

ABSTRACT

An Evaluation of CoWare Inc.'s Processor Designer Tool Suite
for the Design of Embedded Processors

Jonathan D. Franz, M.S.E.C.E

Advisor: Russell W. Duren, Ph.D.

The goal of this thesis is to evaluate the Processor Designer family of tools from CoWare, Inc. Processor Designer uses the *L.I.S.A. 2.0* (*Language for Instruction Set Architecture*) language. The evaluation is being performed to determine the suitability of the toolset for incorporation into a classroom environment and for the use in developing replacements for legacy processors. The main focus will be on the ease of use of the tools. This includes exploring how steep of a learning curve is involved with this new processor designer language and how well the tools have been documented. The limitations of the tools will also be explored, as far as what can and cannot be done in the language. The thesis is also intended to provide a tutorial introduction to the CoWare Inc. tool suite for future students.

An Evaluation of CoWare Inc.'s Processor Designer Tool Suite
for the Design of Embedded Processors

by

Jonathan D. Franz, B.S.

A Thesis

Approved by the Department of Electrical and Computer Engineering

Kwang Lee, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree

of
Master of Science in Electrical and Computer Engineering

Approved by the Thesis Committee

Russell W. Duren, Ph.D., Chairperson

Steven R. Eisenbarth, Ph.D.

Paul C. Grabow, Ph.D.

Accepted by the Graduate School
August 2008

J. Larry Lyon, Ph.D., Dean

Copyright © 2008 by Jonathan D. Franz

All rights reserved

TABLE OF CONTENTS

LIST OF TABLES	vi
LIST OF FIGURES	vii
ACKNOWLEDGMENTS	ix
DEDICATION	x
CHAPTER ONE	1
Introduction	1
1.1 The Problem	1
1.2 Research Objectives	3
1.3 The Strategy	5
1.4 Thesis Organization	6
CHAPTER TWO	8
Background	8
2.1 Similar Work	8
2.2 LISA	10
CHAPTER THREE	13
Processor Architectures and Instruction Sets	13
3.1 SRC	13
3.2 AYK-14	21
CHAPTER FOUR	28
CoWare, Inc. Tool Suite	28
4.1 Processor Designer	28
4.2 Processor Debugger	31
4.3 Processor Generator	33
CHAPTER FIVE	36
Processor Models and the LISA Language	36
5.1 SRC Implementation	36
5.1.1 RESOURCE Definition	36
5.1.2 OPERATION Definitions	41
5.2 Pipelined SRC	60
5.3 VLIW SRC	64
5.4 Pipelined VLIW SRC	66
5.5 AYK-14	66
5.5.1 RESOURCE Definition	66
5.5.2 OPERATION Definition	67
5.6 LISA Pitfalls	69
CHAPTER SIX	72
Verification and Implementation	72
6.1 Verification	72
6.2 Hardware Generation and System Implementation	73
6.2.1 CoWare VHDL Code Structure	73

6.2.2 SRC System Implementation	75
6.2.3 AYK-14 System Implementation	76
6.3 Implementation Results	76
6.3.1 SRC Implementation Results	76
6.3.2 Pipelined SRC Implementation Results	79
6.3.3 VLIW Implementation Results	79
6.3.4 Pipelined VLIW Implementation Results	80
6.3.5 AYK-14 Implementation Results	80
CHAPTER SEVEN	82
Conclusion	82
BIBLIOGRAPHY	322
 APPENDICES	84
APPENDIX A	85
Non-Pipelined SRC Code	85
main.lisa	85
alu.lisa	89
shift.lisa	93
branch.lisa	96
load_store.lisa	100
miscellaneous.lisa	105
immediate.lisa	107
APPENDIX B	109
Pipelined SRC Code	109
main.lisa	109
alu.lisa	114
shift.lisa	121
load_store.lisa	123
branch.lisa	127
miscellaneous.lisa	133
immediate.lisa	136
bypass.lisa	136
APPENDIX C	138
VLIW SRC Code	138
main.lisa	138
alu.lisa	143
shift.lisa	147
load_store.lisa	150
branch.lisa	154
miscellaneous.lisa	158
immediate.lisa	161
APPENDIX D	162
Pipelined VLIW SRC Code	162
main.lisa	162
alu.lisa	168
Shift.lisa	177

load_store.lisa	181
branch.lisa	187
Miscellaneous.lisa	194
Immediate.lisa	197
APPENDIX E	199
AYK-14 Code	199
main.lisa	199
arithmetic.lisa	206
logical.lisa	215
compare.lisa	217
conditionaljump.lisa	221
unconditionaljump.lisa	225
load.lisa	228
store.lisa	237
immediate.lisa	246
APPENDIX F	248
MEMORY_CFG.H	248
MEMORY_IF.H	249
APPENDIX G	250
defines.h	250
APPENDIX H	252
Assembly Source Files	252
Non-pipelined SRC	252
Pipelined SRC	255
VLIW SRC	257
Pipelined VLIW SRC	258
AYK-14	259
SRC UART Echo Program	260
AYK-14 LED Switch Program	261
APPENDIX I	263
UART Source Code	263
UART.v	263
UART_Receiver.v	268
UART_Transmitter.v	271
Clock_Prog.v	273
APPENDIX J	275
SRC Instruction Set Users Manual	275
APPENDIX K	295
AYK-14 Instruction Set Users Manual	295

LIST OF TABLES

Table 3.1: Memory Access Instructions	15
Table 3.2: ALU Instructions	16
Table 3.3: Shift Instructions	17
Table 3.4: Branch Instructions	18
Table 3.5: Miscellaneous Instructions	18
Table 3.6: Arithmetic Instructions	24
Table 3.7: Logical Instructions	24
Table 3.8: Compare Instructions	24
Table 3.9: Unconditional Jump Instructions	25
Table 3.10: Conditional Jump Instructions	25
Table 3.11: Load Instructions	26
Table 3.12: Store Instructions	26
Table 3.13: Processor Control Instructions	26
Table 6.1: SRC Implementation Results	77
Table 6.2: AYK-14 Implementation Results	81

LIST OF FIGURES

Figure 3.1: SRC Instruction formats	14
Figure 3.2: AYK-14 Instruction Formats	22
Figure 4.1: Processor Designer Screenshot	29
Figure 4.2: Processor Designer Toolbar	30
Figure 4.3: non-pipelined SRC Assembly Program	31
Figure 4.4: VLIW SRC Assembly Program	32
Figure 4.5: Processor Debugger Screenshot	34
Figure 4.6: Processor Generator Screenshot	35
Figure 5.1: Resource Definition of Memory	38
Figure 5.2: Resource Definition of Registers and Pins	39
Figure 5.3: Non-pipelined SRC Architecture	40
Figure 5.4: SRC Reset Operation	41
Figure 5.5: SRC Main Operation	42
Figure 5.6: SRC Fetch Operation	44
Figure 5.7: Memory_if.h	45
Figure 5.8: SRC Decode Operation	45
Figure 5.9: SRC Register Operation (Immediate)	46
Figure 5.10: SRC ALUI Instructions	48
Figure 5.11: SRC ALUR Instructions	50
Figure 5.12: SRC ALUU Instructions	50
Figure 5.13: SRC Shift Instructions	51
Figure 5.14: SRC ALU Functionality	53
Figure 5.15: SRC Load Instruction	54
Figure 5.16: SRC Load Relative Instruction	54
Figure 5.17: SRC Memory Interface	56
Figure 5.18: SRC Write Back Functionality	56
Figure 5.19 SRC Branch Link Instructions	57
Figure 5.20 SRC Branch Logic Functionality	58
Figure 5.21: SRC Exception Enable Instruction	59
Figure 5.22 SRC UNIT Definitions	59
Figure 5.23: Pipelined SRC Resource Declarations	60
Figure 5.24: Pipelined SRC Main Operation	61
Figure 5.25: Pipelined SRC Fetch Operation	62
Figure 5.26: Pipelined SRC ALU Functionality	63
Figure 5.27: VLIW SRC Resource Declarations	64
Figure 5.28: VLIW SRC Decode Operation	65
Figure 5.29: VLIW SRC ALUI Instructions	66
Figure 5.30: AYK-14 Resource Declarations	67
Figure 5.31: AYK-14 Fetch Operation	68
Figure 5.32: AYK-14 Decode Operation	69
Figure 6.1: SRC Hazard Detection Source File	73

Figure 6.2: CoWare HDL Code Structure
Figure 6.3: ISE Project Structure

74
76

ACKNOWLEDGMENTS

Special thanks to Dr. Russell Duren for all the help involved with the research and writing of this thesis, to Dr. Eisenbath and Dr. Grabow for all the help editing the paper.

DEDICATION

To my entire family and friends
for all the encouragement and help involved with this paper

CHAPTER ONE

Introduction

1.1 The Problem

While a large number of embedded systems incorporate commercial “off the shelf” (COTS) processors, such as an ARM-7 or PowerPC, a trend has developed towards designing application specific instruction-set processors (ASIPs). Many reasons exist for the need of an ASIP. An ASIP can give a company’s embedded system a competitive advantage. The ASIP allows the designer of the embedded system to achieve better computational efficiency by selecting an optimized set of instructions for the desired application. ASIPs also offer reusability, cost, and flexibility benefits to both the manufacturer and consumer.[1] [2] [3]

New processors may also need to be designed to replace existing legacy processors in embedded applications. There are many reasons for needing to replace an existing processor. The existing processor may no longer be in production and therefore must be remodeled. The legacy processor may also need to be upgraded to increase its usefulness. This may include adding a new set of instructions, reducing the power consumption of the processor or making the speed of the processor faster.

The design of a new embedded processor requires not only the hardware design tools but also a processor specific software toolset consisting of code generation tools as well as simulation tools.[1] The typical design flow for embedded processors has been to define a textual architecture specification of the desired processor and then to separate the design into software and hardware generation phases.[4] The hardware and software

phases are traditionally carried out concurrently. This design practice is commonly referred to as hardware/software (HW/SW) co-design. The typical processor design methodology is an iterative approach, which requires sequential refinements to be made based on software and hardware simulations.[2] This co-design process is generally carried out by large teams of highly skilled software and hardware engineers.

Partitioning and implementing a design in this manner is known to take a lot of work and time.[5]

Due to the decoupling of the hardware and software design phases, the processor being designed must be modeled at different abstraction levels. A cycle accurate model must be designed for the hardware engineers and an instruction accurate model is needed for the software engineers. The software tools, i.e. a simulator, an assembler, a linker and a debugger, are usually designed in a high level language (HLL), such as Java or C++, while the hardware design is typically carried out in a hardware descriptive language (HDL), such as Verilog or VHDL. Handwriting a complete processor-specific set of software development tools is very time consuming and highly susceptible to errors.[1]

[2] This disjointed modeling approach requires that the register-transfer-level (RTL) description of the hardware be available before the software tools can be integrated into the design and verified. RTL is an approach used in HDLs that allows the programmer to describe the synchronous behavior of digital circuits.

With this approach consistency between the software tools and the hardware can not be fully guaranteed. Furthermore, a slight change in the hardware architecture could require major changes to all of the software tools. This often makes the exploration of

different architectures for the embedded processor a lengthy and time consuming process, and many times impractical.[4]

The drawbacks of the typical co-design process have led to an emergence of many new high-level architectural description languages (ADLs). These new ADL tools attempt to close the gap between the hardware design of a processor and the software development. The new languages allow for the design of an embedded processor from a higher abstraction level than normal RTL. Many of these new tools will automatically generate the necessary RTL and software tools from the defined processor model description.

CoWare Inc.'s LISA is one of these new ADLs that enables the instruction set, behavior, and timing of the processor to be defined at a higher abstraction level. LISA is capable of generating a complete set of development tools, a compiler, an assembler, a linker, a simulator and even a debugger, from a single model description of the processor. This guarantees that the software development tools are compatible with the processor, something which can not be guaranteed through the co-design approach. This LISA approach to embedded processor design also allows for different processor architectures to be explored during the design phase, since a change in the LISA processor model will automatically generate consistent software development tools.[1]

1.2 Research Objectives

The primary objective of this thesis is to evaluate the Processor Designer family of tools from CoWare, Inc. Processor Designer uses the LISA 2.0 (Language for Instruction Set Architecture) language. This research is being performed to:

- Evaluate the suitability of the toolset for developing replacements for legacy processors.
- Evaluate the performance of the tools in terms of the quality of the VHDL code produced. The LISA generated VHDL code will be compared against handwritten VHDL code. The comparison will be based on the number of source lines of code (SLOC) required, the design time, FPGA utilization (number of slices utilized by the design), and processor speed.
- Assess the impact of different coding styles on the quality of the VHDL code produced.
- Explore the limitations of the tools, i.e. are there any limits on the type of processor architectures that can be specified or with the software tools created for a particular architecture.
- Evaluate the ease of use of the tools. This includes exploring the learning curve required to design and verify a new processor as well as to explore alternative architectures for a given processor.
- Explore and comment on the documentation provided with the toolset.
- Determine the suitability of the toolset for incorporation into a classroom environment.

Secondary objectives include:

- Develop VHDL models of a universal asynchronous receiver transmitter (UART) and other microprocessor interface components to support the evaluation of the processors in real systems.

- Develop a thesis that can be used as a tutorial introduction to the CoWare tools for future students.

1.3 The Strategy

The remainder of the thesis is spent exploring the CoWare, Inc. Processor Designer tool suite. This will be accomplished by designing multiple implementations of two processors, the Simple RISC Computer or SRC and the AYK-14. The SRC has a simple instruction set architecture and was modeled first to help learn the tools. The SRC processor was developed for instructional purposes by Vincent Heuring and Harry Jordan. The detailed design of the SRC processor is described in their textbook, *Computer Systems Design and Architecture Second Edition* [6]. The SRC design incorporates a RISC (Reduced Instruction Set Computer) instruction set architecture. Four different architectural variants of the SRC are modeled: a non-pipelined implementation, a pipelined implementation incorporating a five stage pipeline as described in *Computer Systems Design and Architecture Second Edition*, a non-pipelined VLIW implementation, and a pipelined-VLIW implementation that incorporates two five stage pipelines. The AYK-14 is a more complex processor with more instructions and a variable-length instruction word architecture. The AYK-14 represents a CISC (Complex Instruction Set Computer) architecture that is used in many military embedded systems. A subset of the AYK-14 instruction set is modeled to investigate LISA's suitability for modeling legacy processors.

The resulting processors and supporting tools, assembler, linker, simulator, and debugger, will be tested by writing and verifying multiple assembly language programs for each processor. The quality of the LISA generated VHDL code will be tested by

incorporating them into a complete embedded system. The embedded system will consist of the processor code generated by the CoWare tools, data memory (ROM), program memory (ROM), and a universal asynchronous receiver transmitter (UART). The embedded system will run on a Xilinx FPGA (Field Programmable Gate Array). The evaluation will examine code size, the amount of FPGA logic required to implement the hardware, and the clock speed of the processor.

The first of the two embedded system consisting of the SRC processor core, will be tested on the Xilinx XUP Virtex 2 Pro [7] board executing a program that communicates over a RS-232 serial link to a host PC. Furthermore, the non-pipelined SRC code generated by the CoWare tools will be compared to handwritten non-pipelined SRC code, written by Dr. Duren. The pipelined SRC code will also be compared to handwritten SRC code without complete hazard detection and data forwarding, also written by Dr. Duren. The second embedded system consisting of the AYK-14 processor core will be tested on [7], executing a program that allows the user to flip switches to light LED's.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter two gives background into other ADLs on the market, as well as the rationale for CoWare, Inc.'s tools selection. Chapter three defines the instruction set of the SRC processor and the subset of the AYK-14 instruction set that was chosen for implementation. Chapter four is a brief overview of the Processor Designer tools along with some screenshots. Chapter five outlines the LISA code developed for the SRC and AYK-14 processors. The key features of the language, along with how the language translates into actual hardware are

detailed. Chapter six describes the verification and implementation results of the SRC and AYK-14 processors. Chapter seven details the learning curve of the tools, shortcomings of the language, including poorly documented areas, and pitfalls possible while learning the LISA language. Chapter eight draws together the results of the research into a concise conclusion. The full LISA code for every processor modeled in LISA is given in the Appendices. Chapters 4 through 7, along with the code contained in the appendices, serve as a tutorial introduction to the CoWare toolset and the LISA language.

CHAPTER TWO

Background

This chapter will describe in detail the different types of Architectural Descriptive Languages (ADLs) available. An overview of the CoWare Processor Designer Tools as well as reasons CoWare's LISA was chosen is also given.

2.1 Similar Work

A need to explore architectural tradeoffs during the design phase of embedded processors has led to an increased interest in ADLs for processor design.[8] Existing processor design ADLs can be categorized into one of three categories. These categories include languages that focus on describing the processor at the architectural level (RTL or structural level), languages that abstract the design to the instruction level (behavioral level), and the languages that incorporate a joint behavioral and structural design approach.[2] [8]

MIMOLA (Machine Independent Microprogramming Language) is an example of an ADL that describes the processor at the RTL level.[2] [8] The MIMOLA language is very similar to that of Verilog or VHDL. The software tool suite uses the structural definition of the processor and therefore, usually results in poor quality compilers and assemblers.[8] Languages such as MIMOLA do not support the exploration of different processor architectures and for this reason were not considered for this research.[2] [9]

nML and ISDL (Instruction Set Description Language) are examples of ADLs that describe the design of an embedded processor at the behavioral level.[2] [8] nML

will produce an assembler based on the defined model, however the generated simulator does not support cycle accurate pipeline or VLIW architectures.[1] The nML processor model can be used with the separate CHESS compiler to generate processor specific code from a higher level language source file.[3] Furthermore, nML must be used with a separate product, GO HDL generator, to produce synthesizable RTL code [2] [10] ISDL is similar to nML and requires a separate compiler tool to generate processor specific assembly files.[3] Since these languages model the processor from a instruction set view, the ability to group common functionality together into dedicated hardware units is severely limited; which in return limits the amount of resource sharing that can occur.[2] For these reasons, behavioral level ADLs were not further considered.

Many newer ADLs are available that describe the embedded processor in a combined behavioral and structural manner. Examples of such ADL tools include EXPRESSION, Xtensa by Tensilica, PICO (Processor In Chip Out) by HP Labs and CoWare Inc.’s LISA.[2] Xtensa is built on a predefined RISC core and is limited in the architectures that it can model.[11] Similarly, PICO is based on a set of predefined hardware components and is also limited in its ability to model arbitrary processor architectures.[1]

EXPRESSION and LISA are more flexible ADLs and allow arbitrary processor architectures and their memories to be designed and simulated. Both languages provide automatic generation of a complete tool suite and RTL code generation from the model description.[1] [5] [8] EXPRESSION is a public-domain product that is readily available at <http://www.ics.uci.edu/~expression/index.htm>. No results have been published on the efficiency of the generated tool suite or RTL code based on an EXPRESSION model.

CoWare's LISA tool suite is the leading commercially supported ADL. CoWare is the only such toolset that has both UNIX and Windows versions, unlike EXPRESSION which requires a Sun Sparcstation.[2] [8] For this reason, the author chose LISA for research purposes in hope that the tools would be easier to install and learn.

