTS, STOP, WAIT, ENDDO

// ENDDO timing is just as demanding as DO
```

```
// The following instructions can never appear immediately
// before an ENDDO
//      MOVEC to LA, LC, SSH, SSL, or SP
//      MOVEM to LA, LC, SSH, SSL, or SP
//      MOVEP to LA, LC, SSH, SSL, or SP
//      MOVEC from SSH
//      MOVEM from SSH
//      MOVEP from SSH
//      ANDI *,MR
//      ORI *,MR
~((((((MOVE? *, LA) |
  (MOVE? *, LC))  |
  (MOVE? *, SSH)) |
  (MOVE? *, SSL)) |
  (MOVE? *, SP)) & [1](ENDDO))
~((MOVE? SSH, *) & [1](ENDDO))
~((ANDI *,MR) & [1](ENDDO))
~((ORI *,MR) & [1](ENDDO))
// ENDDO is also illegal as the last instruction in a DO loop (at LA)

// A J C_CODE instruction cannot be repeated with a REP instruction
~((REP *) & [1](J *))
~((REP *) & [1](JI *))

// A JCLR instruction cannot be repeated with a REP instruction
~((REP *) & [1](JCLR *))
// A JCRL SSH or JCLR SSL cannot follow an instruction that changes SP
~((((((((((((((((DO *) | (ENDDO)) | (JS *)) | (JSI *)) |
   (JSCLR *)) | (JSRI *)) | ( JSR *)) | (JSSET *)) |
   (RTI)) | (RTS)) | (BCHG *,SP)) | (BCLR *,SP)) |
   (BSET *,SP)) | (MOVEC *,SP)) | (MOVEM *,SP)) |
   (MOVEP *,SP)) & [1]((JCLR *,SSH,*) | [1](JCLR *,SSL,*)))
// A JCLR located at LA, LA-1, or LA-2 of the DO loop cannot specify
// the program controller registers SR, SP, SSH, SSL, LA, or LC as
// it's target

// A JMP instruction cannot be repeated with a REP instruction
~((REP *) & [1](JMP *))
~((REP *) & [1](JMPI *))
// A JMP instruction used within a DO loop cannot begin at the address
// LA within that DO loop

// A JS C_CODE instruction cannot be repeated with a REP instruction
~((REP *) & [1](JS *))
~((REP *) & [1](JSI *))
```

```
// A JS C_CODE instruction used within a DO loop cannot begin at the
// address LA within that DO loop
// A JS C_CODE instruction used within a DO loop cannot specify the
// loop address (LA) as it's target

// A JSCLR instruction cannot be repeated with a REP instruction
~((REP *) & [1](JSCLR *))
// A JSCRL SSH or JSCLR SSL cannot follow an instruction that
// changes the SP
~((((((((((((((((DO *) | (ENDDO)) | (JS *)) | (JSI *)) |
   (JSCLR *)) | (JSRI *)) | ( JSR *)) | (JSSET *)) |
   (RTI)) | (RTS)) | (BCHG *,SP)) | (BCLR *,SP)) |
   (BSET *,SP)) | (MOVEC *,SP)) | (MOVEM *,SP)) |
   (MOVEP *,SP)) & [1]((JSCLR *,SSH,*) | (JSCLR *,SSL,*)))
// A JSCLR located at LA, LA-1, or LA-2 of the DO loop cannot specify
// the program controller registers SR, SP, SSH, SSL, LA, or LC as
// it's target
// A JSCLR instruction used within a DO loop cannot specify the
// loop address (LA) as it's target

// A JSET instruction cannot be repeated with a REP instruction
~((REP *) & [1](JSET *))
// A JSET SSH or JSET SSL cannot follow an instruction that changes SP
~((((((((((((((((DO *) | (ENDDO)) | (JS *)) | (JSI *)) |
   (JSCLR *)) | (JSRI *)) | ( JSR *)) | (JSSET *)) |
   (RTI)) | (RTS)) | (BCHG *,SP)) | (BCLR *,SP)) |
   (BSET *,SP)) | (MOVEC *,SP)) | (MOVEM *,SP)) |
   (MOVEP *,SP)) & [1]((JSET *,SSH,*) | (JSET *,SSL,*)))
// A JSET located at LA, LA-1, or LA-2 of the DO loop cannot specify
// the program controller registers SR, SP, SSH, SSL, LA, or LC as
// it's target
// A JSET instruction used within a DO loop cannot specify the
// loop address (LA) as it's target

// A JSR instruction cannot be repeated with a REP instruction
~((REP *) & [1](JSR *))
~((REP *) & [1](JSRI *))
// A JSR instruction used within a DO loop cannot begin at the address
// LA within that DO loop
// A JSR instruction used within a DO loop cannot specify the
// loop address (LA) as it's target

// A JSSET instruction cannot be repeated with a REP instruction
~((REP *) & [1](JSSET *))
// A JSSET SSH or JSSET SSL cannot follow an instruction that
```

```
// changes the SP
~((((((((((((((((DO *) | (ENDDO)) | (JS *)) | (JSI *)) |
   (JSCLR *)) | (JSRI *)) | ( JSR *)) | (JSSET *)) |
   (RTI)) | (RTS)) | (BCHG *,SP)) | (BCLR *,SP)) |
   (BSET *,SP)) | (MOVEC *,SP)) | (MOVEM *,SP)) |
   (MOVEP *,SP)) & [1]((JSSET *,SSH,*) | (JSSET *,SSL,*)))
// A JSSET located at LA, LA-1, or LA-2 of the DO loop cannot specify
// the program controller registers SR, SP, SSH, SSL, LA, or LC as
// it's target
// A JSSET instruction used within a DO loop cannot specify the
// loop address (LA) as it's target

// A MOVEC SSH, SSH instruction is illegal and cannot be used
~(MOVEC SSH,SSH)

// A MOVE? instruction which specifies SP as the destination operand
// cannot be used immediately before a MOVEC, MOVEM, or MOVEP
// instruction which specifies SSH or SSL as the source operand.
~((MOVE? *,SP) & [1]((MOVE? SSH,*) | (MOVE? SSL,*)))

// A REP instruction cannot repeat the following:
// DO, J, JI, JCLR, JMP, JSET, JS, JSI, JSCLR, JSR, JSRI, JSSET, REP,
// RTI, RTS, STOP, SWI, WAIT, ENDDO
~((REP *) & [1](((((((((((((((((((DO *) | (J *)) |
                  (JI *)) | (JCLR *)) | (JMP *)) |
                  (JMPI *)) | (JSET *)) | (JS *)) |
                  (JSI *)) | (JSCLR *)) | (JSR *)) |
                  (JSRI *)) | (JSSET *)) | (REP *)) |
                  (RTI)) | (RTS)) | (STOP)) | (SWI)) |
                  (WAIT)) | (ENDDO)))
// Also, the REP instruction cannot be the last instruction in a DO
// loop

// A RESET cannot be the last instruction in a DO loop (at LA).

// The following instructions can never appear immediately
// before an RTI
//      MOVEC to LA, LC, SSH, SSL, or SP
//      MOVEM to LA, LC, SSH, SSL, or SP
//      MOVEP to LA, LC, SSH, SSL, or SP
//      MOVEC from SSH
//      MOVEM from SSH
//      MOVEP from SSH
//      ANDI *,MR
//      ORI *,MR
```

```
//      ANDI *,CCR
//      ORI *,CCR
~((((((MOVE? *, LA) |
  (MOVE? *, LC))  |
  (MOVE? *, SSH)) |
  (MOVE? *, SSL)) |
  (MOVE? *, SP)) & [1](RTI))
~((MOVE? SSH, *) & [1](RTI))
~((ANDI *,MR) & [1](RTI))
~((ORI *,MR) & [1](RTI))
~((ANDI *,CCR) & [1](RTI))
~((ORI *,CCR) & [1](RTI))
// An RTI cannot be the last instruction in a DO loop (at LA)

// The following instructions can never appear immediately
// before an RTS
//      MOVEC to LA, LC, SSH, SSL, or SP
//      MOVEM to LA, LC, SSH, SSL, or SP
//      MOVEP to LA, LC, SSH, SSL, or SP
//      MOVEC from SSH
//      MOVEM from SSH
//      MOVEP from SSH
~((((((MOVE? *, LA) |
  (MOVE? *, LC))  |
  (MOVE? *, SSH)) |
  (MOVE? *, SSL)) |
  (MOVE? *, SP)) & [1](RTS))
// You cannot repeat an RTS usign REP
~((REP *) & [1](RTS))
// Also, an RTS cannot be the last instruction in a DO loop (at LA)

// A STOP instruction cannot be used in a fast interrupt routine

// An SWI instruction cannot be used in a fast interrupt routine

// A WAIT instruction cannot be used in a fast interrupt routine
// A WAIT instruction cannot be the last instruction in a DO loop
// (at LA)

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

Section Optional
```

Appendix D

Example Applications

D.1 Applications for the SPAM VLIW-1 Architecture

These are the assembly applications for the SPAM VLIW-1 architecture Version a.

D.1.1 Finite Impulse Response (FIR) Filter

```
/*
 * A simple FIR filter for the example architecture
 */

/*
    Memory Map
    Writeback loc   IM[1]
    Outer Loop Cnt  IM[2]
    coefficients    IM[3]-IM[10]
    Samples         DM[0]-DM[1023]
    Output          DM[2048]-DM[3072]

    Prologue
    load coefficient, adjust coefficient offset
    load sample, adjust sample offset
    multiply
    add to running sum, check if inner loop done
    clear acc, write result, increment offset, increment write address
    decrement outer loop, rollback coef. address, check outer loop
*/

#define BLANK \
  { U1_NULL;  \
    U2_NULL;  \
```

```
    U3_NULL;  \
    DB1_NULL; \
    DB2_NULL; \
    DM_NULL;  \
    IM_NULL;  \
    C_NULL; }

    /* jump off to main */
  { U1_nop;
    U2_nop;
    U3_nop;
    DB1_NULL;
    DB2_NULL;
    DM_NULL;
    IM_NULL;
    C_jump main; }