2.2 LISA

The CoWare Inc. LISA language allows for the joint modeling of hardware and software of an embedded processor from within one design environment, Processor Designer. The CoWare Inc. tool suite generates the necessary software tools, which include an assembler, a disassembler, a linker, an instruction set simulator, a debugger, and a C compiler, as well as the RTL code for the processor; all from a single source code model of the processor. The CoWare tool suite also allows for the optional generation of an assembly language user's manual. This design approach guarantees that the software tools are completely compatible with the hardware. Furthermore, CoWare modeling of embedded processors provides an environment for evaluating the impact that different hardware architectures have on the instruction set without having to wait for the software tools to be rewritten, as they are generated automatically from the model description. These factors contributed to the selection CoWare Inc.'s LISA over the other ADLs available on the market.

LISA can be used to model any processor that is defined by an instruction set, such as an SRC or a DSP processor. The LISA tool suite can also be used to develop new application specific processors, study the effect of different computer architectures on an instruction set, as well as develop replacements for legacy processors. The LISA

language allows for the easy representation of pipelined (cycle-accurate) and VLIW processors.

The LISA tool suite includes Processor Designer, Processor Debugger, Processor Generator, and a C Compiler. The tools that are explored in this thesis are Processor Designer, Processor Debugger, and Processor Generator. The C Compiler required a separate tool that was not purchased, and therefore will not be mentioned further in this thesis. All the tools are integrated into Processor Designer so that each step can be launched from within a single development environment.

The full description of the hardware and instruction set of the processor can be developed with Processor Designer. This combining of hardware and software design for an embedded processor greatly reduces the complexity and time of modeling. LISA can be used to describe the instructional hierarchy, which lends itself to easy addition of instructions to an already defined design. The behavior of each instruction within LISA is coded in ANSI-C. This eliminates the need for previous knowledge of an HDL language in order to use Processor Designer. Processor Designer generates an assembler, linker, disassembler, an instruction set simulator, and a debugger, based on the LISA design description. Along with these tools, Processor Designer can generate an instruction set users manual. The instruction set user's manual lists the complete instruction set in the architecture along with syntax and a short description of how each instruction is implemented.

The LISA language model can be tested and debugged using the Processor Debugger tool, before the HDL code is generated. Within the debugger the user has the ability to view all the hardware resources, including registers and memories, as the

assembly code is executed. The debugger also keeps track of statistical information about the processor, such as the percentage of time a pipeline stage spent executing on data or how many times a block of assembly is run (keeps iteration counters on every line of the assembly code).

Once the LISA model is verified with Processor Designer, HDL code can be generated with the Processor Generator tool. The Processor Generator tool allows for generated HDL code in either Verilog or VHDL. Options are also given for the optimization of the generated HDL code; which include the degree of resource sharing between functional units.

CHAPTER THREE

Processor Architectures and Instruction Sets

This chapter will describe in detail the instruction set for the SRC architecture as well as the different hardware implementations, non-pipelined, pipelined, VLIW, and pipelined VLIW. A detailed description of the subset of the AYK-14 instructions and the hardware for the processor is also described.

3.1 SRC

In order to learn the LISA language and tool suite the SRC, Simple Reduced Instruction Set Computer, as defined in *Computer Systems Design and Architecture Second Edition*, was the first architecture that was modeled. The Instruction Set Architecture (ISA) describes the programmer-accessible registers, the external memory and the complete set of instructions, known as the Instruction Set. The programmer-accessible registers included 32 general-purpose 32-bit registers, r0 through r31, a 32-bit instruction register, IR, and a 32-bit program counter, PC. The external memory is organized as 4 GBytes of byte-addressable memory. Instruction and data words are 32-bits long. They are stored in memory as 4 bytes in big-endian order on even (modulus-4) boundaries.

The instruction set is composed of 28 32-bit instructions. Within the instruction word, the first five bits, bits 31 down to 27, define the opcode. The 28 instructions come in eight different instruction formats, depicted in Figure 3.1, and are separated into five

main categories: memory access instructions, arithmetic and logic instructions, shift instructions, branch instructions, and miscellaneous/interrupt instructions.[6]

The SRC processor uses a load-store architecture. In the SRC instruction set there are three different types of memory access instructions: loads, stores, and load address instructions. The load instructions access main memory at a calculated address and load a value from data memory into the general purpose register, ra. There are two types of load instructions, load (*ld*) and load relative (*ldr*).

1)	31...27	26...22	21...17	16	...	0	
	Op	ra	rb		c2		
2)	31...27	26...22	21	...	0		
	Op	ra		c1			
3)	31...27	26...22	21...17	16...12	11	...	0
	Op	ra	unused	rc		unused	
4)	31...27	26...22	21...17	16...12	11	...	3 2..0
	Op	unused	rb	rc	(c3)unused	cond	
5)	31...27	26...22	21...17	16...12	11	...	3 2..0
	Op	ra	rb	rc	(c3)unused	Cond	
6)	31...27	26...22	21...17	16...12	11	...	0
	Op	ra	rb	rc		unused	
7a)	31...27	26...22	21...17	16	...	5 4 ... 0	
	Op	ra	rb		(c3)unused	count	
7b)	31...27	26...22	21...17	16	...	5 4 ... 0	
	Op	ra	rb		(c3)unused	count	
8)	31...27	26	...			0	
	Op		Unused				

Figure 3.1: SRC Instruction formats

The Store instructions store a value from one of the general purpose registers, register ra, into main memory at a calculated address. Store (*st*) and store relative (*str*) are the two types of store instructions. The load address instructions are the only instructions in this group that do not access main memory, but rather load an address into

a general purpose register, ra. The load address instructions consist of a load address (*la*) and a load address relative (*lar*). [6]

The *ld*, *st*, and *la* instructions can work in either a direct addressing mode or indexing mode, sometimes referred to as displacement addressing. For this reason they can be expressed with two different syntaxes to the assembler. When operand rb is equal to zero, the instructions perform direct addressing, where the address is calculated based on the c2 immediate field. When rb is non-zero, the address is calculated based on the immediate value of c2 plus the contents of the general purpose register rb (indexing mode). The relative addressing instructions, *ldr*, *str*, and *lar*, access main memory relative to the immediate field c1 plus the program counter. Table 3.1 shows the memory access instructions along with their syntaxes and opcodes.[6]

Table 3.1: Memory Access Instructions

Instruction	Syntax	Opcode
Load	ld ra,c2	1
	ld ra,c2(rb)	
Load Relative	ldr ra,c1	2
Store	st ra,c2	3
	st ra,c2(rb)	
Store Relative	str ra,c1	4
Load Address	la ra,c2	5
Load Address Relative	la ra,c2(rb)	
	lar ra,c1	6

Within the set of arithmetic and logic (ALU) instructions there are three subsets, the ALU unary, ALU immediate, and ALU register instructions. The ALU unary instructions require only one operand, rc, that is stored in one of the general purpose registers. The ALU unary instructions consist of a negate (*neg*) and a not (*not*). The ALU immediate instructions contain one operand that is stored in a general purpose register, rb, and an immediate operand, c2, that is part of the instruction word. The ALU

immediate instructions include addition (*addi*), bitwise logical or (*ori*), and bitwise logical and (*andi*). Finally the ALU register instructions take two operands, rb and rc, both stored within the general purpose register file. Addition (*add*), subtraction (*sub*), bitwise logical or (*or*), and bitwise logical and (*and*) comprise this subset of ALU instructions. All of the ALU instructions store the results back into the general purpose register file at the location specified by operand ra. The full set of ALU instructions along with their syntax and opcode is given in Table 3.2.[6]

Table 3.2: ALU Instructions

Instruction	Syntax	Opcode
Negate	neg ra,rc	15
Not	not ra,rc	24
Add Immediate	addi ra,rb,c2	13
Add	add ra,rb,rc	12
Subtraction	sub ra,rb,rc	14
Or	or ra,rb,rc	22
Or Immediate	ori ra,rb,c2	23
And	and ra,rb,rc	20
And Immediate	andi ra,rb,c2	21

There are four defined shift instructions for the SRC architecture, a shift right (*shr*), shift right arithmetic (*shra*), shift left (*shl*), and shift left circular (*shc*). The shift instructions have two different syntaxes depending on their use. These instructions shift the value in the register file specified by operand rb by the number of bits equal to an immediate value, count, or by the amount in the register file, expressed by operand rc (count is zero in this case). The shift instructions store the resulting value into the register file at the location given by operand ra. The shift instructions, syntaxes, and opcodes are defined below in Table 3.3.[6]

Table 3.3: Shift Instructions

Instruction	Syntax	Opcode
Shift Right	shr ra,rb,c3 shr ra,rb,rc	26
Shift Right Arithmetic	shra ra,rb,c3 shra ra,rb,rc	27
Shift Left	shl ra,rb,c3 shl ra,rb,rc	28
Shift Circular	shc ra,rb,c3 shl ra,rb,rc	29

Branch instructions within the SRC come in two varieties, the branch (*br*), and the branch and link (*brl*). Branch instructions are used to control the execution flow of a program. Branch and link behaves just as the branch instruction with the added feature of storing the current program counter into a general purpose register. The program counter will always be stored into the general purpose register, specified by operand *ra*, on a branch and link even if the branch is not taken. The branches themselves can be either unconditional or conditional depending on a 3-bit condition code, bits 2 down to 0, in the instruction word. The conditional branches test the contents of a general purpose register, specified by the *rc* field, to decide whether the branch should be taken. The *rb* operand specifies where to jump on a branch. The branch instructions, conditional code (*con*), syntax, and opcode for the variety of branches is shown in Table 3.4.[6]

The miscellaneous group of instructions is comprised of the *nop* (*nop*), *stop* (*stop*), and the interrupt instructions. The *nop* instruction is a do nothing instruction that acts as a place filler in the assembly code. The *stop* instruction is defined to halt the execution of the SRC and was added primarily for debugging of code. There are five interrupt instructions: exception enable (*een*), exception disable (*edi*), return from interrupt (*rfi*), save interrupt information (*svi*), and restore interrupt information (*ri*). The exception enable and exception disable instructions allow the SRC to either process

interrupts, een, or ignore them, edi. Return from interrupt enables the SRC to restore the program counter to the instruction location just before the interrupt was serviced. The save interrupt information and restore interrupt information instructions provide a way to save the interrupt information vector and interrupt program counter into the general purpose registers, at locations ra and rb respectively, and restore them at a later time.

Table 3.5 lists the miscellaneous instructions along with their syntaxes and opcodes.[6]

Table 3.4: Branch Instructions

Instruction	Syntax	Condition	Opcode
Branch Never	Brnv	0	8
Branch Always	br rb	1	8
Branch Zero	brzr rb,rc	2	8
Branch Non-zero	brnz rb,rc	3	8
Branch Positive	brpl rb,rc	4	8
Branch Negative	brim rb,rc	5	8
Branch Link Never	brlnv ra	0	9
Branch Link Always	brl ra,rb	1	9
Branch Link Zero	brlzsra,rb,rc	2	9
Branch Link Non-zero	brlnz ra,rb,rc	3	9
Branch Link Positive	brlpl ra,rb,rc	4	9
Branch Link Negative	brlmi ra,rb,rc	5	9

Table 3.5: Miscellaneous Instructions

Instruction	Syntax	Opcode
Nop	nop	0
Stop	stop	31
Exception Enable	een	10
Exception Disable	edi	11
Return From Interrupt	rfi	30
Save Interrupt Info	svi ra,rb	16
Restore Interrupt Info	ri ra,rb	17

The authors of *Computer Systems Design and Architecture Second Edition* define multiple hardware implementations of the SRC. The book defines and models a non-pipelined SRC, a pipelined SRC, and a pipelined VLIW SRC. Each of these will be modeled using the LISA language. In addition, a non-pipelined Very Long Instruction Word (VLIW) SRC will be developed. All four architectures contain a general-purpose

register array comprised of 32 32-bit registers, a 32-bit instruction register (IR) (64-bit in the VLIW architectures), a 32-bit program counter (PC), and a memory interface unit. All three models also contain an Arithmetic and Logic Unit (ALU), a shifter (included inside the ALU), and conditional code logic for branch detection. All models of the SRC are designed with interrupt handling capabilities.[6]

The non-pipelined SRC is the easiest model to conceptualize and code. In LISA both the non-pipelined SRC and the non-pipelined VLIW SRC are actually modeled using pipelines with one stage. These models allow one instruction to be executed to completion before the next one begins. Non-pipelined models will generally have a slower clock speed than the pipelined models.[6]

The pipelined version, on the other hand, increases the throughput of instructions and supports a faster system clock. This is accomplished by breaking up the execution of the instructions into stages and allowing each stage to work on part of an instruction at any given time. *Computer Systems Design and Architecture Second Edition* defines the pipeline as being broken up into five stages: a fetch (FE) stage, decode (DC) stage, execute (EX) stage, memory access (MEM) stage, and a write back (WB) stage. Since an instruction must travel through five stages this does lead to startup latency, however, once the SRC has been running for five cycles the execution rate is greater than that of the non-pipelined SRC model.[6]

The pipelined version also introduces some new complexities into the hardware design of the SRC. Now that multiple instructions are being executed simultaneously, dependencies may arise between the instructions in the pipeline. The only type of hazard that is of concern with a single pipeline SRC is the Read after Write (RAW) hazard.

RAW hazards are caused by an instruction in the pipeline trying to read from the general purpose register file or data memory at a location that an instruction earlier has not yet written. This results in the instruction reading the wrong (old) value. RAW data hazards can be accounted for by forwarding data in the pipeline to the instruction that needs it, if the data is ready to be forwarded, or by stalling the pipeline until the data becomes available and then forwarding the correct value. RAW data hazards require the introduction of hazard detection and data forwarding logic into the hardware design.[6]

Another technique to increase the SRC system performance is to fetch two 32-bit instructions per 64-bit instruction word and execute them in parallel pipelines. This is known as a VLIW architecture. In a VLIW architecture the compiler is responsible for scheduling the instructions that can be executed in parallel. As defined in [5], only certain instructions can execute in each of the two pipelines. All the memory access instructions that go to main memory, *ld*, *ldr*, *st*, and *str*, and all the interrupt instructions must be executed in pipeline 1. Pipeline 2 is responsible for the shift and the branch instructions. Both pipelines are able to execute ALU instructions, the load address instructions, *nop*, and *stop*.[6]

To further increase the instructions throughput, the pipelines of the VLIW architecture can also contain multiple stages. Allowing two 32-bit instructions to execute in parallel requires a 64-bit instruction register to hold the two instruction words, as well as some very complex data hazard detection and data forwarding. As was the case in the pipelined version, the VLIW version must deal with RAW hazards. However, it must also handle Write after Write (WAW) and Write after Read (WAR) hazards. In addition

hazards have to be caught in the individual pipelines as well as between pipelines and the data needs to be forwarded or stalled when necessary.[6]

3.2 AYK-14

The AYK-14 processor is a Complex Instruction Set Computer (CISC) processor and was modeled using Processor Designer to test how the tools handle a more complex instruction set. The AYK-14 processor was developed by General Dynamics Advanced Information Systems in 1978. It is used by the NAVY as an airborne processor in the F/A-18, the AV-8B, and the EA-6B. The AYK-14 incorporates a Variable Length Instruction Word (VLIW) design. The instructions may be either 16-bits or 32-bits wide. The 32-bit wide instructions have a 16-bit instruction and an additional 16-bit address or arithmetic constant associated with them depending on the instruction format. The AYK-14 is capable of operating on 8-bit, 16-bit, or 32-bit data.[12] [13] [14]

The AYK-14 instructions are divided by type into 12 main categories: Load, Store, Arithmetic, Logical, Shift, Compare, Conditional Jumps, Unconditional Jumps, Miscellaneous, Processor Control, Software, and IOC. The instructions are further broken up into six instruction word formats depending on a format designator, RL (Register – Literal), RR (Register – Register), RI (Register – Indirect Memory) Type-I and Type-II, RK (Register – Constant), and RX (Register – Memory with or without indexing), depicted in Figure 3.2 and described below in order. Due to the sheer number of instructions native to the AYK-14, a minimal set that would give a fully operational processor was chosen. Each of the six instruction word formats and seven of the instruction types are represented.[14]

Type RL instructions are 16-bits long. They perform operations involving one or two general purpose registers and a 4-bit unsigned literal constant. RL instructions combine the *OP CODE* field and the *f* field to specify the instruction to be executed. The *a* operand selects the register and the *m* operand contains the 4-bit unsigned literal constant. RL instructions are constrained to OP codes 60 through 63.[14]

RL)	15 ... 10 9 8 7 ... 4 3 ... 0
	OP CODE f a m
RR, RI-II)	15 ... 10 9 8 7 ... 4 3 ... 0
	OP CODE f a m
RI-I)	15 ... 10 9 8 7 ... 0
	OP CODE f d
RK, RX)	15 ... 10 9 8 7 ... 4 3 ... 0
	OP CODE f a m
	Y

Figure 3.2: AYK-14 Instruction Formats

Type RR instructions are also 16-bits long. They perform operations that involve only general purpose registers, no memory interface or literal constants are used with this format. The *a* operand selects one register and the *m* operand selects the other. The *f* field contains a binary 00 in this instruction class. A few of the unary instructions of this format share the same *OP CODE* value and are distinguished by the four-bit value in the *m* field.[14]

RI Type-I instructions and Type-II are 16-bit wide instructions. The *f* field for all RI instructions contains a binary 01. Type-I instructions are responsible for handling local jumps within the AYK-14. The *d* field contains an 8-bit two's compliment value that is added to the program counter to determine the new fetch address. Type-II instructions are 16-bits in length and are responsible for operations accessing a general-

purpose register and main memory. The a and m designators select register values, and the m register value is used as the index into main memory.[14]

RK instructions are 32-bits in length. RK instructions use a register value and a main memory reference. The a field selects a general-purpose register value. The m and y fields combine to form a 16-bit memory reference operand, Y . When m is equal to zero, the Y operand is equal to y , else the Y operand is equal to the sum of the y field plus the contents of the general-purpose register referenced by the m designator. All RK instructions can have two separate syntaxes, one with m and y explicitly listed, index addressing, and one where m is left off, direct addressing. The f field contains a binary 10 for instructions of this format.[14]

RX instructions are 32-bit wide instructions that perform either word (16-bit) or byte (8-bit) operations using a register value and a main memory reference. The a operand selects a general-purpose register value and the m and y fields combine in the same manner described above to generate a memory reference operand, Y . All RX instructions, like the RK, can have two separate syntaxes, one with m and y explicitly listed, index addressing, and one where m is absent, direct addressing. The f field for this format contains a binary 11.[14]

A small subset of arithmetic instructions was chosen to model with the AYK-14. These instructions include two different additions and four shifts. The complete list of arithmetic instructions used is given in Table 3.6 below. Notice that the unary instructions listed, *ones compliment*, *twos compliment*, *increase by one*, *decrease by one*, and *make positive*, share the same opcode value of two. These operations are distinguished by their m operand values. Most the RL format instructions listed share a

common opcode value and are distinguished by their *f* field. The shift instructions include two 16-bit shifts, *algebraic left single shift* and *logical right single shift*, as well as two 32-bit shifts, *algebraic left double shift* and *algebraic right double shift*.[14]

Table 3.6: Arithmetic Instructions

Instruction	Syntax	Opcode	Format
Add (Index)	A	22	RX
Increase by 1	IROR	2	RR
Decrease by 1	DROR	2	RR
Add (Register)	AR	22	RR
Ones Compliment	OCR	2	RR
Twos Compliment	TCR	2	RR
Make Positive	PR	2	RR
Subtract Literal	LSU	62	RL
Add Literal	LA	62	RL
Algebraic Left Double Shift	LALD	61	RL
Algebraic Left Single Shift	LALS	61	RL
Algebraic Right Double Shift	LARD	60	RL
Logic Right Single Shift	LLRS	60	RL