    /*
     * this is where the coefficients should be - these will be loaded
     * later from a file - unfortunately our assembler doesn't support
     * dot-notation yet. Let's just create some blank holders
     */
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK

main:
  { U1_sub U1.R1, U1.R1, U1.R1;
    U2_nop;
    U3_nop;
    DB1_move_im 1, U1.R0;
    DB2_move_im 1, U3.R2;
    DM_NULL;
    IM_NULL;
    C_NULL; }

  { U1_addc U1.R0, 1;
    U2_nop;
```

```
    U3_nop;
    DB1_NULL;
    DB2_NULL;
    DM_NULL;
    IM_ld U3.R0, U1.R0;
    C_NULL; }

  { U1_addc U1.R0, 1;
    U2_nop;
    U3_nop;
    DB1_NULL;
    DB2_NULL;
    DM_NULL;
    IM_ld U3.R1, U1.R0;
    C_NULL; }

loopout:
  { U1_nop;
    U2_sub U2.R3, U2.R3, U2.R3;
    U3_nop;
    DB1_move_im 3, U1.R0;
    DB2_move_im 7, U1.R3;
    DM_NULL;
    IM_NULL;
    C_NULL; }

loopin:
  { U1_addc U1.R0, 1;
    U2_nop;
    U3_nop;
    DB1_NULL;
    DB2_NULL;
    DM_NULL;
    IM_ld U2.R0, U1.R0;
    C_NULL; }

  { U1_addc U1.R1, 1;
    U2_nop;
    U3_nop;
    DB1_NULL;
    DB2_NULL;
    DM_ld U2.R1, U1.R1;
    IM_NULL;
    C_NULL; }
```

```
  { U1_nop;
    U2_mul U2.R0, U2.R1, U2.R2;
    U3_nop;
    DB1_nop;
    DB2_nop;
    DM_NULL;
    IM_NULL;
    C_NULL; }

  { U1_addc U1.R3, -1;
    U2_add U2.R2, U2.R3, U2.R3;
    U3_nop;
    DB1_NULL;
    DB2_NULL;
    DM_NULL;
    IM_NULL;
    C_brnz U1.R3, loopin; }

  { U1_addc U1.R1, -7;
    U2_nop;
    U3_add U3.R0, U3.R2, U3.R0;
    DB1_NULL;
    DB2_NULL;
    DM_st U2.R3, U3.R0;
    IM_NULL;
    C_NULL; }

  { U1_addc U1.R0, -7;
    U2_nop;
    U3_add U3.R1, U3.R2, U3.R1;
    DB1_NULL;
    DB2_NULL;
    DM_NULL;
    IM_NULL;
    C_brnz U3.R1, loopout; }

halt:
  { U1_nop;
    U2_nop;
    U3_nop;
    DB1_nop;
    DB2_NULL;
    DM_NULL;
    IM_NULL;
    C_halt; }
```

```
END
```

D.1.2 Infinite Impulse Response (IIR) Filter

```
/*
 * A simple IIR filter for the example architecture
 */

/*
    Memory Map
    Writeback loc   IM[1]
    Outer Loop Cnt  IM[2]
    coefficients    IM[3]-IM[10]
    Samples         DM[0]-DM[1023]
    Output          DM[2048]-DM[3072]

    Prologue
    load coefficient, adjust coefficient offset
    load sample, adjust sample offset
    multiply
    add to running sum, check if inner loop done
    clear acc, write result, increment offset, increment write address
    decrement outer loop, rollback coef. address, check outer loop
*/

#define BLANK \
  { U1_NULL;  \
    U2_NULL;  \
    U3_NULL;  \
    DB1_NULL; \
    DB2_NULL; \
    DM_NULL;  \
    IM_NULL;  \
    C_NULL; }

    /* jump off to main */
  { U1_nop;
    U2_nop;
    U3_nop;
    DB1_NULL;
    DB2_NULL;
    DM_NULL;
    IM_NULL;
    C_jump main; }

    /*
     * this is where the coefficients should be - these will be loaded
```

```
     * later from a file - unfortunately our assembler doesn't support
     * dot-notation yet. Let's just create some blank holders
     */
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK
     BLANK

main:
  { U1_sub U1.R1, U1.R1, U1.R1;
    U2_nop;
    U3_nop;
    DB1_move_im 1, U1.R0;
    DB2_move_im 1, U3.R2;
    DM_NULL;
    IM_NULL;
    C_NULL; }

  { U1_addc U1.R0, 1;
    U2_nop;
    U3_nop;
    DB1_NULL;
    DB2_NULL;
    DM_NULL;
    IM_ld U1.R1, U1.R0;
    C_NULL; }

  { U1_addc U1.R0, 1;
    U2_nop;
    U3_nop;
    DB1_NULL;
    DB2_NULL;
    DM_NULL;
    IM_ld U3.R1, U1.R0;
    C_NULL; }

  { U1_nop;
    U2_nop;
    U3_nop;
```

```
    DB1_move_im 0, U3.R0;
    DB2_nop;
    DM_NULL;
    IM_NULL;
    C_NULL; }

loopout:
  { U1_addc U1.R1, -7;
    U2_nop;
    U3_nop;
    DB1_move_im 3, U1.R0;
    DB2_move_im 7, U1.R3;
    DM_NULL;
    IM_NULL;
    C_NULL; }

  { U1_addc U1.R0, 1;
    U2_nop;
    U3_nop;
    DB1_NULL;
    DB2_NULL;
    DM_NULL;
    IM_ld U2.R0, U1.R0;
    C_NULL; }

  { U1_nop;
    U2_nop;
    U3_add U3.R0, U3.R2, U3.R0;
    DB1_NULL;
    DB2_NULL;
    DM_ld U2.R1, U3.R0;
    IM_NULL;
    C_NULL; }

  { U1_nop;
    U2_mul U2.R0, U2.R1, U2.R3;
    U3_nop;
    DB1_nop;
    DB2_nop;
    DM_NULL;
    IM_NULL;
    C_NULL; }

loopin:
  { U1_addc U1.R0, 1;
```

```
  U2_nop;
  U3_nop;
  DB1_NULL;
  DB2_NULL;
  DM_NULL;
  IM_ld U2.R0, U1.R0;
  C_NULL; }

{ U1_addc U1.R1, 1;
  U2_nop;
  U3_nop;
  DB1_NULL;
  DB2_NULL;
  DM_ld U2.R1, U1.R1;
  IM_NULL;
  C_NULL; }

{ U1_nop;
  U2_mul U2.R0, U2.R1, U2.R2;
  U3_nop;
  DB1_nop;
  DB2_nop;
  DM_NULL;
  IM_NULL;
  C_NULL; }

{ U1_addc U1.R3, -1;
  U2_add U2.R2, U2.R3, U2.R3;
  U3_nop;
  DB1_NULL;
  DB2_NULL;
  DM_NULL;
  IM_NULL;
  C_brnz U1.R3, loopin; }

{ U1_addc U1.R1, -1;
  U2_nop;
  U3_nop;
  DB1_nop;
  DB2_nop;
  DM_NULL;
  IM_NULL;
  C_NULL; }

{ U1_addc U1.R1, 1;
```

```
    U2_nop;
    U3_nop;
    DB1_NULL;
    DB2_NULL;
    DM_st U2.R3, U1.R1;
    IM_NULL;
    C_NULL; }

  { U1_nop;
    U2_nop;
    U3_add U3.R1, U3.R2, U3.R1;
    DB1_NULL;
    DB2_NULL;
    DM_NULL;
    IM_NULL;
    C_brnz U3.R1, loopout; }

halt:
  { U1_nop;
    U2_nop;
    U3_nop;
    DB1_nop;
    DB2_NULL;
    DM_NULL;
    IM_NULL;
    C_halt; }

END
```

D.2 Applications for the SPAM VLIW-2 Architecture

These are the assembly applications for the SPAM VLIW-2 architecture Version a.

D.2.1 Finite Impulse Response (FIR) Filter

```
/*
    include files - std.asm has some basic definitions, vector.asm
    defines the exception vectors - including reset (which calls main
    if you want to change any vectors you need to redefine the
    appropriate vectore label (usually in the form V_TRAPx) to point
    to the label of your trap handler. The only exception right now
    is V_EXC1 which can be redefined as well.
    */

#include "../include/std.asm"
#include "../include/vector.asm"

#define H0 0x3b7ffde7      /* 0.003906125 */
#define H1 0x3c000000      /* 0.0078125   */
#define H2 0x3c800000      /* 0.015625    */
#define H3 0x3d000000      /* 0.03125     */
#define H4 0x3d800000      /* 0.0625      */
#define H5 0x3e000000      /* 0.125       */
#define H6 0x3e800000      /* 0.25        */
#define H7 0x3f000000      /* 0.5         */

/* start by loading the coeficients in the registers */

main:  {Control_NOP;
        ALU_IDLE;
        MAC_IDLE;
        DB_move H0, MAC.R0;
        AG1_NULL;
        AG2_NULL;
        DM1_idle;
        DM2_idle; }

       {Control_NOP;
        ALU_IDLE;
        MAC_IDLE;
        DB_move H1, MAC.R1;
        AG1_NULL;
```

```
 AG2_NULL;
 DM1_idle;
 DM2_idle; }

{Control_NOP;
 ALU_IDLE;
 MAC_IDLE;
 DB_move H2, MAC.R2;
 AG1_NULL;
 AG2_NULL;
 DM1_idle;
 DM2_idle; }

{Control_NOP;
 ALU_IDLE;
 MAC_IDLE;
 DB_move H3, MAC.R3;
 AG1_NULL;
 AG2_NULL;
 DM1_idle;
 DM2_idle; }

{Control_NOP;
 ALU_IDLE;
 MAC_IDLE;
 DB_move H4, MAC.R4;
 AG1_NULL;
 AG2_NULL;
 DM1_idle;
 DM2_idle; }

{Control_NOP;
 ALU_IDLE;
 MAC_IDLE;
 DB_move H5, MAC.R5;
 AG1_NULL;
 AG2_NULL;
 DM1_idle;
 DM2_idle; }

{Control_NOP;
 ALU_IDLE;
 MAC_IDLE;
 DB_move H6, MAC.R6;
 AG1_NULL;
```

```
         AG2_NULL;
         DM1_idle;
         DM2_idle; }

        {Control_NOP;
         ALU_IDLE;
         MAC_IDLE;
         DB_move H7, MAC.R7;
         AG1_NULL;
         AG2_NULL;
         DM1_idle;
         DM2_idle; }