Two logical instructions were chosen from the AYK-14 set of instructions, *and* and *or*. Table 3.7 below lists these instructions along with their syntax, opcodes, and instruction format. Four compare instructions were chosen from the AYK-14 instruction set, *compare constant*, *compare*, *compare literal*, and *compare bit to zero*, depicted in Table 3.8.[14]

Table 3.7: Logical Instructions

Instruction	Syntax	Opcode	Format
And (Constant)	ANDK	30	RK
Or (Register)	ORR	31	RR

Table 3.8: Compare Instructions

Instruction	Syntax	Opcode	Format
Compare Constant	CK	24	RK
Compare	C	24	RX
Compare Literal	LC	63	RL
Compare Bit to Zero	CBR	7	RR

The AYK-14 has two different types of control flow instructions, unconditional jumps, and conditional jumps, depicted in Tables 3.9 and 3.10 below respectively. The jump address will always be taken in the case of the unconditional jumps. With the conditional jumps, a bit in the status register is compared to zero and a decision to take the jump is made depending on the result of the compare and the instruction being executed.[14]

Table 3.9: Unconditional Jump Instructions

Instruction	Syntax	Opcode	Format
Jump, Link Register	JLR	42	RK
Jump, Link Register	JLR	42	RX
Local Jump	LJ	40	RI-I

Table 3.10: Conditional Jump Instructions

Instruction	Syntax	Opcode	Format
Index Jump	XJ	41	RK
Local Jump Equal	LJE	44	RI-I
Local Jump Not Equal	LJNE	45	RI-I
Local Jump Greater or Equal	LJGE	46	RI-I
Local Jump Less	LJLS	47	RI-I

The load instructions read a value, either from memory or from an operand within the instruction itself and loads it into one of the 32 general-purpose registers. The particular set of load instructions chosen from the AYK-14 class of instructions are shown below in Table 3.11. The subset of load instructions is made up of 16-bit loads as well as one 32-bit load, *Load Double*.[14]

Store instructions are defined as those that store a value into the main memory. The subset of store instructions chosen from the AYK-14 instruction set is given in Table 3.12 below. Note that the subset of store instructions is capable of storing 8-bit data, *byte store*, 16-bit data as well as 32-bit data, *store double*. [14]

Table 3.11: Load Instructions

Instruction	Syntax	Opcode	Format
Load Constant	LK	1	RK
Load Index	L	1	RX
Load Double	LD	2	RX
Load Indirect	LI	1	RI-II
Load Literal	LL	63	RL
Load Register	LR	1	RR
Set Bit Register	SBR	5	RR

Table 3.12: Store Instructions

Instruction	Syntax	Opcode	Format
Store Index	S	11	RX
Store Double	SD	12	RX
Store Zeros	SZ	17	RX
Byte Store	BS	10	RX
Store and Index by 1	SXI	15	RI-II
Store Indirect	SI	11	RI-II

Three processor control instructions were chosen from the AYK-14 instruction set. These instructions are described in Table 3.13 below. Two of the instructions act upon Status Register 1 of the processor, *Load Status Register 1* and *Store Status Register 1*. The other instruction loads an address directly into the program fetch, register.[14]

Table 3.13: Processor Control Instructions

Instruction	Syntax	Opcode	Format
Load Status Register 1	LSOR	3	RR
Load P Register	LPR	3	RR
Store Status Register 1	SSOR	3	RR

The AYK-14 instruction set was defined to operate on a non-pipelined architecture. The AYK-14 has two 16-bit status registers, *Status Register 1* and *Status Register 2*. Based on the subset of instructions chosen, the only status register that is of concern is Status Register 1. Within Status Register 1 the control bits that are of importance to the subset of instructions are the General Purpose Stack Designator

(GPSD) bit 14, Carry Designator (CD) bit 11, Overflow Designator (OD) bit 10, and Condition Code (CC) bits 9 and 8.[14]

The Overflow Designator, Carry Designator and Condition Codes are set (or cleared) whenever a value in the general-purpose register file is manipulated. These bits are then used to determine if a conditional jump is to be taken. The AYK-14 processor contains 32 16-bit general purpose registers broken up into two 16 16-bit general-purpose register stacks. The particular register stack is chosen based on the GPSD bit of Status Register 1. The interrupt handling features of the AYK-14 processor were not modeled in LISA.[14]

CHAPTER FOUR

CoWare, Inc. Tool Suite

This chapter will provide an overview of the CoWare Processor Designer software. The first section details Processor Designer. It will describe the main features of the design environment. The following section will describe the Processor Debugger. The Processor Generator will be the last CoWare Processor Designer tool described.

4.1 Processor Designer

The entire CoWare Inc.'s tool suite is integrated into the Processor Designer application. This allows the entire modeling process to be run from within a single design environment. Processor Designer is the heart of the CoWare tool suite. It is the environment in which the embedded processor is described using the LISA language. A screenshot of Processor Designer is given in Figure 4.1. Note three main sections in the figure. The left side pane, labeled Project Files, shows all the LISA files present in the current design, as well as any C header files. The center pane is where the current file that is being edited is displayed. It is here that the LISA code is written/edited. The lower pane is the area in which all error messages are reported when the code is compiled. The lower pane is also responsible for displaying the results of a search within the current project.[15]

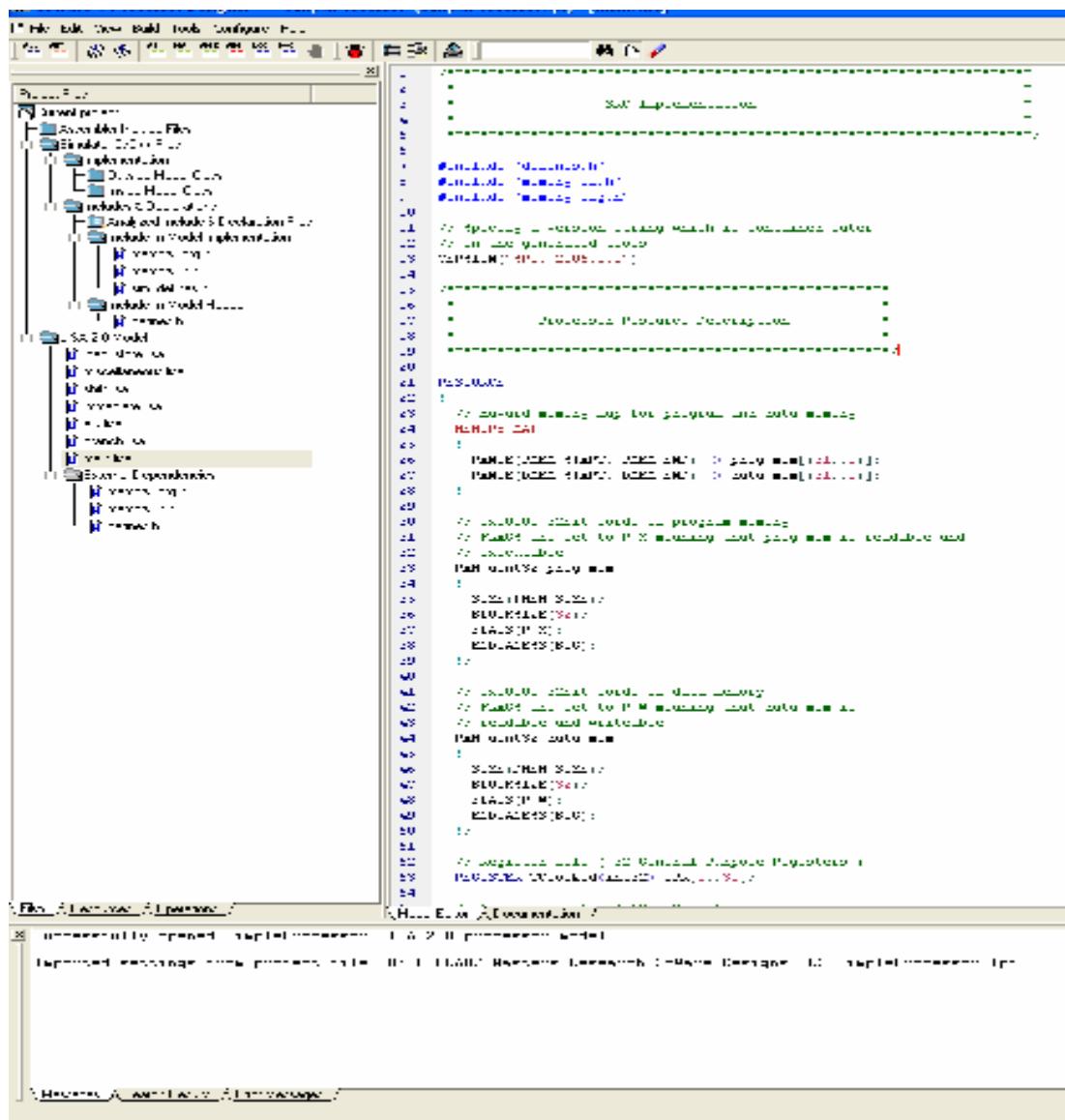


Figure 4.1: Processor Designer Screenshot

The toolbar that runs along the top of the Processor Designer development environment has all the menus and options that are needed to launch the other tools in the CoWare Inc. tool suite, see Figure 4.2. The ladybug icon launches the Processor Debugger. The Processor Debugger enables the LISA code to be fully tested before any actual hardware is generated. The two input *And* gate icon is responsible for launching

the Processor Generator, which is responsible for turning the LISA language code into RTL logic, either Verilog or VHDL.[15]

The toolbar also contains action icons for compiling the project. The entire project can be compiled at once. Compiling the LISA model results in the generation of a processor specific assembler, linker, disassembler, instruction set simulator, and debugger. The processor specific software tools can also be generated independently of each other. There is also an option for generating a documentation guide for the instruction set. The white search box will search all files located in the current project for a keyword or words.[15]

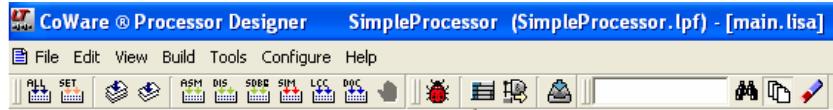


Figure 4.2: Processor Designer Toolbar

The assembly language program for the current embedded processor can be written/edited within the Processor Designer application. Figure 4.3 gives an example of an assembly program written for the non-pipelined SRC. The assembly program file is saved as app.asm and processed by the embedded processor specific generated assembler to create a binary image. This binary image file can then be executed by the debugger or the RTL generated hardware. The assembler has several processing options that can be selected via parameters. The .bend parameter tells the assembler to store the data in the current section in a big-endian format. Similarly, the .lend alerts the assembler that the data should be stored in little-endian format. The global _start alerts the assembler of the entry point into the assembly code.

```

// begin data
.data
.bend
// begin program code
.text
.bend
.global _start
.bend
_start:
    la r1,3(r0)
    st r1,0
    la r1,4
    st r1,0(r1)
    la r1,5(r0)
    st r1,4(r0)
    la r1,7(r0)
    st r1,8(r0)
    la r1,11(r0)
    st r1,12(r0)
    la r1,8(r0)
    ld r2,1(r0)
    ld r6,0(r0)
.end

```

Figure 4.3: non-pipelined SRC Assembly Program

When Very Long Instruction Word (VLIW) architectures are implemented, the assembly code must be modified. Figure 4.4 below shows an example assembly program written for the VLIW SRC architecture. Notice in this assembly program example two instructions are specified per line, separated by a ‘|’ character. It is up to the programmer to define the correct instructions to be executed in the correct pipelines. If the programmer assigns an instruction to the incorrect pipeline the assembler will error out during assembly.[15] [16]

4.2 Processor Debugger

The Processor Debugger can be launched from within Processor Designer application as mentioned earlier. The Debugger allows for the exhaustive testing of the LISA code before any RTL code is ever generated. The LISA code must be compiled before the Processor Debugger can be run. When the project is compiled an assembler

for the current design is created. The assembler can then be used to generate an .out file that is used by the debugger. A view of the Processor Debugger is given in Figure 4.5 below.[17]

```
// begin data
.data
.bend
// begin program code
.text
.bend
;
.global _start
.bend
;
_start:
    addi r1,r1,3      | addi r0,r0,1
    nop               | add r2,r1,r0
    la r5,0xff        | nop
    st r2,0           | nop
    ld r3,0           | la r4,0xff
    een              | shr r6,r4,5
.end
```

Figure 4.4: VLIW SRC Assembly Program

There are many configurable options for the debugger. Below in the image, all the clocked resources are shown on the right-hand side. These include all the embedded processor's general-purpose registers, program counter, etc. The processor's memory area is given in the lower left. The memory area contains the values stored in the processors program memory (ROM) and data memory (RAM). These resources can be monitored from within the debugger. When debugging a pipelined architecture the pipeline registers can be monitored through Processor Debugger. The Processor Debugger permits the user to gather statistical information about the current program. Information can be gathered about how often an instruction is executed, whether or not all the instructions are being utilized, and how often a pipeline stage is active during the current programs execution.[17]

The code that is currently executed by the debugger is depicted in the center box of the Processor Debugger. This area shows an arrow pointing to the current executing instruction. The toolbar on the left of the Processor Debugger environment displays commands that can be used to step through the code. The code can be executed in its entirety or run to a breakpoint. The code may be single-stepped one instruction at a time by using the arrow icon pointing into the braces. This is a very useful feature when debugging certain instructions. The Processor Debugger also comes with a micro-step feature, which can be turned on or off within the debugger by selecting the Microstep On/Off button. The micro-step feature is activated using the micro-step button. It permits the user to execute one instruction at a time while being able to see exactly what sections of the LISA code affect the current instruction.[17]

4.3 Processor Generator

Once the LISA code has been fully tested and verified the RTL generator can be launched. The RTL generator is launched from within Processor Designer application in the manner described above. Figure 4.6 below shows a screenshot of the Processor Generator. The first time the Processor Generator is launched a new configuration file must be added. There may be many different configurations that may be added at this point. Each configuration must define a memory interface, which includes defining the number of read and write ports for each memory module defined. There are also multiple design tradeoffs/options that can be selected. Along with these configurable options, the desired HDL language or languages must be selected. The code can be generated in either Verilog or VHDL.[18]

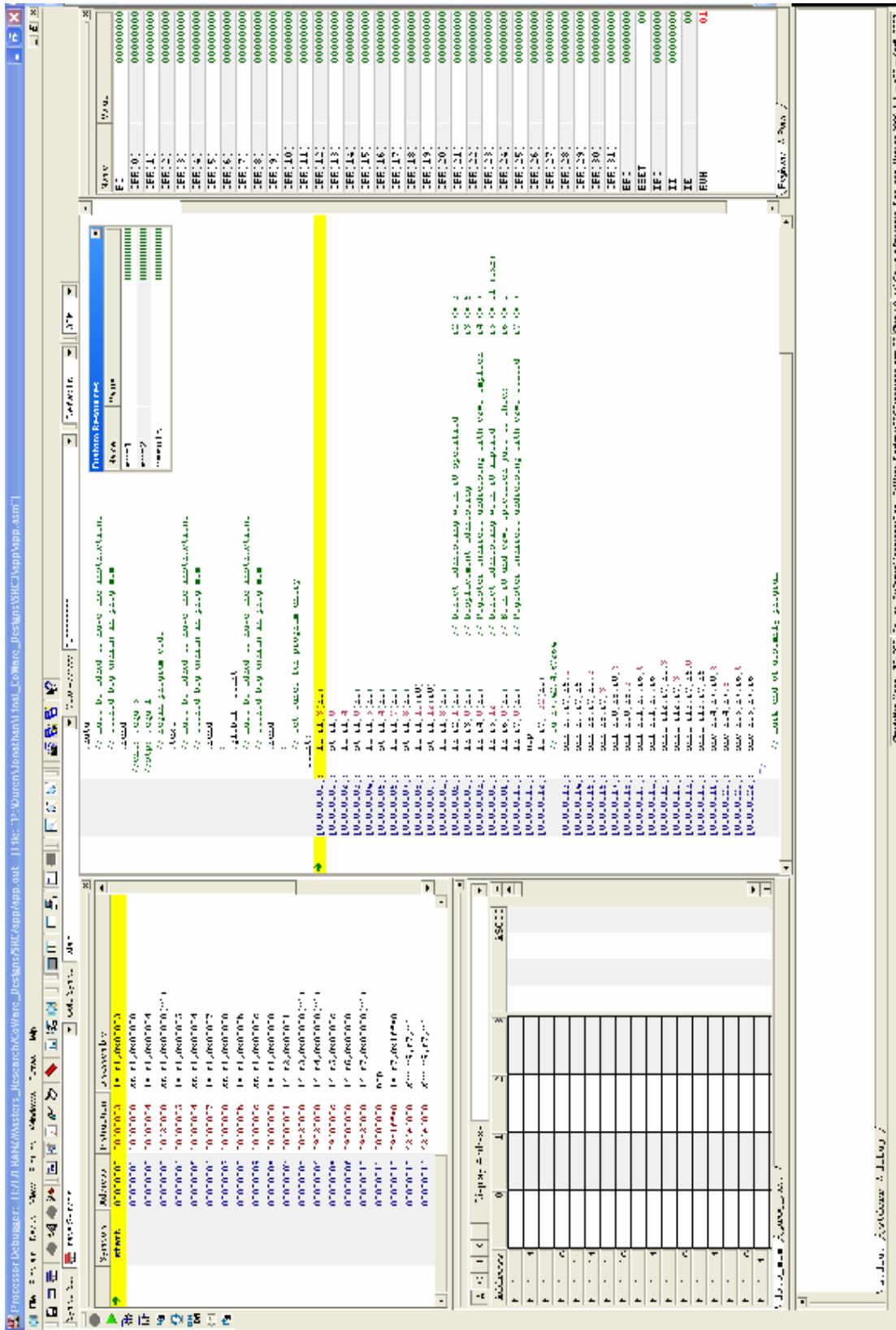


Figure 4.5: Processor Debugger Screenshot

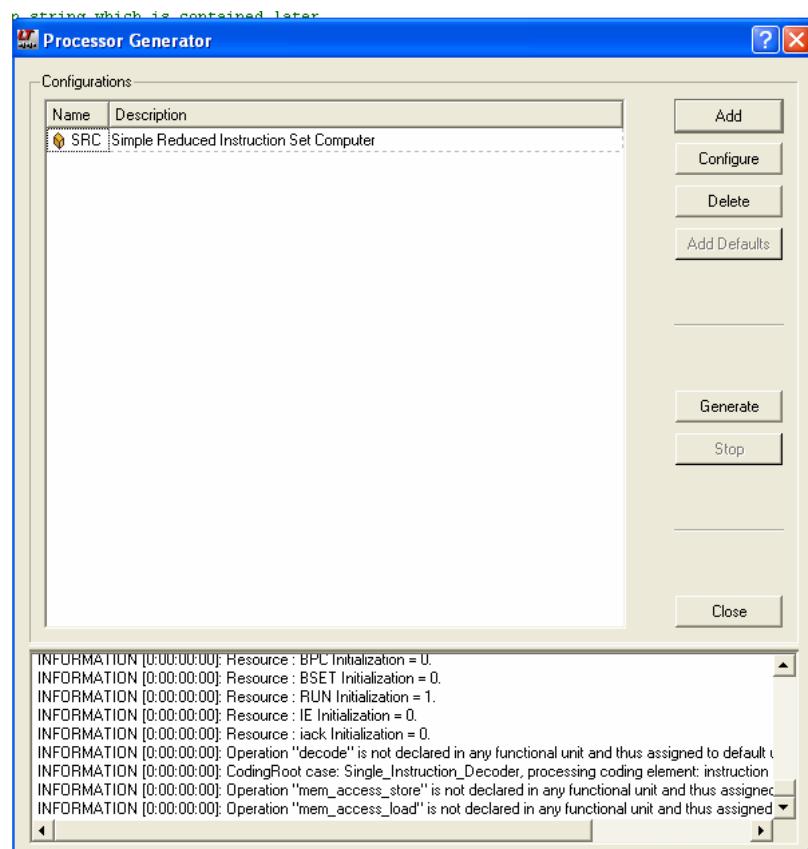


Figure 4.6: Processor Generator Screenshot

CHAPTER FIVE

Processor Models and the LISA Language

In this chapter, the non-pipelined SRC implementation and portions of the pipelined SRC, VLIW SRC, and AYK-14 are detailed to help illustrate the structure of the LISA language. Shortcomings of the LISA language, including poorly documented areas and pitfalls associated with learning the language, are also discussed.