/* Load the AG2 coefficients into the registers */

        {Control_NOP;
         ALU_IDLE;
         MAC_IDLE;
         DB_move -1023, ALU.R31;
         AG1_NULL;
         AG2_NULL;
         DM1_idle;
         DM2_idle; }

        {Control_NOP;
         ALU_IDLE;
         MAC_IDLE;
         DB_move ALU.R31, AG2.R2;
         AG1_idle;
         AG2_idle;
         DM1_idle;
         DM2_idle; }

        {Control_NOP;
         ALU_IDLE;
         MAC_IDLE;
         DB_move 0, ALU.R31;
         AG1_NULL;
         AG2_NULL;
         DM1_idle;
         DM2_idle; }

        {Control_NOP;
         ALU_IDLE;
         MAC_IDLE;
```

```
         DB_move ALU.R31, AG2.R0;
         AG1_idle;
         AG2_idle;
         DM1_idle;
         DM2_idle; }

        {Control_NOP;
         ALU_IDLE;
         MAC_IDLE;
         DB_move 2048, ALU.R31;
         AG1_NULL;
         AG2_NULL;
         DM1_idle;
         DM2_idle; }

        {Control_NOP;
         ALU_IDLE;
         MAC_IDLE;
         DB_move ALU.R31, AG2.R1;
         AG1_idle;
         AG2_idle;
         DM1_idle;
         DM2_idle; }

/* here goes the main loop */

/* first load the first sample and check the bound at the same time */
loop:  {Control_ldjrpc;
         ALU_IDLE;
         MAC_clr;
         DB_NULL;
         AG1_idle;
         AG2_add AG2.R0, AG2.R2, AG2.R3;
         DM1_idle;
         DM2_dir_load_m MAC.R8, AG2.R0; }

/* load the rest of the samples and mac them */

        {Control_NOP;
         ALU_IDLE;
         MAC_mac MAC.R0, MAC.R8;
         DB_NULL;
         AG1_idle;
         AG2_idle;
         DM1_idle;
```

```
 DM2_dir_load_ic MAC.R8, AG2.R0, 1; }

{Control_NOP;
 ALU_IDLE;
 MAC_mac MAC.R1, MAC.R8;
 DB_NULL;
 AG1_idle;
 AG2_idle;
 DM1_idle;
 DM2_dir_load_ic MAC.R8, AG2.R0, 2; }

{Control_NOP;
 ALU_IDLE;
 MAC_mac MAC.R2, MAC.R8;
 DB_NULL;
 AG1_idle;
 AG2_idle;
 DM1_idle;
 DM2_dir_load_ic MAC.R8, AG2.R0, 3; }

{Control_NOP;
 ALU_IDLE;
 MAC_mac MAC.R3, MAC.R8;
 DB_NULL;
 AG1_idle;
 AG2_idle;
 DM1_idle;
 DM2_dir_load_ic MAC.R8, AG2.R0, 4; }

{Control_NOP;
 ALU_IDLE;
 MAC_mac MAC.R4, MAC.R8;
 DB_NULL;
 AG1_idle;
 AG2_idle;
 DM1_idle;
 DM2_dir_load_ic MAC.R8, AG2.R0, 5; }

{Control_NOP;
 ALU_IDLE;
 MAC_mac MAC.R5, MAC.R8;
 DB_NULL;
 AG1_idle;
 AG2_idle;
 DM1_idle;
```

```
        DM2_dir_load_ic MAC.R8, AG2.R0, 6; }

       {Control_NOP;
        ALU_IDLE;
        MAC_mac MAC.R6, MAC.R8;
        DB_NULL;
        AG1_idle;
        AG2_idle;
        DM1_idle;
        DM2_dir_load_ic MAC.R8, AG2.R0, 7; }

/* write the result back and increment the pointer into the memory */

       {Control_NOP;
        ALU_IDLE;
        MAC_macw MAC.R7, MAC.R8, MAC.R8;
        DB_IDLE;
        AG1_idle;
        AG2_inc AG2.R0, AG2.R0;
        DM1_idle;
        DM2_idle; }

/* store the result back into a different portion of the memory */
/* break out of the loop if you are done */

       {Control_brcz AG2.R3, stop;
        ALU_IDLE;
        MAC_IDLE;
        DB_IDLE;
        AG1_idle;
        AG2_NULL;
        DM1_idle;
        DM2_idle; }

/* loop back otherwise */

       {Control_jump;
        ALU_IDLE;
        MAC_IDLE;
        DB_NULL;
        AG1_idle;
        AG2_inc AG2.R1, AG2.R1;
        DM1_idle;
        DM2_dir_save_m MAC.R8, AG2.R1; }
```

```
stop:   HALT_INST

END
```

D.2.2 Infinite Impulse Response (IIR) Filter

```
/*
    include files - std.asm has some basic definitions, vector.asm
    defines the exception vectors - including reset (which calls main
    if you want to change any vectors you need to redefine the
    appropriate vectore label (usually in the form V_TRAPx) to point
    to the label of your trap handler. The only exception right now
    is V_EXC1 which can be redefined as well.
    */