Processor Designer is a design environment in which the user can fully express the processor architecture that they are modeling. Within Processor Designer the development process is broken up into two parts: the data path and the control unit. The data path is defined as “the set of interconnections and auxiliary registers needed to accomplish the overall changes an instruction makes in the programmer-visible objects” [4] and the control unit is responsible for affecting the data flow through the architecture. The data path is defined within the LISA model with use of the keyword *RESOURCE* while the control unit portion of the design is structured around a LISA defined process known as an *OPERATION*.

5.1 SRC Implementation

5.1.1 RESOURCE Definition

The *RESOURCE* section in LISA contains the declaration of combinational and sequential hardware; this includes memories, registers, buses, pipelines, and input/output pins. The LISA language provides syntax constructs to help with the definition of

processor resources. The following sections describe the declaration of the SRC hardware within LISA as well as the corresponding hardware generated.[19] [20] [21]

Memory must be defined within the *RESOURCE* section of the LISA code. The code in Figure 5.1 below shows how the memory was specified in LISA. Many options are available for defining the type of memory used. The SRC that was modeled implemented a Harvard memory architecture in which program instructions are stored in memory separate from program data. The *SIZE* parameter defines the total size of the memory block. In the SRC model 64k was defined for the program instructions and 64k for the program data. The *BLOCKSIZE* parameter defines the bit-width of the memory. 32-bit blocks of data and program memory were defined for the non-pipelined SRC. The contents of program memory was readable and executable only while the contents of data memory was readable and writeable. These requirements are communicated to Processor Designer within the *FLAGS* section. The SRC memory was to be organized in big-endian format. The LISA keyword *ENDIANESS* was used within the memory definition, followed by either *BIG* or *SMALL* to distinguish between big-endian and small-endian formats. The first block of *RAM* code implemented a 64k block of ROM for program memory while the second *RAM* code block implemented a 64k block of RAM for the data memory.[20] [21]

The RAM and ROM are connected to the SRC via the *MEMORY_MAP* declaration. The *MEMORY_MAP* section in LISA is used to provide a link between the virtual memory addresses and physical memory resources.[20] [21] The parameters PMEM_START, PMEM_END, PMEM_SIZE, DMEM_START, DMEM_END, and D_MEM_SIZE are part of a header file, memory_cfg.h, and are given in APPENDIX F.

In simulation within the debugger this memory design worked fine. The design specifications from *Computer Systems Design and Architecture Second Edition* for the SRC, however, called for byte size memory. It was simpler to debug the LISA code with 32-bit address. For this reason it was tested and verified with 32-bit blocks of memory and then the code was modified to allow for a byte size memory to be attached when the RTL code was generated. The memory declaration in the Processor Designer defines an external memory. When Processor Generator is run a MemoryFile_gen.vhd (or .v if verilog was used) file is created. This file has control signals for the memory and allows the user to attach their own external memory to interface with these control signals.

```

RESOURCE
{
    MEMORY_MAP
    {
        RANGE(PMEM_START, PMEM_END) -> prog_mem[(31..0)];
        RANGE(DMEM_START, DMEM_END) -> data_mem[(31..0)];
    }

    RAM uint32 prog_mem
    {
        SIZE(PMEM_SIZE);
        BLOCKSIZE(32);
        FLAGS(R|X);
        ENDIANESS(BIG);
    };
    RAM uint32 data_mem
    {
        SIZE(DMEM_SIZE);
        BLOCKSIZE(32);
        FLAGS(R|W);
        ENDIANESS(BIG);
    };
}

```

Figure 5.1: Resource Definition of Memory

Figure 5.2 below shows the declaration of all the registers, pins, and signals that the SRC needs to function properly. LISA provides the keyword *REGISTER* in conjunction with the TClocked identifier to define clocked resources. Below a set of 32

32-bit clocked GPR (General Purpose Registers) are specified and a 32-bit clocked PC (Program Counter) is also specified.[16] [20] [21]

```

REGISTER TClocked<int32> GPR[0..31];

PROGRAM_COUNTER TClocked<uint32> PC;
uint32 IR;

REGISTER TClocked<uint32> BPC;
REGISTER TClocked<bool> BSET;

int32 src1;
int32 src2;
int32 result;
unsigned char alu_mode;

uint32 MD; // Data Signal to Memory

REGISTER TClocked<uint32> IPC;
REGISTER TClocked<uint32> II;

REGISTER TClocked<bool> IE;
REGISTER TClocked<bool> RUN;

PIN IN bool ireq;
PIN OUT bool iack;

PIPELINE pipe = { ONE };

```

Figure 5.2: Resource Definition of Registers and Pins

The 32-bit IR (Instruction Register) is treated as a wire since it must be read in the same clock cycle as it is being written. Wires are declared without special keywords. LISA treats the entire instruction as executing in a single clock cycle if no pipeline is defined in the model, or if a single stage pipeline is defined; as a result, a few of the registers that were defined in the hardware model from the previous chapter are now implemented as wires.

The *PIN* declaration tells Processor Designer that the current signal will be read from an external source (*PIN IN*), written to an external source (*PIN OUT*), or both (*PIN INOUT*). All the resources needed for the SRC processor itself (that is without external

memory) are shown in Figure 5.2. A single stage pipeline is also declared within the Resource section. Since Processor Generator requires that the architecture be pipelined.[21] Figure 5.3 shows a layout of the hardware that the description of the non-pipelined SRC would produce.

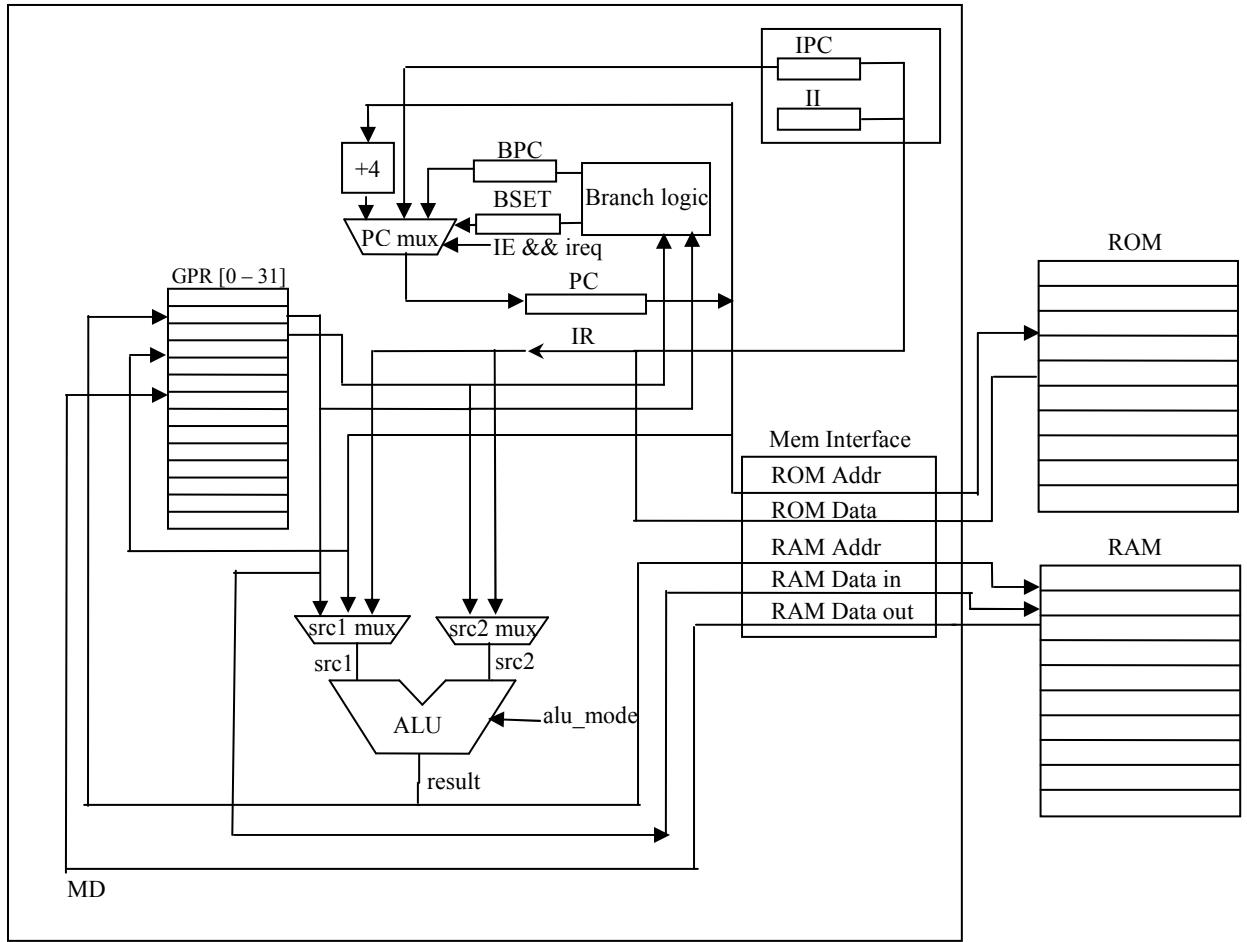


Figure 5.3: Non-pipelined SRC Architecture

In Processor Debugger all of the *RESOURCE* declarations can be monitored. This allows for complete verification and testing of the processor in software before any hardware (RTL) is created.[17] All the resources declared in the *RESOURCE* section are translated into RTL in three processes: a write multiplexer (combinational logic), a

storage element (sequential logic), and a read multiplexer (combinational logic). Not all resources declared will contain the three RTL processes. For example, the input pin defined in the SRC model would only have the read process instantiated as it does not need to be written to or store a value.[18]

5.1.2 OPERATION Definitions

Every LISA model must contain three mandatory *OPERATION* sections: reset, main, and fetch. The reset operation for the SRC is shown below in Figure 5.4. The operation must be named reset and it may only contain a *BEHAVIOR* section. The purpose for the reset operation is to define an initial state for the resources of the model.[16] [20] [21]

```
OPERATION reset {
    BEHAVIOR
    {
        int i;
        for(i = 0; i < 32; i++)
        {
            GPR[i] = 0;
        }
        PC = LISA_PROGRAM_COUNTER;
        BPC = 0;
        BSET = false;
        RUN = true;
        IE = false;
        iack = false;
        PIPELINE(pipe).flush();
    }
}
```

Figure 5.4: SRC Reset Operation

The *BEHAVIOR* section of an *OPERATION* is coded in ANSI-C. The *for loop* clears the SRC's GPR file. LISA provides a special keyword, *LISA_PROGRAM_COUNTER*, which works with the *_start* identifier of the assembly program to define the code's entry point. This is shown assigned to the PC below. The

one stage pipeline is flushed in the reset operation for completeness, but could have been left off as no pipeline registers were defined in the model. The reset operation defines the reset circuitry created by Processor Generator.[20] [21]

LISA requires that a main operation also be present in every design. The main operation is triggered every control step and is used to step the design one clock cycle. The main operation from the SRC model is given in Figure 5.5. Its main functionality is to execute and then shift the pipeline every control step (clock cycle). The main operation introduces a new LISA section, *DECLARE*. The *DECLARE* section is responsible for declaring local variables for the current *OPERATION*. Within the *DECLARE* section an *INSTANCE* of another *OPERATION* (*fetch*) is defined. *INSTANCE* is a keyword in LISA that alerts the current *OPERATION* of another *OPERATION* that will be called from within the current *OPERATION* at a latter time. The *ACTIVATION* section acts as a function call to the *fetch* operation which then starts its execution. Operations that are activated from another will occur in their assigned pipeline stage. In the non-pipelined SRC model they will occur immediately since we have defined only one pipeline stage.[20] [21]

```
OPERATION main {
    DECLARE {
        INSTANCE fetch;
    }
    BEHAVIOR {
        PIPELINE(pipe) .execute();
        PIPELINE(pipe) .shift();
    }
    ACTIVATION { fetch }
}
```

Figure 5.5: SRC Main Operation

Operations main and reset are not assigned to a pipeline stage because neither of them have a direct mapping into the generated HDL code. The remainder of the operations that effect the instruction execution in the SRC design will be assigned to a pipeline stage.[16] [20] [21]

A fetch operation is a third mandatory operation for any LISA processor model. The fetch operation must be assigned to a pipeline stage, and in the case of the single stage SRC it is assigned to stage ONE.[16] [20] [21] The SRC fetch operation is depicted in Figure 5.6. The fetch operation is responsible for accessing the instruction word to be executed from memory. Program memory is accessed via a function call to PMEM_LD. PMEM_LD is defined in the memory_if.h header file in the APPENDIX F, as well as Figure 5.7.

The fetch operation is also responsible for making sure the SRC execution has not been halted (RUN) and that a branch has not been detected (BSET). Interrupts are also detected and serviced from within the SRC fetch operation. This is accomplished in the generated hardware by a PC multiplexer as shown in Figure 5.3. If an interrupt is detected then the interrupt vector is read from the highest word of data memory. The fetch operation also declares and calls the decode operation. The reset, main, and fetch operations are instruction independent and for that reason never contain *CODING* or *SYNTAX* sections. The #ifdef LT_PROCESSOR_GENERATOR segment of code allows the LISA model run within Processor Debugger to address 32-bit word address of memory while allowing Processor Generator to create HDL code that addresses byte size memory address. The program memory is 64k in size and so the PC counter should not read memory above address 0xffff.

```

OPERATION fetch IN pipe.ONE {
    DECLARE {
        INSTANCE decode;
    }
    BEHAVIOR {
        uint32 temp_PC;
        if(!RUN) { PC += 0; }
        else {
            if(BSET) { temp_PC = BPC; }
            else { temp_PC = PC; }

            if(ireq && IE)
            {
                uint32 iack_data;
                iack = true;

                PMEM_LD(iack_data,temp_PC);
                II = 0x0000ffff & iack_data;
                PC = 0x000000ff & (iack_data >> 16);
                IPC = temp_PC;
                IR = 0;
            }
            else
            {
                iack = false;
                PMEM_LD(IR,temp_PC);

                if (PC >= 0xffff) {
                    temp_PC = 0;
                }
                else
                {
#ifdef LT_PROCESSOR_GENERATOR
                    temp_PC += 4;
#else
                    temp_PC += 1;
#endif
                }
                PC = temp_PC;
            }
        }
    }
    ACTIVATION { decode }
}

```

Figure 5.6: SRC Fetch Operation

All designs in LISA must also contain a coding root, for the case of the SRC this is the decode operation, Figure 5.8. The coding root in a LISA model is distinguished by the use of the reserved LISA keywords, *CODING AT*. The keyword *AT* defines a *RESOURCE* containing the memory address of the current instruction. The decode operation is the top of the instruction tree, where the initial decision is made in

```

#ifndef MEMORY_IF_H
#define MEMORY_IF_H

#include "memory_cfg.h"

#define PMEM_LD(dst,addr)
dst=prog_mem[addr];
#define DMEM_LD(dst,addr)
dst=data_mem[addr];
#define DMEM_ST(src,addr)
data_mem[addr]=src;

#endif

```

Figure 5.7: Memory_if.h

the decoding process. The decode process contains a *GROUP* in the *DECLARE* section. A *GROUP* declaration is similar to an enum declaration in C/C++, and lists all the operations or sets of operations that the instruction can represent (that are present in the current architecture).[20] [21] The *BEHAVIOR* section contains logic to reset the BSET flag for all instructions that are not in the branch class. BRANCH and BRANCHL are constants defined in a header file, defines.h, located in the APPENDIX G.

```

OPERATION decode IN pipe.ONE {
    DECLARE {
        GROUP instruction = { nop || stop || alui || alur || aluu ||
                                branch || branchl || ld || ld_simple ||
                                la || la_simple || st || st_simple ||
                                ldr || lar || str || shift || shift_count ||
                                shift_reg || ri || svi || rfi || edi || een };
    }
    CODING AT ( PC ) { IR == instruction }
    SYNTAX { instruction }
    BEHAVIOR {
        char op = IR >> 27 & 0x1f;
        if((op != BRANCH) && (op != BRANCHL)) { BSET = false; }
    }
    ACTIVATION { instruction }
}

```

Figure 5.8: SRC Decode Operation

LISA permits an *OPERATION* to model instruction operands, as in Figure 5.9. In the case of the SRC, almost all the instructions access a general purpose register, whether

to read from it or write to it. Instead of coding this common functionality for every instruction, an *OPERATION* can be defined that will define the *CODING* and *SYNTAX* for the register operands, and return an index of the register. The *SYNTAX* section defines the syntax for the current instruction, and the *CODING* section defines the binary encoding of the instruction. LISA permits a reserved word *IMMEDIATE* in place of *OPERATION* to distinguish certain operations as immediate, or instruction independent. Because *IMMEDIATE* operations are instruction independent they do not have to be assigned to a pipeline stage.[20] [21]

The *EXPRESSION* identifier acts like the reserved word *return* in C/C++, and is used in place of the *BEHAVIOR* section to express a numerical value (constant or label identifier) that is passed back to a parent or calling operation. *EXPRESSION* and *BEHAVIOR* sections are mutually exclusive, that is, an operation can have one or the other, but not both. Two more immediate operations, imm5 and imm17, are defined in the SRC model (see APPENDIX A for their coding) to represent the c2 and count operands in instruction formats one and seven respectively (Refer to Figure 3.1 for the instruction formats).[20] [21]

```
IMMEDIATE reg {
    DECLARE { LABEL idx; }
    SYNTAX { "r" ~idx=#U5 }
    CODING { idx=0bx[5] }
    EXPRESSION {idx}
    DOCUMENTATION { General Purpose Register r0 - r31 }
}
```

Figure 5.9: SRC Register Operation (Immediate)

Instructions that share the same format can be grouped together in the same operation. Since all the ALU immediate instructions share the same format (format 1)

the instructions may be grouped in the same operation, *alu*, Figure 5.10. The opcode *GROUP* defines the different instructions that are part of the ALU immediate class. The group members of the opcode group are called depending on the instruction currently being executed. If *addi* were the instruction then the *addi* operations coding and syntax section would contribute to the syntax and coding section of the overall instruction. At the same time during decoding, the reg immediate operation, defined above, and c2 immediate operation are called to fill in the specified coding and syntax sections of the instruction as well as return expressions for the group variables. All members of a group must contain a *CODING* and *SYNTAX* section and all coding sections must have the same bit-width.[20] [21]

To reduce the amount of redundant logic, the ALU functionality has been extracted and placed inside its own *OPERATION* (*alu*, defined later). This reduces the amount of logic that will be present in the generated HDL code. The sub-operations of the *alu* operation, i.e. one of the three members of the opcode group, return an expression that will tell the ALU what operation to perform (add, sub, not, etc...). Since the opcode group members do not contribute to the behavior of the instruction they do not have to be assigned to a pipeline stage. As a general rule: A leaf operation that does not contain a *BEHAVIOR* section need not be assigned to a pipeline stage. The write back functionality is also similar in many operations and has also been extracted to its own operation, *writeback*, defined later.[20] [21]

The *DOCUMENTATION* keyword can be used within LISA *OPERATIONS*. Statements enclosed within *DOCUMENTATION* sections are not part of the generated HDL code or software toolsuite, instead they are used in the creation of an instruction set

users manual.[22] Figure 5.10 below, shows the use of the *DOCUMENTATION* keyword. APPENDICES J and K show the instruction set users manuals created for the SRC and AYK-14 processors respectively.

```

OPERATION alui IN pipe.ONE {
    DECLARE {
        GROUP opcode = { addi || ori || andi };
        GROUP ra,rb = { reg };
        GROUP c2      = { imm17 };
        INSTANCE alu;
        INSTANCE writeback;
    }
    CODING   { opcode ra rb c2 }
    SYNTAX   { opcode ~" " ra "," rb "," c2 }
    BEHAVIOR {
        src1 = GPR[rb]; // Load GPR[rb] into src1
        src2 = SIGN_EXTEND_15(c2); // Put immediate into src2
        alu_mode = opcode; // Set the ALU mode
    }
    ACTIVATION { alu, writeback } // Activate the ALU & writeback
    DOCUMENTATION("ALU IMMEDIATE") { ALU Immediate Instructions }
}

OPERATION addi {
    CODING   { 0b01101 }
    SYNTAX   { "addi" }
    EXPRESSION { ALU_ADD }
    DOCUMENTATION { Add rb to immediate constant, and put result into ra }
}

OPERATION ori {
    CODING   { 0b10111 }
    SYNTAX   { "ori" }
    EXPRESSION { ALU_OR }
    DOCUMENTATION { "OR" rb and immediate constant, and put result into ra }
}

OPERATION andi {
    CODING   { 0b10101 }
    SYNTAX   { "andi" }
    EXPRESSION { ALU_AND }
    DOCUMENTATION { "AND" rb and immediate constnat, and put result into ra }
}

```

Figure 5.10: SRC ALUI Instructions

An ALU operation may be either called from within the *BEHAVIOR* section of the parent operation or activated from within an *ACTIVATION* section. Calling an operation from within a *BEHAVIOR* section causes all the behavioral code of the called

operation to be duplicated and placed inline with the calling operation during RTL generation. This counteracts our efforts to reduce the amount of logic generated by the grouping of common functionality. For this reason, the ALU operation is called from within an *ACTIVATION* section. This causes the ALU to be referenced and the code is not duplicated at RTL generation. Operations that occur in a different pipeline stage than their parents must be called from within an *ACTIVATION* section.[20] [21]

The ALU register instructions share the same instruction format (format 6) and may be grouped into a single parent operation similar to the ALU Immediate instructions. Figure 5.11 shows the *alur* operation. It differs from the *alui* operation defined above in the sense that there is no immediate value operand; instead both operands come from the register file. Group c2 defined above is therefore replaced with group rc. The ALU unary instructions are grouped together in an *aluu* operation as shown in Figure 5.12. The ALU unary instructions take only one operand, rc.

The shift instructions also share similar instruction formats (format 7a and 7b). Since the shift instructions may be expressed with two distinct syntaxes to the assembler, LISA needs a way of modeling this. The shift instruction can either shift a value by an immediate constant value, count, or by a register value, rc. The behavior of the shift instruction will be the same with either syntax; the only difference is where the source of the shift count operand comes. The actual binary encoding of the shift instruction will be similar. Figure 5.13 shows the shift instructions in LISA.

LISA provides a reserved word *ALIAS* to model an instruction with two different syntaxes. Operations defined with the keyword *ALIAS* are given a higher priority and will be examined first

```

OPERATION alur IN pipe.ONE {
    DECLARE {
        GROUP opcode = { add || sub || and || or };
        GROUP ra,rb,rc = { reg };
        INSTANCE alu;
        INSTANCE writeback;
    }
    CODING { opcode ra rb rc 0b0[12] }
    SYNTAX { opcode ~" " ra "," rb "," rc }
    BEHAVIOR {
        src1 = GPR[rb];
        src2 = GPR[rc];
        alu_mode = opcode;
    }
    ACTIVATION { alu, writeback } // Activate the alu & writeback
    DOCUMENTATION("ALU REGISTER") { ALU Register Instructions }
}

OPERATION add {
    CODING { 0b01100 }
    SYNTAX { "add" }
    EXPRESSION { ALU_ADD }
    DOCUMENTATION { Add rb to rc, and put result into ra }
}

```

Figure 5.11: SRC ALUR Instructions

```

OPERATION aluu IN pipe.ONE {
    DECLARE {
        GROUP opcode = { neg || not };
        GROUP ra,rc = { reg };
        INSTANCE alu;
        INSTANCE writeback;
    }
    CODING { opcode ra 0b0[5] rc 0b0[12] }
    SYNTAX { opcode ~" " ra "," rc }
    BEHAVIOR {
        src1 = 0;
        src2 = GPR[rc];
        alu_mode = opcode;
    }
    ACTIVATION { alu, writeback } // Activate the alu & writeback
    DOCUMENTATION ("ALU UNARY") { ALU Unary Instructions }
}

OPERATION neg {
    CODING { 0b01111 }
    SYNTAX { "neg" }
    EXPRESSION { ALU_NEG }
    DOCUMENTATION { Place 2's compliment negative of rc into ra }
}

```

Figure 5.12: SRC ALUU Instructions

```

OPERATION shift IN pipe.ONE {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP c3 = { imm5 };
        GROUP ra,rb,rc = { reg };
        INSTANCE alu, writeback;
    }
    CODING { opcode ra rb rc 0bx[7] c3 }
    SYNTAX { opcode ~" " ra "," rb "," rc "," c3 }
    BEHAVIOR {
        if(c3) { src2 = c3; }
        else { src2 = GPR[rc]; }
        src1 = GPR[rb];
        alu_mode = opcode;
    }
    ACTIVATION { alu, writeback }
}

ALIAS OPERATION shift_count IN pipe.ONE {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP c3 = { imm5 };
        GROUP ra,rb = { reg };
        INSTANCE alu, writeback;
    }
    CODING { opcode ra rb 0b0[12] c3 }
    SYNTAX { opcode ~" " ra "," rb "," c3 }
    BEHAVIOR {
        src2 = c3;
        src1 = GPR[rb];
        alu_mode = opcode;
    }
    ACTIVATION { alu, writeback }
}

ALIAS OPERATION shift_reg IN pipe.ONE {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP ra,rb,rc = { reg };
        INSTANCE alu, writeback;
    }
    CODING { opcode ra rb rc 0b0[12] }
    SYNTAX { opcode ~" " ra "," rb "," rc }
    BEHAVIOR {
        src2 = GPR[rc];
        src1 = GPR[rb];
        alu_mode = opcode;
    }
    ACTIVATION { alu, writeback }
}

OPERATION shr {
    CODING { 0b11010 }
    SYNTAX { "shr" }
    EXPRESSION { ALU_SHR }
}

```

Figure 5.13: SRC Shift Instructions

by the assembler. Since *ALIAS* operations are used to model special instances of already defined instructions, RTL code is not generated for these operations. This helps reduce the amount of logic that is present in the generated RTL code and reduces the decoding logic.[20] [21]

The implementation of the ALU is shown in Figure 5.14. The functionality is grouped together into an *OPERATION* named *alu*. The disadvantage of grouping the ALU into a single operation as opposed to having multiple adders and other logic on the board is the introduction of two new multiplexers. The multiplexers are used to select which value of *src1* and which value of *src2* are to be passed to the ALU operation. The ALU and multiplexers are shown in Figure 5.3 above.

The memory access instructions for the SRC come in two distinct formats and three addressing modes: direct, indexing, and relative. The operation encoding of these instructions in LISA is very similar and differs only in the operations activated within the operation. All three operations will activate the ALU, however the loads and load addresses are the only two that activate writeback. The loads also activate operation *mem_access_load* and the stores activate operation *mem_access_store*. For this reason only the loads will be considered in the discussion below.

The load (*ld*) instruction may work in either a direct addressing or an indexing mode, depending on the *rb* operand. Figure 5.15 shows two operations, *ld* and *ld_simple*. The *ld* operation takes care of the syntax for the load instruction in which all parameters are present. It accepts all the parameters, *ra*, *rb*, and *c2*, and depending on *rb* will implement either direct or index addressing. The *ALIAS* operation *ld_simple* covers the

```

OPERATION alu IN pipe.ONE {
    BEHAVIOR {
        switch(alu_mode) {
            case ALU_ADD: // Add
                result = src1 + src2;
                break;
            case ALU_SUB: // Sub
                result = src1 - src2;
                break;
            case ALU_AND: // And
                result = src1 & src2;
                break;
            case ALU_OR: // Or
                result = src1 | src2;
                break;
            case ALU_NEG: // Neg
                result = -src2;
                break;
            case ALU_NOT: // Not
                result = ~src2;
                break;
            case ALU_SHR: // Shift Right
                result = (uint32)src1 >> src2;
                break;
            case ALU_SHRA: // Shift Right Arithmetic
                result = src1 >> src2;
                break;
            case ALU_SHL: // Shift Left
                result = src1 << src2;
                break;
            case ALU_SHC: // Shift Circular
                result = (src1 << src2) | ((uint32)src1 >> (32-src2));
                break;
            default:
                result = 0;
                break;
        }
    }
}

```

Figure 5.14: SRC ALU Functionality

other possible syntax case.[21] In the *ld_simple* case the *rb* field is explicitly left off and direct addressing is used. The store (*st*) and load address (*la*) instructions are similar to the load instruction operations and are given in the complete code in APPENDIX A.

The load relative instruction introduces a few new features of the LISA language, and is declared in the *OPERATION ldr*, Figure 5.16. Operand *c1* of the instruction is declared within the operation as a *LABEL* instead of a *GROUP* as the previous examples have been. This is done because *c1* is a variable that is declared and defined completely

```

OPERATION ld IN pipe.ONE {
    DECLARE
    {
        GROUP ra,rb = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu, mem_access_load, writeback;
    }
    CODING { 0b00001 ra rb c2 }
    SYNTAX { "ld" ~" " ra "," c2 "(" rb ")" }
    BEHAVIOR {
        if(rb) { src2 = GPR[rb]; }
        else { src2 = 0; }
        src1 = SIGN_EXTEND_15(c2);
        alu_mode = ALU_ADD;
    }
    ACTIVATION { alu, mem_access_load, writeback }
    DOCUMENTATION("LD") { Load from displacement address }
}

ALIAS OPERATION ld_simple IN pipe.ONE {
    DECLARE {
        GROUP ra = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu, mem_access_load, writeback;
    }
    CODING { 0b00001 ra 0b0[5] c2 }
    SYNTAX { "ld" ~" " ra "," c2 }
    BEHAVIOR {
        src2 = 0;
        src1 = SIGN_EXTEND_15(c2);
        alu_mode = ALU_ADD;
    }
    ACTIVATION { alu, mem_access_load, writeback }
    DOCUMENTATION("LD") { Load from absolute address; rb is register 0 }
}

```

Figure 5.15: SRC Load Instruction

```

OPERATION ldr IN pipe.ONE {
    DECLARE {
        GROUP ra = { reg };
        LABEL c1;
        INSTANCE alu, mem_access_load, writeback;
    }
    CODING { 0b00010 ra c1=0bx[22] }
    SYNTAX { "ldr" ~" " ra "," SYMBOL(((c1=#$22))+CURRENT_ADDRESS)=#X32) }
    BEHAVIOR {
        src1 = PC;
        src2 = SIGN_EXTEND_10(c1);
        alu_mode = ALU_ADD;
    }
    ACTIVATION { alu, mem_access_load, writeback }
    DOCUMENTATION("LDR") { Load from a relative address }
}

```

Figure 5.16: SRC Load Relative Instruction

within the current operation. An operand defined by a *LABEL* may be used within the *CODING*, *SYNTAX*, and *BEHAVIOR* sections of the current operation.[20] [21]

The *ldr* operation calculates the address relative to the current PC. In order to make this easier on the designer, LISA reserves the keywords *CURRENT_ADDRESS* to define the current address of the instruction being executed (PC).[21] In the example below, c1 would hold a signed value that is equal to the desired memory address plus the current PC. The complete source code for the load address relative (*lar*) and store relative (*str*) instructions are given APPENDIX A. The memory access relative instructions make use of a new LISA construct, *SYMBOL*. The *SYMBOL* keyword allows symbolic immediate values, such as labels, to be used in the processor's assembly syntax definition.[21] The c2 and count operands, defined in OPERATIONS imm17 and imm5 respectively in APPENDIX A, also make use of the *SYMBOL* keyword, which allows them to be defined by symbolic constants.

The memory interface unit the SRC uses to communicate with external memory, either RAM or ROM, is described within the operations *mem_access_load* and *mem_access_store*, Figure 5.17. The *mem_access_load* operation takes the desired value from RAM and writes it to the MD wire. A *REFERENCE* is used in the *mem_access_store* operation to pass the index to the general purpose register from the parent operation. The contents of that register index are then stored to RAM. The use of a *REFERENCE* in LISA is similar to parameter passing between functions in C/C++.[16] [20] [21]

The majority of the SRC instructions require a value to be written to the general purpose register file. Because of this common functionality, the writeback operation is

declared, Figure 5.18. Combining the write back functionality of the SRC into one operation reduces the number of access paths to the register file. This in return reduces the logic and board space when the LISA model is compiled into RTL.

```

OPERATION mem_access_load IN pipe.ONE {
    DECLARE { REFERENCE ra; }
    BEHAVIOR {
        uint32 addr;
        uint32 data;

        addr = result;
        DMEM_LD(data,addr);
        MD = data;
    }
}

OPERATION mem_access_store IN pipe.ONE {
    DECLARE { REFERENCE ra; }
    BEHAVIOR {
        uint32 addr;
        uint32 data;

        addr = result;
        data = GPR[ ra ];
        DMEM_ST(data,addr);
    }
}

```

Figure 5.17: SRC Memory Interface

```

OPERATION writeback IN pipe.ONE {
    DECLARE { REFERENCE ra; }
    BEHAVIOR {
        char op = IR >> 27 & 0x1f;
        if(op == LD || op == LDR)
            { GPR[ ra ] = MD; }
        else { GPR[ ra ] = result; }
    }
}

```

Figure 5.18: SRC Write Back Functionality

The SRC's branch instructions are modeled in two separate operations, one for branch and one for branch link. The difference between the instructions, besides the opcode, is that branch link stores the PC into a general-purpose register. Both

instructions share a common logic unit, branch logic, which has been extracted and placed into a separate operation. The Branch Link operation is shown in Figure 5.19.

```

OPERATION branchl IN pipe.ONE {
    DECLARE {
        GROUP ra,rb,rc = { reg };
        GROUP c3 = { brlnv || brl || brlzs || brlnz || brlpl || brlmi };
        INSTANCE Branch_Logic;
    }
    CODING { 0b01001 ra rb rc 0b0[9] c3 }
    SYNTAX { c3 }
    BEHAVIOR {
        GPR[ra] = PC;
    }
    ACTIVATION { Branch_Logic }
    DOCUMENTATION("BRANCH LINK") { Branch to rb if rc satisfies c3,
        and save PC in ra }
}

OPERATION brlnv {
    DECLARE {
        REFERENCE ra,rb,rc;
    }
    CODING { 0b000 }
    SYNTAX { "brlnv" ~" " ra }
    EXPRESSION { BRNV }
    DOCUMENTATION { Do not branch, but save PC in ra }
}

```

Figure 5.19 SRC Branch Link Instructions

The Branch Logic operation is show in Figure 5.20 below and produces the Branch Logic box shown in Figure 5.3, SRC hardware. The Branch Logic operation is responsible for testing the branch instruction condition and setting a branch flag, if the branch is to be taken, which the fetch operation uses to adjust the PC.

The majority of the interrupt instruction as well as *nop* and *stop* have instruction formats that only use the opcode bit field, format 8. These instructions are coded within their own operations, and do not access the register file or main memory, with the exception of the *svi* and *ri* instructions. While each one behaves differently, they all write to a status register, RUN or IE, or the PC. The exception enable instruction is given below as in example, Figure 5.21.

```

OPERATION Branch_Logic IN pipe.ONE {
    DECLARE {
        REFERENCE c3;
        REFERENCE rb,rc;
    }
    BEHAVIOR {
        char br_cond;

        br_cond = c3;

        switch(br_cond) {
            case BRNV: // branch never
                BSET = false;
                break;
            case BR: // branch always
                BPC = (unsigned int)GPR[rb];
                BSET = true;
                break;
            case BRZR: // branch when condition(rc) is zero
                if((signed int)GPR[rc] == 0) {
                    BPC = (unsigned int)GPR[rb];
                    BSET = true;
                }
                else { BSET = false; }
                break;
            case BRNZ: // branch when condition(rc) is non-zero
                if((signed int)GPR[rc] != 0) {
                    BPC = (unsigned int)GPR[rb];
                    BSET = true;
                }
                else { BSET = false; }
                break;
            case BRPL: // branch when condition(rc) is positive (>= 0)
                if((signed int)GPR[rc] >= 0) {
                    BPC = (unsigned int)GPR[rb];
                    BSET = true;
                }
                else { BSET = false; }
                break;
            case BRMI: // branch when condition(rc) is negative (< 0)
                if((signed int)GPR[rc] < 0) {
                    BPC = (unsigned int)GPR[rb];
                    BSET = true;
                }
                else { BSET = false; }
                break;
            default:
                BSET = false;
                break;
        }
    }
}

```

Figure 5.20 SRC Branch Logic Functionality

LISA also provides a very useful method of organizing *OPERATIONS* and *REGISTERS* into user defined modules (functional units) within the generated RTL.

```

OPERATION een IN pipe.ONE {
    CODING { 0b01010 0b0[27] }
    SYNTAX { "een" }
    BEHAVIOR {
        IE = true;
    }
    DOCUMENTATION("EEN") { Exception enable. Set overall exception
        enable bit }
}

```

Figure 5.21: SRC Exception Enable Instruction

Figure 5.22 below shows how to use the identifier *UNIT* to group *OPERATIONS* and *REGISTERS*. The *UNIT* declarations are made within the *RESOURCE* section of the LISA code. Each *UNIT* declared will cause the Processor Generator to create a new HDL file. The HDL files will take on the names given them by the *UNIT* definition in LISA.[18]

```

UNIT U_FETCH IN pipe.ONE {
    OPERATIONS (fetch);
    REGISTERS (IR, PC);
};

UNIT U_EXECUTE IN pipe.ONE {
    OPERATIONS (alu, alui, alur, aluu, shift);
    REGISTERS (src1, src2, result, alu_mode);
};

UNIT U_BRANCH IN pipe.ONE {
    OPERATIONS (Branch_Logic, branchl);
    REGISTERS (BPC, BSET);
};

UNIT U_MEMORY IN pipe.ONE {
    OPERATIONS (ld, la, st, ldr, lar, str);
    REGISTERS (MD);
};

UNIT U_WRITEBACK IN pipe.ONE {
    OPERATIONS (writeback);
};

UNIT U_MISCELLANEOUS IN pipe.ONE {
    OPERATIONS (stop);
    REGISTERS (RUN);
};

UNIT U_INTR IN pipe.ONE {
    OPERATIONS (een, edi, rfi, svi, ri);
    REGISTERS (IPC, II, IE);
};