#include "../include/std.asm"
#include "../include/vector.asm"

#define H0 0x3f800000      /* 1.0        */
#define H1 0xbe800000      /* -0.25      */
#define H2 0xbe800000      /* -0.25      */
#define H3 0xbe800000      /* -0.25      */
#define H4 0xbe800000      /* -0.25      */
#define H5 0xbe800000      /* -0.25      */
#define H6 0xbe800000      /* -0.25      */
#define H7 0xbe800000      /* -0.25      */

/* start by loading the coeficients in the registers */

main:  {Control_NOP;
        ALU_IDLE;
        MAC_IDLE;
        DB_move H0, MAC.R0;
        AG1_NULL;
        AG2_NULL;
        DM1_idle;
        DM2_idle; }

       {Control_NOP;
        ALU_IDLE;
        MAC_IDLE;
        DB_move H1, MAC.R1;
        AG1_NULL;
        AG2_NULL;
        DM1_idle;
        DM2_idle; }

       {Control_NOP;
        ALU_IDLE;
```

```
 MAC_IDLE;
 DB_move H2, MAC.R2;
 AG1_NULL;
 AG2_NULL;
 DM1_idle;
 DM2_idle; }

{Control_NOP;
 ALU_IDLE;
 MAC_IDLE;
 DB_move H3, MAC.R3;
 AG1_NULL;
 AG2_NULL;
 DM1_idle;
 DM2_idle; }

{Control_NOP;
 ALU_IDLE;
 MAC_IDLE;
 DB_move H4, MAC.R4;
 AG1_NULL;
 AG2_NULL;
 DM1_idle;
 DM2_idle; }

{Control_NOP;
 ALU_IDLE;
 MAC_IDLE;
 DB_move H5, MAC.R5;
 AG1_NULL;
 AG2_NULL;
 DM1_idle;
 DM2_idle; }

{Control_NOP;
 ALU_IDLE;
 MAC_IDLE;
 DB_move H6, MAC.R6;
 AG1_NULL;
 AG2_NULL;
 DM1_idle;
 DM2_idle; }

{Control_NOP;
 ALU_IDLE;
```

```
        MAC_IDLE;
        DB_move H7, MAC.R7;
        AG1_NULL;
        AG2_NULL;
        DM1_idle;
        DM2_idle; }

/* Load the AG2 coefficients into the registers */

       {Control_NOP;
        ALU_IDLE;
        MAC_IDLE;
        DB_move -1023, ALU.R31;
        AG1_NULL;
        AG2_NULL;
        DM1_idle;
        DM2_idle; }

       {Control_NOP;
        ALU_IDLE;
        MAC_IDLE;
        DB_move ALU.R31, AG2.R2;
        AG1_idle;
        AG2_idle;
        DM1_idle;
        DM2_idle; }

       {Control_NOP;
        ALU_IDLE;
        MAC_IDLE;
        DB_move 0, ALU.R31;
        AG1_NULL;
        AG2_NULL;
        DM1_idle;
        DM2_idle; }

       {Control_NOP;
        ALU_IDLE;
        MAC_IDLE;
        DB_move ALU.R31, AG2.R0;
        AG1_idle;
        AG2_idle;
        DM1_idle;
        DM2_idle; }
```

```
        {Control_NOP;
         ALU_IDLE;
         MAC_IDLE;
         DB_move 7, ALU.R31;
         AG1_NULL;
         AG2_NULL;
         DM1_idle;
         DM2_idle; }

        {Control_NOP;
         ALU_IDLE;
         MAC_IDLE;
         DB_move ALU.R31, AG2.R4;
         AG1_idle;
         AG2_idle;
         DM1_idle;
         DM2_idle; }

        {Control_NOP;
         ALU_IDLE;
         MAC_IDLE;
         DB_move 2041, ALU.R31;
         AG1_NULL;
         AG2_NULL;
         DM1_idle;
         DM2_idle; }

        {Control_NOP;
         ALU_IDLE;
         MAC_IDLE;
         DB_move ALU.R31, AG2.R1;
         AG1_idle;
         AG2_idle;
         DM1_idle;
         DM2_idle; }

/* here goes the main loop */

/* first load the first sample and check the bound at the same time */
loop:  {Control_ldjrpc;
         ALU_IDLE;
         MAC_clr;
         DB_NULL;
         AG1_idle;
         AG2_add AG2.R0, AG2.R2, AG2.R3;
```

```
        DM1_idle;
        DM2_dir_load_m MAC.R8, AG2.R0; }

/* load the rest of the samples and mac them */

       {Control_NOP;
        ALU_IDLE;
        MAC_mac MAC.R0, MAC.R8;
        DB_NULL;
        AG1_idle;
        AG2_idle;
        DM1_idle;
        DM2_dir_load_ic MAC.R8, AG2.R1, 0; }

       {Control_NOP;
        ALU_IDLE;
        MAC_mac MAC.R1, MAC.R8;
        DB_NULL;
        AG1_idle;
        AG2_idle;
        DM1_idle;
        DM2_dir_load_ic MAC.R8, AG2.R1, 1; }