```

Figure 5.22 SRC UNIT Definitions

5.2 Pipelined SRC

The *OPERATIONS* defined in the SRC are very similar to the ones defined for the non-pipelined SRC and for the VLIW SRC. For this reason only the pipelined SRC code, and in the following section the VLIW SRC, Pipelined VLIW SRC, and AYK-14 code, that illustrates new features of the LISA language will be documented. The changes in the *RESOURCE* section between the non-pipelined and the pipelined version of the SRC are shown in Figure 5.23 below. The main difference is the addition of five pipeline stages and the declaration of pipeline registers. The pipeline stages are declared in the same manner as the one stage pipe was declared above. The *PIPELINE* keyword identifies the declaration of a pipeline and its stages. The pipeline registers are declared in a *PIPELINE_REGISTER* section. Declaring resources in the *PIPELINE_REGISTER* section allows Processor Designer to set up an array of registers between the pipeline stages. To save resources, when the LISA model is synthesized, the pipeline registers will be declared between the earliest stage to write to that register through to the last stage that references the register.[16] [18] [21]

```
PIPELINE pipe = { FE; DC; EX; MEM; WB };

PIPELINE_REGISTER IN pipe {
    PROGRAM_COUNTER uint32 PC2;

    uint32 IR;

    // ALU resources
    int32 X; // ALU input operand 1
    int32 Y; // ALU input operand 2
    int32 Z;
    unsigned char ALU_MODE; // selects ALU operation

    uint32 MD;
};
```

Figure 5.23: Pipelined SRC Resource Declarations

The main *OPERATION*, Figure 5.24, has only one slight adjustment over the non-pipelined counterpart. The operation fetch is now dependent on pipeline stales. For this reason the *ACTIVATION* section of the main operation checks to make sure the decode stage is not stalled before it activates the decode stage of the pipeline. This is done through a LISA command, *stalled()*, that checks the referenced pipeline stages status. The program counter for the debugger is now updated within its own *OPERATION* update_pc.[16] [21]

```

OPERATION main {
    DECLARE {
        INSTANCE fetch, update_pc;
    }
    BEHAVIOR {
        PIPELINE(pipe) .execute();
        PIPELINE(pipe) .shift();
    }
    ACTIVATION {
        update_pc
        if(!pipe.DC.IN.stalled()) { fetch }
    }
}

```

Figure 5.24: Pipelined SRC Main Operation

The fetch *OPERATION*, depicted in Figure 5.25 below, controls the execution flow of instructions similar to that of the fetch of the non-pipelined SRC. The fetch *OPERATION* also shows how to assign values to the pipeline registers. If an operation is writing a value to a pipeline register for the next stage the *OUT* identifier is used. Due to the length of the fetch *OPERATION* only the sections that show the new feature are depicted in the figure. The full code, with comments, can be found at the end of the thesis in APPENDIX B.

```

OPERATION fetch IN pipe.FE {
    DECLARE {
        INSTANCE decode;
    }
    BEHAVIOR {
        if(!RUN) { FPC += 0; }
        else {
            uint32 temp_FPC;
            .
            .
            .
            if(ireq && IE) {
                if(!branch_set) {
                    uint32 idata;
                    iack = true;

                    PMEM_LD(idata,temp_FPC);
                    II = 0x0000ffff & idata;
                    FPC = 0x000000ff & (idata >> 16);
                    IPC = temp_FPC;
                    OUT.IR = 0;
                    OUT.PC2 = temp_FPC;
                }
                else {
                    OUT.IR = 0;
                    OUT.PC2 = temp_FPC;
                    FPC = FPC;
                }
            }
            .
            .
            .
        }
    }
    ACTIVATION { decode }
}

```

Figure 5.25: Pipelined SRC Fetch Operation

When reading from a pipeline register, the *IN* identifier can be used to identify registers that are directly in front of the current stage. When accessing a pipeline register that is not directly in front of the current stage, the stage of the pipeline that is to be accessed followed by the *IN* identifier can be used. The ALU *OPERATION*, Figure 5.26 below shows the different ways to read from a pipeline register. The ALU operation also includes the hazard detection and data forwarding hardware for the memory unit. The hazard detection and data forwarding hardware for the ALU registers can be found in APPENDIX B at the end of this paper.[16] [21]

```

OPERATION alu POLL IN pipe.EX {
    BEHAVIOR {
        char ex_rb = EX.IN.IR >> 17 & 0x1f;
        char ex_rc = EX.IN.IR >> 12 & 0x1f;
        char ex_ra = EX.IN.IR >> 22 & 0x1f;
        char ex_op = EX.IN.IR >> 27 & 0x1f;

        char mem_ra = MEM.IN.IR >> 22 & 0x1f;
        char mem_op = MEM.IN.IR >> 27 & 0x1f;

        char wb_ra = WB.IN.IR >> 22 & 0x1f;
        char wb_op = WB.IN.IR >> 27 & 0x1f;

        if(ex_op == ST || ex_op == STR) {
            if((ex_ra == mem_ra) && // if(ra3==ra4)
               (mem_op != NOP && mem_op != BRANCH && mem_op != STOP &&
                mem_op != EEN && mem_op != EDI && mem_op != RFI &&
                mem_op != RI && mem_op != SVI && mem_op != ST && mem_op != STR))
            {
                if(mem_op == LD || mem_op == LDR)
                {
                    pipe.DC.IN.stall();
                    pipe.EX.IN.stall();
                    pipe.MEM.IN.stall();
                }
                else { OUT.MD = MEM.IN.Z; } // else update MD <- Z4
            }
            else if((ex_ra == wb_ra) &&
                   (wb_op != NOP && wb_op != NOP && wb_op != BRANCH &&
                    wb_op != STOP && wb_op != EEN && wb_op != EDI &&
                    wb_op != RFI && wb_op != RI && wb_op != SVI &&
                    wb_op != ST && wb_op != STR)) // else if(ra3==ra5) then MD <- Z5
            { OUT.MD = WB.IN.Z; }
            else { OUT.MD = IN.MD; }
        }
        .
        .
        .
    }
}

```

Figure 5.26: Pipelined SRC ALU Functionality

When a pipeline stage is stalled in LISA, the stage still reads and manipulates the data from the pipeline registers; the current pipeline stage is kept from writing this result to the next pipeline stage. The next cycle after the stall, the stalled *OPERATION* will not reprocess any new data that may have been shifted into its input pipeline registers, instead it remembers what the value was before the stall and then forwards the result to the next stage. To make the *OPERATION* check the pipeline registers after a stall the

POLL identifier is used. The ALU *OPERATION* depicted in Figure 5.26, gives an example of the use of *POLL*.[16] [21]

5.3 VLIW SRC

LISA allows the declaration of template *RESOURCES* and *OPERATIONS* to aide in the design of VLIW architectures. In LISA *RESOURCES* can be declared with an ‘< 0 .. n >’ extension to identify the particular *RESOURCE* as a template. This creates multiple instances of the particular resource and gives each pipeline access to one of them. Figure 5.27, below, shows the declaration of multiple template *RESOURCES*.[16] [21]

```

PIPELINE_REGISTER IN pipe {
    PROGRAM_COUNTER uint32 PC2;

    uint32 IR0;
    uint32 IR1;

    // ALU resources
    int32 X<0..1>; // ALU input operand 1
    int32 Y<0..1>; // ALU input operand 2
    int32 Z<0..1>; // Writeback value
    unsigned char ALU_MODE<0..1>; // selects ALU operation

    char op0<0..1>;
    char ra0<0..1>;
    char rb0<0..1>;
    char rc0<0..1>;

    uint32 MD;
};

```

Figure 5.27: VLIW SRC Resource Declarations

The VLIW architecture is capable of executing two 32-bit instructions in parallel. This requires the processor fetch a 64-bit double word from memory. To model this in LISA two 32-bit instruction registers are declared, IR1 and IR2, and then concatenated together within the *CODING AT* section to make a 64-bit instruction. The two groups,

declared as insn1 and insn2, define the instructions that are capable of being executed in the two pipelines. Certain *OPERATIONS* that are capable of being executed in both pipelines are defined with a template parameter that declares the pipeline that it is assigned to. The assembler requires that the instructions be given with insn1 for pipeline path 0 first, followed by a ‘|’ and then the second instruction, insn2. Figure 5.28 shows all these new features present in the design of a VLIW architecture.[15] [16] [21]

```

OPERATION decode IN pipe.DC {
    DECLARE {
        GROUP insn1 = { nop<0> || stop<0> ||
                        alui<0> || alur<0> || aluu<0> ||
                        la<0> || la_simple<0> || lar<0> ||
                        ld || ld_simple || ldr ||
                        st || st_simple || str ||
                        ri || svi || rfi || edi || een };
        GROUP insn2 = { nop<1> || stop<1> ||
                        alur<1> || alui<1> || aluu<1> ||
                        la<1> || la_simple<1> || lar<1> ||
                        shift || shift_reg || shift_count ||
                        branch || branchl };
    }
    CODING AT ( IN.PC2 ) { (IN.IR1 == insn2) && (IN.IR0 == insn1) }
    SYNTAX { insn1 "|" insn2 }
    ACTIVATION { insn1, insn2 }
}

```

Figure 5.28: VLIW SRC Decode Operation

The template *OPERATIONS* are defined similar to *alui* below in Figure 5.29. The ‘<id>’ extension after the *OPERATION* name identifies the particular *OPERATION* as being a template. This permits the particular *OPERATION* to be defined once but be active in both pipelines at the same time, thus reducing the complexity of modeling VLIW architectures.[16] [21] The complete LISA annotated code for the VLIW architecture can be found in APPENDIX C.

```

OPERATION alui<id> IN pipe.DC
{
    DECLARE
    {
        GROUP opcode = { addi || ori || andi };
        GROUP ra,rb = { reg };
        GROUP c2      = { imm17 };
        INSTANCE alu<id>, writeback<id>;
    }
    CODING   { opcode ra rb c2 }
    SYNTAX   { opcode ~" " ra "," rb "," c2 }
    BEHAVIOR
    {
        if((WB.IN.ra0<0> == rb) &&
           (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP && WB.IN.op0<0> != EEN &&
            WB.IN.op0<0> != EDI && WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
            WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST && WB.IN.op0<0> != STR))
        { OUT.X<id> = WB.IN.Z<0>; }

        else if((WB.IN.ra0<1> == rb) &&
                (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
                 WB.IN.op0<0> != BRANCH))
        { OUT.X<id> = WB.IN.Z<1>; }

        else { OUT.X<id> = GPR[rb]; }

        OUT.Y<id> = SIGN_EXTEND_15(c2);
        OUT.ALU_MODE<id> = opcode;
    }
    ACTIVATION { alu, writeback }
    DOCUMENTATION("ALU IMMEDIATE") { ALU Immediate Instructions }
}

```

Figure 5.29: VLIW SRC ALUI Instructions

5.4 Pipelined VLIW SRC

The Pipelined VLIW SRC architecture illustrates nothing new about the LISA language and for this reason is not described in this chapter. The fully commented Pipelined VLIW SRC code can be seen in APPENDIX D.

5.5 AYK-14

5.5.1 RESOURCE Definition

The AYK-14 architecture is very different than the SRC. The AYK-14 contains two 16-bit status registers and two 16-bit general-purpose register files. The programmer

can change a bit in Status Register 1 to determine which of the two general-purpose register files are used. Figure 5.30 below depicts the LISA declarations for the AYK-14 processor.

```

REGISTER TClocked<int16> R0[0..15];
REGISTER TClocked<int16> R1[0..15];

PROGRAM_COUNTER TClocked<uint32> P;

REGISTER TClocked<uint16> SR1;
REGISTER TClocked<uint16> SR2;

PIN OUT bool UB;
PIN OUT bool LB;

uint16 IR0;
uint16 IR1;

uint16 tmp_P;

bool clr_OD;
bool clr_CD;

REGISTER TClocked<bool> bset;
REGISTER TClocked<uint16> bpc;

int16 op1;
int16 op2;
int32 dst;
uint8 alu_mode;

```

Figure 5.30: AYK-14 Resource Declarations

5.5.2 *OPERATION* Definition

The AYK-14 architecture is capable of executing 16-bit and 32-bit instructions. For this reason the AYK-14 is modeled with two 16-bit instruction registers in LISA. The fetch *OPERATION* defined below in Figure 5.31, shows how the instructions are read from memory. The program counter, P, is not updated in the fetch *OPERATION* as was the case with the SRC design. The updating of the program counter must be done in a later *OPERATION* once the instruction has been decoded. This allows the processor to update the program counter by either one or two memory locations, depending on whether or not the current instruction to be executed is 32-bits or 16-bits respectively.

```

OPERATION fetch IN pipe.ONE {
    DECLARE {
        INSTANCE decode;
    }
    BEHAVIOR {
        int16 tmp_P0;

        if(bset) {
            tmp_P0 = bpc;
            bset = false;
        }
        else { tmp_P0 = P; }

        #ifdef LT_PROCESSOR_GENERATOR
        IR0 = prog_mem[ tmp_P0 ];
        IR1 = prog_mem[ tmp_P0 + 1 ];
        #else
        uint8 tmp1 = prog_mem[ tmp_P0 ];
        uint8 tmp2 = prog_mem[ tmp_P0 + 1 ];
        IR0 = (tmp2<<8) | tmp1;

        uint8 tmp3 = prog_mem[ tmp_P0 + 2 ];
        uint8 tmp4 = prog_mem[ tmp_P0 + 3 ];
        IR1 = (tmp4<<8) | tmp3;
        #endif
        tmp_P = tmp_P0;
    }
    ACTIVATION { decode }
}

```

Figure 5.31: AYK-14 Fetch Operation

The decode *OPERATION* of the AYK-14 introduces some new LISA constructs.

The *ENUM* keyword is used in LISA when modeling VLIW architectures. *ENUM* serves as an identifier to the processor to distinguish between the two separate coding trees of the VLIW architecture. The correct coding root is selected based on the coding of the current instruction being executed. The *ENUM* identifier is used inside a *SWITCH-CASE* construct only, as is depicted in Figure 5.32.[21]

SWITCH-CASE statements, as well as *IF-THEN-ELSE* statements, are LISA defined constructs and permit the combination of two or more *OPERATIONS* that differ only slightly to be defined within a single *OPERATION*. The correct conditional block is selected based on the coding of the current instruction. This allows the LISA defined *SWITCH-CASE* and *IF-THEN-ELSE* control structures to be evaluated at compile time,

whereas ANSI C if-then-else and switch-case structures are evaluated at run time and are dependent on the data being manipulated.[21]

```

INSTRUCTION decode IN pipe.ONE {
    DECLARE {
        ENUM format = { x16, x32 };
        GROUP insn_x16 = { nop || LR || LL || SBR || ZBR || SXI || LI ||
                            SI || IROR || DROR || AR || LLRS || LARD ||
                            LALS || LALD || LSU || LA || LC || CBR ||
                            LJ || LPR || LJE || LJNE || LJLS || LJGE ||
                            ORR || LSOR || SSOR || OCR || TCR || PR };
        GROUP insn_x32 = { LK || LK_simple || L || L_simple ||
                            LD || LD_simple || S || S_simple ||
                            SD || SD_simple || SZ || SZ_simple ||
                            BS || BS_simple || SM || SM_simple ||
                            A || A_simple || JLR_RX || JLR_RX_simple ||
                            JLR_RK || JLR_RK_simple || CK || CK_simple ||
                            ANDK || ANDK_simple || XJ || XJ_simple || C };
    }
    SWITCH (format) {
        CASE x16:
        {
            CODING AT ( P ) { IRO == insn_x16 }
            SYNTAX { insn_x16 }
            #ifdef LT_PROCESSOR_GENERATOR
                BEHAVIOR { P = tmp_P + 1; }
            #else
                BEHAVIOR { P = tmp_P + 2; }
            #endif
            ACTIVATION { insn_x16 }
        }
        CASE x32:
        {
            CODING AT ( P ) { (IRO == insn_x32=[16..31]) && (IR1 == insn_x32=[0..15]) }
            SYNTAX { insn_x32 }
            #ifdef LT_PROCESSOR_GENERATOR
                BEHAVIOR { P = tmp_P + 2; }
            #else
                BEHAVIOR { P = tmp_P + 4; }
            #endif
            ACTIVATION { insn_x32 }
        }
    }
}

```

Figure 5.32: AYK-14 Decode Operation

5.6 LISA Pitfalls

While modeling the various processor architectures several pitfalls of the LISA language were found. These included areas of the language that were poorly documented, or not documented at all, as well as some modeling techniques that produce

poor quality processors or that do not work at all. The next few paragraphs will detail a few of the more time consuming traps that were fallen into while learning the language.

The very first problem ran into was the modeling of processor memory. The SRC called for byte size big-endian memory. Trying to model this accurately in LISA took some time. The use of the LISA keyword *ENDIANESS* within the declaration of the memory was not working. It was noticed that no matter what endianess was defined within the LISA model, in memory the data was stored as little-endian. After a few weeks of different techniques to try and solve this problem, it was found that the .bend identifier needed to be used in the assembly program for the memory to be stored in big-endian format. The .bend identifier instructs the assembler to store the current instruction word or data in a big-endian format. The *ENDIANESS* LISA keyword instructs the processor to interpret the contents of memory as either big-endian or little-endian, depending on the accompanying identifier. Together these two identifiers model a completely working big-endian memory format. The use of the .bend identifier was not mentioned in the same section of the documentation with the *ENDIANESS* keyword. The .bend identifier was declared in a separate assembly directives file.

The next big stumbling block ran into while learning the LISA language involved pipelining. Pipeline stalls proved to be a hard feature to model with the LISA language. When a pipeline stage was stalled, the current stage was allowed to finish its calculations and the result was then stalled from input to the next stage by one cycle. The following clock cycle after the stall the result calculated prior to the stall was forwarded to the waiting stage. The stalled pipeline stage did not recalculate a new output value based on the input registers. This problem was remedied by the use of the LISA keyword *POLL*

used to define an *OPERATION* that needed to recalculate its value after a stall and before forwarding the results. This was a very poor documented area, and substantial modeling time was consumed to try and figure out why the pipeline stage would forward the incorrect value after a stall. This was a very important topic to grasp as none of the hazard detection or data forwarding hardware would have been possible without it.

Another problem with LISA was the way that an instruction with several syntaxes could be modeled. The same instruction could be modeled within multiple processes to allow for the different instruction syntaxes. If this approach is taken without the use of the *ALIAS* keyword then each separate operation is synthesized into RTL. This can greatly increase the chip size of the final processor. Using the *ALIAS* keyword allows the assembler to accept different syntaxes for the same instruction while only generating one instance of the instruction in RTL.

The Documentation Generator was also a source of a minor problem. After spending several days trying to get the Documentation Generator to run on the Very Long Instruction Word (VLIW) SRC architecture, it was discovered that LISA does not currently support Documentation Generator on VLIW architectures. This was again one of the poorly documented areas of the LISA language.

CHAPTER SIX

Verification and Implementation

This chapter describes the test plan used to verify the LISA processor models.

The generated VHDL code structure is also explored. The generated VHDL code is put into an embedded system and verified to be functional. The non-pipelined and pipelined SRC code is compared against Dr. Duren's handwritten models. They AYK-14 is compared against its original design.

6.1 Verification

To verify the SRC processor models designed in LISA, assembly language programs were written and tested within Processor Debugger. To ensure that the instructions were working, simple programs were written to test each instruction. An example of an assembly program that tests some of the SRC instructions on the non-pipelined architecture is given in Figure 4.3. Similarly, Figure 4.4 depicts a sample assembly program that tests the SRC instructions on the VLIW architecture. APPENDIX H gives more sample source code that was run through Processor Debugger to test the SRC instruction set on the various processor implementations.

The hazard detection and data forwarding capabilities of the pipelined SRC and pipelined VLIW SRC were also fully tested and verified within Processor Debugger prior to hardware generation. The source code to test the data hazard detection and data forwarding capabilities of the pipelined SRC is given below in Figure 6.1 and again in APPENDIX H. For the pipelined SRC only RAW hazards detection had to be tested and

verified. This was accomplished by writing an assembly program that contained read after write operations one to five clock cycles apart and verified that the correct pipelines received forwarded data, when appropriate, or stalled. Processor Debuggers was used to monitor the pipeline registers during this process.

```
// begin data
.data
.bend
// begin program code
.text
.bend
;
.global _start
.bend
;
_start:
    addi r1,r1,1    // r1 <- 1
    nop
    nop
    ld r2,0(r1)   // alu-alu dependency
    ld r3,0(r2)   // alu-load dependency
    sub r4,r3,r1 // load-alu dependency
    la r5,0(r4)   // alu-ladr dependency
    brlpl r6,r0,r5 // ladr-brl dependency
.end
```

Figure 6.1: SRC Hazard Detection Source File

The AYK-14 processor did not have interrupt capabilities and was therefore easier to model and verify with LISA. To ensure a working instruction set, an assembly language program was written to test each instruction modeled. A sample of the assembly program used is given in APPENDIX H.

6.2 Hardware Generation and System Implementation

6.2.1 CoWare VHDL Code Structure

The HDL code generated from a LISA project is broken into three entities, a register module, a memory module and a pipeline module. All of these modules are grouped into a top-level wrapper file that allows the processor design to be inserted into

an existing HDL project. Figure 6.2 below shows the VHDL directory structure created from the non-pipelined SRC LISA model. All HDL files created by Processor Generator are appended with _gen to easily distinguish them from non-LISA files. The files are created in a folder named VHDL or Verilog, within the current project directory.[18]

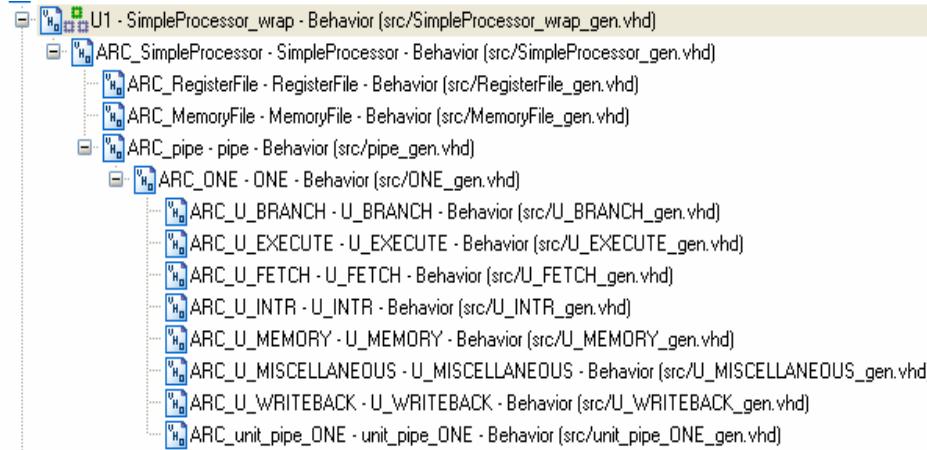


Figure 6.2: CoWare HDL Code Structure

The top-level wrapper file created from the SRC LISA project contains all input and output ports, including the memory ports, I/O pins, and the system clock and reset pins. This wrapper file is named SimpleProcessor_gen.vhd and is shown in Figure 6.2. The RegisterFile_gen.vhd file controls all data access to the clocked registers defined in the SRC architecture, this includes the general-purpose register file. The MemoryFile_gen.vhd file handles all accesses made to external memory; for the SRC design this would include accesses to program memory and to data memory. The pipe_gen.vhd file instantiates modules for each pipeline stage and pipeline registers defined.[18]

Since we have only one stage in the pipeline all LISA *OPERATIONS* are grouped into a single module, ONE_gen.vhd, and one of the eight other sub modules, grouped

according to the functional unit declarations made in the *RESOURCE* section of the SRC model (See Figure 5.22 SRC UNIT Definitions). For the pipelined processor models, each pipeline stage would be broken into its own module. For example, the five stage SRC pipeline would include five additional modules, FE_gen, DC_gen, EX_gen, MEM_gen and WB_gen. The five stage pipeline model will also contain four pipeline register modules, FE_DC_gen, DC_EX_gen, EX_MEM_gen, MEM_WB_gen. These modules contain all registers required to pass information between pipeline stages.[18]

6.2.2 SRC System Implementation

To test the generated HDL code, the files listed above were added into an existing ISE project. The ISE project structure, shown in Figure 6.3, consisted of a ROM for storing program memory, a RAM for data memory, a UART module and a top level wrapper to route control signals and all input/output ports between modules. The UART code was written for a previous class, and is a modified version of the UART described by Michael D. Ciletti in his textbook *Verilog HDL*.[23] The complete HDL source code for the modified UART is given in APPENDIX I.

The ISE project permitted a practical test of the SRC processors. The SRC was tested running an echo program. The presence of the UART allowed the SRC design to be loaded onto a Xilinx XUP Virtex II Pro FPGA [7], where the FPGA was connected to a host PC over an RS-232 link. The UART's presence allowed a user to communicate using HyperTerminal on the host PC with the processor and its executing program. The echo program used to test the interrupt capabilities of the SRC is given in APPENDIX H.

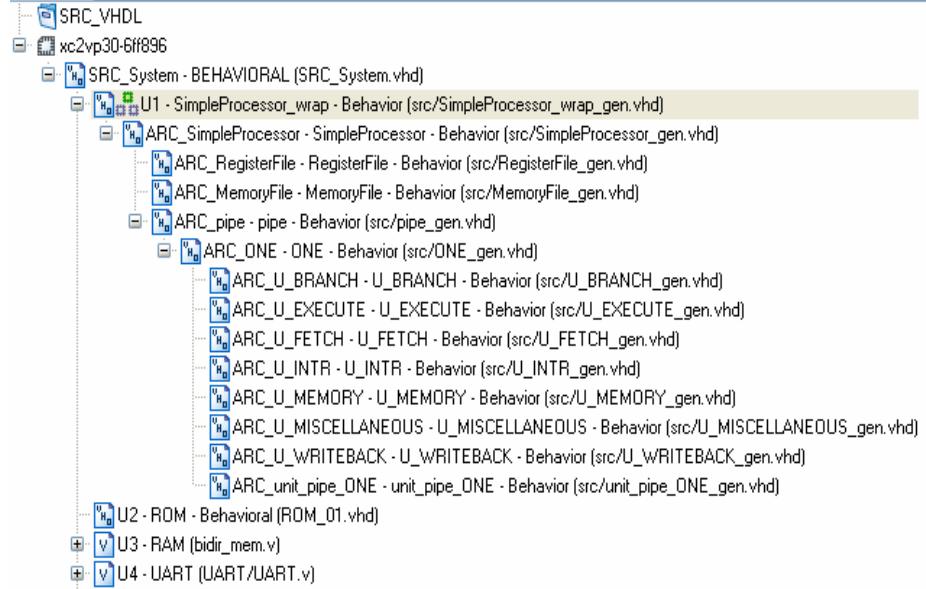


Figure 6.3: ISE Project Structure

6.2.3 AYK-14 System Implementation

The AYK-14 processor was incorporated into the same existing project as the SRC. The only difference was the absence of the UART module. The AYK-14 project did not implement interrupt capabilities and therefore could not be tested using the interrupt driven version of the echo program. The AYK-14 was instead tested on the FPGA with an LED switch program. The complete program is given in APPENDIX H. The LED switch program allows a user to flip on one of the four switches on the FPGA, reads in the status of the switches, shifts them to the right by one and turns the four LEDs on the FPGA accordingly.

6.3 Implementation Results

6.3.1 SRC Implementation Results

The LISA generated VHDL model of the SRC was compared to the handwritten 1-bus SRC model written by Professor Duren. It took approximately two months to

model the non-pipelined SRC within Processor Designer. The fundamental reason for such a lengthy modeling time can be attributed to the steep learning curve involved with a new language. Professor Duren has many years experience writing and teaching VHDL and Verilog, and was able to model the handwritten 1-bus SRC processor in approximately one month. Given an experienced LISA designer it is believed that the development times would have been similar. Table 6.1 depicts the results of the different SRC architectures modeled with LISA as well as the two architectures written in VHDL by Professor Duren.

Table 6.1: SRC Implementation Results

Processor	Language	SLOCs	Development Time	FPGA Slices	Clock Speed
SRC	VHDL	886	1 month	496	58.7 MHz
Pipelined SRC	VHDL	291	1 week	650	61.1 MHz
SRC	LISA	949	2 months	2750	32 MHz
Pipelined SRC	LISA	1176	1 month	3489	44.3 MHz
VLIW SRC	LISA	983	2 weeks	4925	32 MHz
Pipelined VLIW SRC	LISA	1412	1 week	8177	46.7 MHz

The non-pipelined SRC processor had a maximum clock speed of 32 MHz. The handwritten model had a maximum clock speed of 58.7MHz. The LISA non-pipelined SRC had a slower clock due to the fact that in LISA, a non-pipelined processor is actually implemented as a one-stage pipeline and each pipeline stage completes in its entirety during one clock cycle. This means that all the register reads and writes, all the memory access and the ALU operations happen in one clock cycle. This limits the LISA SRC processor to a clock speed no greater than the longest instruction execution time in the model. In the case of the handwritten code, the SRC instructions were broken into clock cycles. The number of clock cycles is instruction dependent and ranged from four to seven. Therefore the handwritten SRC model had faster clock speeds. For a better

comparison, it was esitmated that roughly half the instructions executed in a typical SRC program are memory accesses and require 7 clock cycles in the handwritten model, while the remainder of the instruction set requires an average of 5.5 clock cycles. This implies that the average clock speed per instruction on the handwritten SRC model was roughly 10MHz. Therefore, the LISA SRC would execute instructions approximately 3 times faster than the handwritten VHDL model.

The LISA non-pipelined SRC took 949 source lines of code and occupied 2750 slices on the FPGA. The handwritten SRC processor required 886 source lines of code to model and occupied 496 slices of logic on the FPGA. The 5.5 times larger logic required to implement the LISA model compared to the handwritten model is not surprising. More abstract modeling languages, like LISA, generally result in larger targets than lower-level languages, like VHDL or Verilog. This holds true for software modeling languages as well, where a C++ project would generally produce a larger target than would a well written assembly project.

The LISA SRC processor required a few more source lines of code to model than did the handwritten SRC processor. This is in part due to the inclusion of sections in the LISA model that do not contribute to the generated HDL code. The DOCUMENTATION sections are optional in a LISA model and contribute only to the generation of an Instruction Set Users Manual. The SYNTAX sections of the LISA model do not contribute to the HDL code and are used solely in the generation of the processor software tools.

6.3.2 Pipelined SRC Implementation Results

The LISA pipelined SRC was compared to the handwritten pipelined SRC with out complete hazard detection, written by Professor Duren. The LISA modeled pipelined SRC took less modeling time than did the LISA non-pipelined SRC model. The total modeling time was approximately one month. The shorter modeling time involved can be attributed to the learning curve of the LISA language. This compares to a modeling time of 1 week for the handwritten version of the pipelined SRC.

The LISA modeled pipelined SRC had a significantly faster clock than the LISA non-pipelined SRC. A maximum clock speed of 44.3 MHz was recorded for this architecture. The handwritten processor had a maximum clock speed of 61.1 MHz. The processor clocks for the two pipelined SRC models differs by a less than a factor of 1.5.

The amount of occupied logic slices for the LISA model on the FPGA was 3489 as compared to 650 for the handwritten pipelined version. Part of the discrepancies in size, can be contributed to the fact that the handwritten code does not include complete hazard detection and forwarding logic. The remaining size discrepancy can be attributed to the difference between the non-pipelined SRC models, The LISA pipelined SRC model took approximately 1176 lines of code to model completely. The handwritten pipelined SRC required only 291 lines of code. The majority of the difference in the modeling size was due to the inclusion of complete hazard detection and data forwarding in the LISA design.

6.3.3 VLIW Implementation Results

Once a working model for the non-pipelined SRC was created, designing a VLIW architecture version was a quick process. The non-pipelined SRC model was easily

modified to create a VLIW version of the SRC. The design time involved was approximately two weeks. The VLIW SRC had a clock speed of 32 MHz and occupied 4925 logic slices on the FPGA. For a better comparison with the other SRC architectures the actual instruction clock speed of this design was 64 MHz. This clock speed was calculated based on the fact that the VLIW SRC can execute two instructions per clock cycle. 983 lines of LISA code were required to model the VLIW SRC.

6.3.4 Pipelined VLIW Implementation Results

The pipelined SRC model was easily adapted to a VLIW pipelined SRC architecture within Processor Designer. Modeling the VLIW pipelined SRC took approximately one week. The pipelined VLIW architecture occupied by far the most FPGA chip space of all the SRC architectures at 8177 slices, but was still able to fit on the FPGA, which contains 9280 available logic slices. The pipelined VLIW architecture also had the fastest processor clock at a speed of 46.7 MHz. For comparison purposes the actual instruction rate of the pipelined VLIW is closer to 95 MIPS. The LISA Pipelined VLIW SRC processor also required more source lines of code to model at approximately 1412. This is due to the fact that the hazard detection and data forwarding becomes more complex with a pipelined VLIW architecture as we are no longer concerned with only RAW hazards, we must also be concerned with WAW and WAR hazards.

6.3.5 AYK-14 Implementation Results

The AYK-14 processor was the final processor architecture designed using the CoWare Inc. tool suite. The subset of instructions from the AYK-14 processor instruction set was accurately modeled using Processor Designer in approximately one

week. The AYK-14 embedded processor occupied 2808 logic slices on the the FPGA and had a processor clock speed of 29.8 MHz. The LISA modeled AYK-14 processor required 2620 lines of source code to model completely. The current AYK-14 embedded processor has a clock speed of 18 MHz and was originally modeled using schematic capture and an ASIC standard cell library. The original AYK-14 model required roughly 500 schematic pages and required an estimated 3-5 man-years to model. Table 6.2 below depicts the comparison between the current AYK-14 embedded processor and the LISA modeled AYK-14 processor. The LISA AYK-14 processor only modeled roughly 10%, or 41 instructions, of the AYK-14 instructions set.

Table 6.2: AYK-14 Implementation Results

Processor	Language	SLOCs	Development Time	FPGA Slices	Clock Speed
AYK-14 (Current)	Schematic capture	500 pages	3-5 man-years (est.)		18 MHz
AYK-14	LISA	2620	1 week	2808	29.8 MHz

CHAPTER SEVEN

Conclusion

It is evident from this evaluation of the CoWare Inc. tool suite that LISA is a very beneficial ANSI-C based embedded processor ADL. The LISA language had a few weak points but overall, once the language was learned, modeling of different processors and architectures was an easy and rapid process. This ease of use comes with a tradeoff, as the generated RTL from a LISA model will invariably be of lower quality than that of a handwritten HDL model, where this quality can be measured in terms of clock speed and FPGA circuitry utilization.

The LISA tools perform best with pipelined architectures. In the test cases used in this thesis, the hardware generated by LISA operated about 50% slower than the hardware generated from the handwritten model and required about 5 times more FPGA logic than the handwritten model. A significant part of these differences might be due to the fact that the LISA models incorporated hazard detection and data forwarding where the handwritten model did not. Taking this into account, clock speed and logic requirements for the LISA models are probably still somewhat slower and larger than handwritten code, but this is not unexpected when using a higher-level language.

The learning curve involved with the LISA language is similar to that of Verilog or VHDL. However, once a processor is modeled, the user gets a processor-specific assembler, disassembler, instruction set simulator/debugger, and instruction set documentation guide at almost no extra effortt. These products are the chief advantage of using the LISA language.

The CoWare tool's capability to accurately model a non-pipelined, a pipelined, a VLIW, and a pipelined VLIW architecture make it ideal for classroom utilization in the learning of processor architecture and design. The LISA language's ANSI-C-like structure should allow students to be able to design different ISA processors on numerous hardware architectures with no prior knowledge of an HDL. The ability to get a full suite of software tools as well as the RTL hardware from one common LISA model would allow students to easily explore the effect that different processor architectures would have on the speed of their applications. This again makes the tool very desirable for classroom applications. Given an instructor experienced with the LISA language, a 3-credit hour class should be adequate in allowing students to learn the LISA language and explore the design of processors. It is estimated that the students would have time to learn the language, study various architectures, and complete a final project that includes the design and verification of a unique processor.

The ease in which the language was able to model a CISC processor, like the AYK-14, shows the tool's usefulness in updating and maintaining legacy processors. Future research should be done to explore LISA's ability to model superscalar and DSP processor architectures. The CoWare Inc. C Compiler and the Instruction Set Designer should also be studied for incorporated into the classroom environment.