       {Control_NOP;
        ALU_IDLE;
        MAC_mac MAC.R2, MAC.R8;
        DB_NULL;
        AG1_idle;
        AG2_idle;
        DM1_idle;
        DM2_dir_load_ic MAC.R8, AG2.R1, 2; }

       {Control_NOP;
        ALU_IDLE;
        MAC_mac MAC.R3, MAC.R8;
        DB_NULL;
        AG1_idle;
        AG2_idle;
        DM1_idle;
        DM2_dir_load_ic MAC.R8, AG2.R1, 3; }

       {Control_NOP;
        ALU_IDLE;
        MAC_mac MAC.R4, MAC.R8;
        DB_NULL;
```

```
        AG1_idle;
        AG2_idle;
        DM1_idle;
        DM2_dir_load_ic MAC.R8, AG2.R1, 4; }

       {Control_NOP;
        ALU_IDLE;
        MAC_mac MAC.R5, MAC.R8;
        DB_NULL;
        AG1_idle;
        AG2_idle;
        DM1_idle;
        DM2_dir_load_ic MAC.R8, AG2.R1, 5; }

       {Control_NOP;
        ALU_IDLE;
        MAC_mac MAC.R6, MAC.R8;
        DB_NULL;
        AG1_idle;
        AG2_add AG2.R1,AG2.R4,AG2.R5;
        DM1_idle;
        DM2_dir_load_ic MAC.R8, AG2.R1, 6; }

/* write the result back and increment the pointer into the memory */

       {Control_NOP;
        ALU_IDLE;
        MAC_macw MAC.R7, MAC.R8, MAC.R8;
        DB_IDLE;
        AG1_idle;
        AG2_inc AG2.R0, AG2.R0;
        DM1_idle;
        DM2_idle; }

/* store the result back into a different portion of the memory */
/* break out of the loop if you are done */

       {Control_brcz AG2.R3, stop;
        ALU_IDLE;
        MAC_IDLE;
        DB_IDLE;
        AG1_idle;
        AG2_NULL;
        DM1_idle;
        DM2_idle; }
```

```
/* loop back otherwise */

       {Control_jump;
        ALU_IDLE;
        MAC_IDLE;
        DB_NULL;
        AG1_idle;
        AG2_inc AG2.R1, AG2.R1;
        DM1_idle;
        DM2_dir_save_m MAC.R8, AG2.R5; }

stop:   HALT_INST

END
```

D.3 Applications for the SPAM RISC Architecture

This is the assembly application for the SPAM RISC architecture Version a.

D.3.1 Divide-and-Conquer Array Accumulate Function

```
/*
 * int acc(a, c) {
 *   if (c == 1) return a[0];
 *   return (acc(a, c/2) + acc(a + c/2, c - c/2));
 * }
 */

#define RET R31
#define SP  R30

start:
   { movm SP, 2; }
   { stc 0, SP, SP; }
   { movm R0, 1; }
   { stc 1, SP, R0; }
   { mov R0, 40; }
   { stc 2, SP, R0; }
   { call acc; }
   { ldc 3, SP, R0; }
   { halt; }

acc:
   { ldc 2, SP, R1; }
   { subc 1, R1, R2; }
   { brnz R2, LL1; }

   { ldc 1, SP, R0; }
   { ldc 0, R0, R15; }
   { br LL2; }

LL1:
   { stc 4, SP, RET; }
   { ldc 1, SP, R0; }
   { asrc 1, R1, R2; }
   { stc 5, SP, R2; }
   { stc 7, SP, SP; }
   { addc 7, SP, SP; }
```

```
   { stc 1, SP, R0; }
   { stc 2, SP, R2; }
   { call acc; }
   { ldc 10, SP, R3; }
   { stc 6, SP, R3; }
   { ldc 1, SP, R0; }
   { ldc 2, SP, R1; }
   { ldc 5, SP, R2; }
   { add R0, R2, R4; }
   { sub R1, R2, R5; }
   { stc 7, SP, SP; }
   { addc 7, SP, SP; }
   { stc 1, SP, R4; }
   { stc 2, SP, R5; }
   { call acc; }
   { ldc 10, SP, R3; }
   { ldc 6, SP, R4; }
   { add R3, R4, R15; }
   { ldc 4, SP, RET; }

LL2:
   { stc 3, SP, R15; }
   { ldc 0, SP, SP; }
   { ret; }

END
```

Bibliography

[1] M. R. Barbacci, G. E. Barnes, R. G. Cattell, and D. P. Siewiorek. The ISPS Computer Description Language. Technical report, Dept. of EECS, CMU, Pittsburgh, PA, August 16, 1979.

[2] David Gordon Bradlee. *Retargetable Instruction Scheduling for Pipelined Processors.* PhD thesis, University of Washington, 1991.

[3] M. A. Breuer. A Random and an Algorithmic Technique for Fault Detection and Test Generation. In *IEEE Transactions on Computers*, volume C-20, pages 1366–1370, November 1971.

[4] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] F. Depuydt. *Register Optimization and Scheduling for Real-Time Digital Signal Processing Architectures.* PhD thesis, Katholieke Universiteit Leuven, October 1993.

[6] Daniel W. Engels. *Scheduling for System Synthesis: Theory and Practice.* PhD thesis, Massachusetts Institute of Technology, June 2000.

[7] D. Lanneer et al. CHESS: Retargetable Code Generation for Embedded DSP Processors. In *Code Generation for Embedded Processors*, pages 85–102. Kluwer Academic Publishers, 1995.

[8] G. Goossens et al. Integration of Medium-Throughput Signal Processing Algorithms on Flexible Instruction-Set Architectures. *Journal of VLSI Signal Processing*, 9(1):49–65, 1995.

[9] M. A. Hartoog et al. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. In *Proceedings of the* 34^{th} *Design Automation Conference*, pages 303–306, 1997.

[10] A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Sets Using nML (Extended Version). Technical report, Technische Universität Berlin and IMEC, Berlin (Germany)/Leuven (Belgium), 1995.

[11] G. Goossens, F. Catthoor, D. Lanneer, and H. De Man. Integration of Signal Processing Systems on Heterogeneous IC Architectures. In *Proceedings of the 6th International Workshop on High-Level Synthesis*, pages 16–26, November 1992.

[12] R. K. Gupta and G. De Micheli. Hardware–Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers*, pages 29–41, September 1993.

[13] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau. HMDES Version 2.0 Specification. Technical Report IMPACT-96-3, University of Illinois, Urbana, 1996.

[14] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. Technical report, MIT, 1996. (http://rle-vlsi.mit.edu/spam/pubs/ISDL-TR.html).

[15] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proceedings of the* 34^{th} *Design Automation Conference*, pages 299–302, June 1997.

[16] G. Hadjiyiannis, P. Russo, and S. Devadas. A Methodology for Accurate Performance Evaluation in Architecture Exploration. In *Proceedings of the* 36^{th} *Design Automation Conference*, pages 927–932, June 1999.

[17] G. I. 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).

[18] G. I. Hadjiyiannis, P. Russo, and S. Devadas. Automatic Architecture Exploration for Hardware/Software Codesign. In *Proceedings of the* 4^{th} *International Conference on Electronics, Circuits and Systems*, September 1999.

[19] George I. Hadjiyiannis, Silvina Z. Hanono, and Srinivas Devadas. ISDL: An instruction Set Description Language for Retargetability and Architecture Exploration. *Design Automation for Embedded Systems, To Appear.*

[20] S. Hanono and S. Devadas. Instruction Selection, Resource Allocation, and Scheduling in the Aviv Retargetable Code Generator. In *Proceedings of the* 35^{th} *Design Automation Conference*, pages 510–515, June 1998.

[21] Silvina Z. Hanono. *Aviv: A Retargetable Code Generator for Embedded Processors.* PhD thesis, Massachussetts Institute of Technology, June 1999.

[22] James C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. In *Proceedings of X IFIP International Conference on VLSI*, December 1999.

[23] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL PlayDoh Architecture Specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, 1994.

[24] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, chapter 5, pages 85–102. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[25] J. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly & Associates, Inc., 1992.

[26] Maria-Cristina Marinescu and Martin Rinard. A Synthesis Algorithm for Modular Design of Pipelined Circuits. In *Proceedings of X IFIP International Conference on VLSI*, December 1999.

[27] P. Marwedel. The MIMOLA Design System: Tools for the Design of Digital Processors. In *Proceedings of the 21th Design Automation Conference*, pages 587–593, 1984.

[28] P. Marwedel. Tree-Based Mapping of Algorithms to Predefined Structures. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design*, pages 586–593, November 1993. Extended version: Technical report 431, Lehrstuhl Informatik XII, University of Dortmund, Germany. January 1993.

[29] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors.* Kluwer Academic Publishers, Boston, Massachusetts, 1995. Proceedings of the 1994 Dagstuhl Workshop on Code Generation for Embedded Processors. ISBN 0-7923-9577-8.

[30] P. Paulin, C. Liem, T. May, and S. Sutarwala. FlexWare: A Flexible Firmware Development Environment for Embedded Systems. In *Code Generation for Embedded Processors*, chapter 4, pages 67–84. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[31] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. CodeSyn: A Retargetable Code Synthesis System. In *Proceedings of the 7th International High-Level Synthesis Workshop*, 1994.

[32] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective. *Journal of VLSI Signal Processing*, 9(1/2):23–47, January 1995.

[33] Pietro Russo. The Hgen Hardware Synthesis System. Master's thesis, Massachusetts Institute of Technology, June 1999.

[34] C. Siska. A Processor Description Language Supporting Retargetable Multi-Pipeline DSP Program Development Tools. In *Proceedings of the 1998 International Symposium on System Synthesis*, 1998.

[35] Soner Önder and Rajiv Gupta. Automatic Generation of Microarchitecture Simulators. In *Proceedings of 1998 International Conference on Computer Languages*, May 1998.

[36] S. Sutarwala, P. G. Paulin, and Y. Kumar. Insulin: An Instruction Set Simluation Environment. In *Proceedings of the 1993 Conference on Hardware Description Languages*, pages 355–362, 1993.

[37] G. Zimmermann. The MIMOLA Design System: A Computer Aided Digital Processor Design Method. In *Proceedings of the 16th Design Automation Conference*, pages 53–58, 1979.

[38] 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*, 1996.