APPENDICES

APPENDIX A

Non-Pipelined SRC Code

main.lisa

```
*****SRC Implementation*****
*
*          Processor Resource Description
*
*****/
```

```
#include "defines.h"
#include "memory_if.h"
#include "memory_cfg.h"

// Specify a version string which is contained later
// in the generated tools
VERSION("SRC, 2006.1.1")

*****Processor Resource Description*****
*
*          RESOURCE
{
    // Harvard memory map for program and data memory
    MEMORY_MAP
    {
        RANGE(PMEM_START, PMEM_END) -> prog_mem[(31..0)];
        RANGE(DMEM_START, DMEM_END) -> data_mem[(31..0)];
    }

    // 0x10000 32bit words of program memory
    // FLAGS are set to R|X meaning that prog_mem is readable and
    // executable
    RAM uint32 prog_mem
    {
        SIZE(PMEM_SIZE);
        BLOCKSIZE(32);
        FLAGS(R|X);
        ENDIANESS(BIG);
    };

    // 0x10000 32bit words of data memory
    // FLAGS are set to R|W meaning that data mem is
    // readable and writeable
    RAM uint32 data_mem
    {
        SIZE(DMEM_SIZE);
        BLOCKSIZE(32);
    };
}
```

```

FLAGS (R|W);
ENDIANESS (BIG);
};

// Register file ( 32 General Purpose Registers )
REGISTER TClocked<int32> GPR[0..31];

// Program Counter ( PC ) Register
PROGRAM_COUNTER TClocked<uint32> PC;

// 32-bit instruction register ( IR )
uint32 IR;

// Branch Program Counter (BPC) Register
// and Status Flag (BSET)
REGISTER TClocked<uint32> BPC;
REGISTER TClocked<bool> BSET;

// ALU resources (LISA signals, not REGISTERS)
int32 src1; // ALU input operand 1
int32 src2; // ALU input operand 2
int32 result; // ALU output result
unsigned char alu_mode; // selects ALU operation

// Interface to Memory Registers (LISA signal not REGISTER)
uint32 MD; // Data Signal to Memory

// 32-bit Interrupt Program Counter ( IPC ) Register
REGISTER TClocked<uint32> IPC;
// 32-bit Interrupt Information ( II ) Register
REGISTER TClocked<uint32> II;
// 1-bit interrupt enable ( IE ) flag
REGISTER TClocked<bool> IE;
REGISTER TClocked<bool> RUN;

// Interrupt Request and Acknowledge PINS
PIN IN bool ireq;
PIN OUT bool iack;

PIPELINE pipe = { ONE };

// UNIT declarations for RTL generation
UNIT U_FETCH IN pipe.ONE {
    OPERATIONS (fetch);
    REGISTERS (IR, PC);
};
UNIT U_EXECUTE IN pipe.ONE {
    OPERATIONS (alu, alui, alur, aluu, shift);
    REGISTERS (src1, src2, result, alu_mode);
};
UNIT U_BRANCH IN pipe.ONE {
    OPERATIONS (Branch_Logic, branchl);
    REGISTERS (BPC, BSET);
};
UNIT U_MEMORY IN pipe.ONE {

```

```

OPERATIONS (ld, la, st, ldr, lar, str);
REGISTERS (MD);
};

UNIT U_WRITEBACK IN pipe.ONE {
    OPERATIONS (writeback);
};

UNIT U_MISCCELLANEOUS IN pipe.ONE {
    OPERATIONS (stop);
    REGISTERS (RUN);
};

UNIT U_INTR IN pipe.ONE {
    OPERATIONS (een, edi, rfi, svi, ri);
    REGISTERS (IPC, II, IE);
};

}

/***** Processor Instruction Set Description *****/
*
*          Processor Instruction Set Description
*
***** /


OPERATION reset {
    BEHAVIOR
    {
        // C Behavior Code to reset all the processor
        // resources to a well defined state

        // Zero out General Purpose Register
        int i;
        for(i = 0; i < 32; i++)
        {
            GPR[i] = 0;
        }

        // Set program counter to the program entry point
        PC = LISA_PROGRAM_COUNTER;

        // Set Branch Program Counter and Branch Set Flag to defined states
        BPC = 0;
        BSET = false;
        RUN = true;
        IE = false;
        iack = false;

        PIPELINE(pipe).flush();
#pragma analyze(off)
        // declare an text-output channel with the name "debug",
        // this channel can be used to print debug messages into
        // a window that is created for each channel
        decl_out_channel(DBGOUT,"debug");
#pragma analyze(on)
    }
}

```

```

OPERATION main {
    // Operation main is triggered every control step (clock)
    DECLARE {
        INSTANCE fetch;
    }
    BEHAVIOR {
        PIPELINE(pipe).execute();
        PIPELINE(pipe).shift();
    }
    ACTIVATION { fetch }
}

OPERATION fetch IN pipe.ONE {
    DECLARE
    {
        INSTANCE decode;
    }
    BEHAVIOR {
        uint32 temp_PC;

        // Check RUN bit to see if Machine has been halted
        if(!RUN) { PC += 0; }
        else {
            if(BSET) { temp_PC = BPC; }
            else { temp_PC = PC; }

            // If we have an interrupt request and if interrupts are enabled
            // Save the PC and load the Interrupt Vector
            if(ireq && IE)
            {
                uint32 iack_data;
                iack = true;

                PMEM_LD(iack_data,temp_PC); // data mem read from highest word
                                            // PMEM_LD(tmp1,temp_PC);
                // prog_bus.read(temp_PC,&iack_data);
                II = 0x0000ffff & iack_data;
                PC = 0x000000ff & (iack_data >> 16);
                IPC = temp_PC;
                IR = 0;
            }
            else
            {
                iack = false;
                // Get an instruction from the program memory at address
                // temp_PC;
                PMEM_LD(IR,temp_PC);
                // prog_bus.read(temp_PC,&IR);

                // No, update the program counter to the next address
                if (PC >= 0xffff) {
#pragma analyze(off)
                    fprintf(stderr,"Possible Model Fault -- PC goes beyond valid
range and wraps to 0\n");
# pragma analyze(on)
                    temp_PC = 0;

```

```

        }
    else
    {
#ifndef LT_PROCESSOR_GENERATOR
        // Increment the program counter to the next address
        temp_PC += 4; // For RTL generation
#else
        temp_PC += 1;
#endif
    }
    PC = temp_PC;
}
}

ACTIVATION { decode }

OPERATION decode IN pipe.ONE {
DECLARE {
    GROUP instruction = { nop || stop ||
                          alui || alur || aluu ||
                          branch || branchl ||
                          ld || ld_simple ||
                          la || la_simple ||
                          st || st_simple ||
                          ldr || lar || str ||
                          shift || shift_count || shift_reg ||
                          ri || svi || rfi || edi || een };
}

// The current instruction word is in "IR"
// The current instruction word is at "PC"
CODING AT ( PC ) { IR == instruction }

// The complete syntax description is implemented in "instruction"
SYNTAX { instruction }
BEHAVIOR {
    char op = IR >> 27 & 0x1f;
    if((op != BRANCH) && (op != BRANCHL)) { BSET = false; }
}
// Execute the instruction that was selected by
// the decoder.
ACTIVATION { instruction }
}

```

alu.lisa

```

*****
*                                         *
*          ALU Implementation          *
*                                         *
*****
```

```

***** /*****
#include "defines.h"

/***** *****/
*          ALU Immediate
* 31..27  26..22 21..17 16      ...      0  *
* -----|-----|-----|-----|-----|-----|-----*
* | Op   | ra    | rb    |           c2           |   *
* -----|-----|-----|-----|-----|-----|-----*
*          *
***** /*****

OPERATION alui IN pipe.ONE {
    DECLARE {
        GROUP opcode = { addi || ori || andi };
        GROUP ra,rb = { reg };
        GROUP c2      = { imm17 };
        INSTANCE alu;
        INSTANCE writeback;
    }
    CODING { opcode ra rb c2 }
    SYNTAX { opcode ~" " ra "," rb "," c2 }
    BEHAVIOR {
        src1 = GPR[rb]; // Load GPR[rb] into src1
        src2 = SIGN_EXTEND_15(c2); // Put immediate into src2
        alu_mode = opcode; // Set the ALU mode
    }
    ACTIVATION { alu, writeback } // Activate the ALU & writeback
    DOCUMENTATION("ALU IMMEDIATE") { ALU Immediate Instructions }
}

OPERATION addi {
    // addi opcode := 13
    CODING { 0b01101 }
    SYNTAX { "addi" }
    EXPRESSION { ALU_ADD }
    DOCUMENTATION { Add rb to immediate constant, and put result into
                    ra }
}

OPERATION ori {
    // ori opcode := 23
    CODING { 0b10111 }
    SYNTAX { "ori" }
    EXPRESSION { ALU_OR }
    DOCUMENTATION { "OR" rb and immediate constant, and put result into
                    ra }
}

OPERATION andi {
    // andi opcode := 21
    CODING { 0b10101 }
    SYNTAX { "andi" }
    EXPRESSION { ALU_AND }
    DOCUMENTATION { "AND" rb and immediate constnat, and put result into
                    ra }
}

```

```

/*
 *          ALU Register
 *  31..27  26..22 21..17 16..12 11      ...      0  *
 *  -----|-----|-----|-----|-----|-----|-----|
 *  | Op   | ra    | rb    | rc    |      unused      |   *
 *  -----|-----|-----|-----|-----|-----|-----|
 *                                              *
 *****/
OPERATION alur IN pipe.ONE {
    DECLARE {
        GROUP opcode = { add || sub || and || or };
        GROUP ra,rb,rc = { reg };
        INSTANCE alu;
        INSTANCE writeback;
    }

    CODING { opcode ra rb rc 0b0[12] }
    SYNTAX { opcode ~" " ra "," rb "," rc }
    BEHAVIOR {
        // Load register into resource operand 1 and 2
        src1 = GPR[rb];
        src2 = GPR[rc];
        alu_mode = opcode;
    }
    ACTIVATION { alu, writeback } // Activate the alu & writeback
    DOCUMENTATION("ALU REGISTER") { ALU Register Instructions }
}

OPERATION add {
    // add opcode := 12
    CODING { 0b01100 }
    SYNTAX { "add" }
    EXPRESSION { ALU_ADD }
    DOCUMENTATION { Add rb to rc, and put result into ra }
}

OPERATION sub {
    // sub opcode := 14
    CODING { 0b01110 }
    SYNTAX { "sub" }
    EXPRESSION { ALU_SUB }
    DOCUMENTATION { Subtract rc from rb, and put result into ra }
}

OPERATION and {
    // and opcode := 20
    CODING { 0b10100 }
    SYNTAX { "and" }
    EXPRESSION { ALU_AND }
    DOCUMENTATION { "AND" rb and rc, and put result into ra }
}

OPERATION or {

```

```

// or opcode := 22
CODING { 0b10110 }
SYNTAX { "or" }
EXPRESSION { ALU_OR }
DOCUMENTATION { "OR" rb and rc, and put result into ra }
}

/********************* Neg/Not Register *****/
*           Neg/Not Register          *
*   31..27  26..22 21 ... 17 16..12 11    ...      0  *
*   -----|-----|-----|-----|-----|-----|-----|-----*
*   | Op  | ra  | unused | rc  |     unused    |      |
*   -----|-----|-----|-----|-----|-----|-----|-----*
*           *****

OPERATION aluu IN pipe.ONE {
DECLARE {
    GROUP opcode = { neg || not };
    GROUP ra,rc = { reg };
    INSTANCE alu;
    INSTANCE writeback;
}
CODING { opcode ra 0b0[5] rc 0b0[12] }
SYNTAX { opcode ~" " ra "," rc }
BEHAVIOR {
    src1 = 0;
    src2 = GPR[rc];
    alu_mode = opcode;
}
ACTIVATION { alu, writeback } // Activate the alu & writeback
DOCUMENTATION ("ALU UNARY") { ALU Unary Instructions }
}

OPERATION neg {
// neg opcode := 15
CODING { 0b01111 }
SYNTAX { "neg" }
EXPRESSION { ALU_NEG }
DOCUMENTATION { Place 2's compliment negative of rc into ra }
}

OPERATION not {
// not opcode := 24
CODING { 0b11000 }
SYNTAX { "not" }
EXPRESSION { ALU_NOT }
DOCUMENTATION { Place logical "NOT" of rc into ra }
}

/********************* ALU Definition *****/
*           ALU Definition          *
*           *****

OPERATION alu IN pipe.ONE {
BEHAVIOR {

```

```

        switch(alu_mode) {
            case ALU_ADD: // Add
                result = src1 + src2;
                break;
            case ALU_SUB: // Sub
                result = src1 - src2;
                break;
            case ALU_AND: // And
                result = src1 & src2;
                break;
            case ALU_OR: // Or
                result = src1 | src2;
                break;
            case ALU_NEG: // Neg
                result = -src2;
                break;
            case ALU_NOT: // Not
                result = ~src2;
                break;
            case ALU_SHR: // Shift Right
                result = (uint32)src1 >> src2;
                break;
            case ALU_SHRA: // Shift Right Arithmetic
                result = src1 >> src2;
                break;
            case ALU_SHL: // Shift Left
                result = src1 << src2;
                break;
            case ALU_SHC: // Shift Circular
                result = (src1 << src2) | ((uint32)src1 >> (32-src2));
                break;
            default:
                result = 0;
                break;
        }
    }
}

```

```

OPERATION writeback IN pipe.ONE {
    DECLARE { REFERENCE ra; }
    BEHAVIOR {
        // uint32 dest = ra;
        char op = IR >> 27 & 0x1f;
        if(op == LD || op == LDR)
            { GPR[ ra ] = MD; }
        /* else if(op == BRANCHL)
           { GPR[ dest ] = PC; } */
        else { GPR[ ra ] = result; }
    }
}

```

shift.lisa

```
*****
*
```

```

*
*          SHIFT Implementation           *
*
***** ****
#include "defines.h"

/*****
*          shift immediate           *
*  31..27  26..22 21..17 16      ...    5 4 ... 0   *
*  -----   -----
* | Op   |   ra  |   rb  | (c3)  unused   | count  |   *
* -----   -----
*          shift register           *
*  31..27  26..22 21..17 16..12 11 ... 5 4 ... 0   *
*  -----   -----
* | Op   |   ra  |   rb  | rc |(c3) unused| 00000 |   *
* -----   -----
***** ****
OPERATION shift IN pipe.ONE {
    DECLARE
    {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP c3 = { imm5 };
        GROUP ra,rb,rc = { reg };
        INSTANCE alu, writeback;
    }
    CODING { opcode ra rb rc 0bx[7] c3 }
    SYNTAX { opcode ~" " ra "," rb "," rc "," c3 }
    BEHAVIOR {
        if(c3) { src2 = c3; }
        else { src2 = GPR[rc]; }
        src1 = GPR[rb];
        alu_mode = opcode;
    }
    ACTIVATION { alu, writeback }
}

ALIAS OPERATION shift_count IN pipe.ONE {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP c3 = { imm5 };
        GROUP ra,rb = { reg };
        INSTANCE alu, writeback;
    }
    CODING { opcode ra rb 0b0[12] c3 }
    SYNTAX { opcode ~" " ra "," rb "," c3 }
    BEHAVIOR {
        src2 = c3;
        src1 = GPR[rb];
        alu_mode = opcode;
    }
    ACTIVATION { alu, writeback }
    SWITCH(opcode) {
        CASE shr:
            { DOCUMENTATION("SHR") { shift rb right into ra by constant
c3 } }
    }
}

```

```

CASE shra:
{ DOCUMENTATION("SHRA") { shift rb right with sign-extend into
                           ra by constant c3 } }
CASE shl:
{ DOCUMENTATION("SHL") { shift rb left into ra by constant c3 } }
CASE shc:
{ DOCUMENTATION("SHC") { shift rb left circularly into ra by
                           constant c3 } }
}

ALIAS OPERATION shift_reg IN pipe.ONE {
DECLARE {
    GROUP opcode = { shr || shra || shl || shc };
    GROUP ra,rb,rc = { reg };
    INSTANCE alu, writeback;
}
CODING { opcode ra rb rc 0b0[12] }
SYNTAX { opcode ~" " ra "," rb "," rc }
BEHAVIOR {
    src2 = GPR[rc];
    src1 = GPR[rb];
    alu_mode = opcode;
}
ACTIVATION { alu, writeback }
SWITCH(opcode) {
    CASE shr:
        { DOCUMENTATION("SHR") { shift rb right into ra by count in rc;
                                   c3 is 0 } }
    CASE shra:
        { DOCUMENTATION("SHRA") { shift rb right with sign-extend into ra
                                   by count in rc; c3 is 0 } }
    CASE shl:
        { DOCUMENTATION("SHL") { shift rb left into ra by count in rc; c3
                                   is 0 } }
    CASE shc:
        { DOCUMENTATION("SHC") { shift rb left circularly into ra by
                                   count in rc; c3 is 0 } }
}
}

OPERATION shr {
// opcode := 26
CODING { 0b11010 }
SYNTAX { "shr" }
EXPRESSION { ALU_SHR }
}

OPERATION shra {
//opcode := 27
CODING { 0b11011 }
SYNTAX { "shra" }
EXPRESSION { ALU_SHRA }
}

OPERATION shl {

```

```

// opcode := 28
CODING { 0b11100 }
SYNTAX { "shl" }
EXPRESSION { ALU_SHL }
}

OPERATION shc {
// opcode := 29
CODING { 0b11101 }
SYNTAX { "shc" }
EXPRESSION { ALU_SHC }
}

```

branch.lisa

```

***** Branch Instructions *****
#include "defines.h"

***** Branch *****
*      31..27 26..22 21..17 16..12 11      ...      3 2..0  *
* -----|-----|-----|-----|-----|-----|-----|-----|-----|
* | Op   |       | rb    | rc    | (c3) unused | c3   |   *
* -----|-----|-----|-----|-----|-----|-----|-----|-----|
*      *                                         *
***** Branch IN pipe.ONE {                         *
DECLARE {                                         *
    GROUP rb,rc = { reg };                         *
    GROUP c3 = { brnv || br || brzr || brnz || brpl || brmi }; *
    INSTANCE Branch_Logic;                         *
}                                         *
// opcode = 8                                     *
CODING { 0b01000 0b0[5] rb rc 0b0[9] c3 }     *
SYNTAX { c3 }                                     *
BEHAVIOR {                                         *
    // Branch_Logic();                           *
}                                         *
ACTIVATION { Branch_Logic }                      *
DOCUMENTATION("BRANCH") { Branch to target in rb if rc satisfies *
    condition c3 }                                *
}

OPERATION brnv {                                 *
DECLARE {                                         *
    REFERENCE rb,rc;                            *
}                                         *
// cond = 0                                     *
CODING { 0b000 }                                *
SYNTAX { "brnv" }                               *
EXPRESSION { BRNV }                            *

```

```

DOCUMENTATION { branch never }

}

OPERATION br {
    DECLARE {
        REFERENCE rb,rc;
    }
    // cond = 1
    CODING { 0b001 }
    SYNTAX { "br" ~" " rb }
    EXPRESSION { BR }
    DOCUMENTATION { branch unconditionally to rb }
}

OPERATION brzr {
    DECLARE {
        REFERENCE rb,rc;
    }
    // cond = 2
    CODING { 0b010 }
    SYNTAX { "brzr" ~" " rb "," rc }
    EXPRESSION { BRZR }
    DOCUMENTATION { branch to rb if rc is zero }
}

OPERATION brnz {
    DECLARE {
        REFERENCE rb,rc;
    }
    // cond = 3
    CODING { 0b011 }
    SYNTAX { "brnz" ~" " rb "," rc }
    EXPRESSION { BRNZ }
    DOCUMENTATION { branch to rb if rc is non-zero }
}

OPERATION brpl {
    DECLARE {
        REFERENCE rb,rc;
    }
    // cond = 4
    CODING { 0b100 }
    SYNTAX { "brpl" ~" " rb "," rc }
    EXPRESSION { BRPL }
    DOCUMENTATION { branch to rb if rc is positive or zero (sign is plus) }
}

OPERATION brmi {
    DECLARE {
        REFERENCE rb,rc;
    }
    // cond = 5
    CODING { 0b101 }
    SYNTAX { "brmi" ~" " rb "," rc }
    EXPRESSION { BRMI }
    DOCUMENTATION { branch to rb if rc is negative (sign is minus) }
}

```

```

}

/*
*          Branch Link
*  31..27  26..22 21..17 16..12 11      ...      3 2..0  *
*  -----|-----|-----|-----|-----|-----|-----|-----|
*  | Op   | ra    | rb    | rc    | (c3) unused | c3   | *
*  -----|-----|-----|-----|-----|-----|-----|-----|
*          *
*/
OPERATION branchl IN pipe.ONE {
    DECLARE {
        GROUP ra,rb,rc = { reg };
        GROUP c3 = { brlnv || brl || brlzs || brlnz || brlpl || brlmi };
        INSTANCE Branch_Logic;
    }
    // opcode = 9
    CODING { 0b01001 ra rb rc 0b0[9] c3 }
    SYNTAX { c3 }
    BEHAVIOR {
        GPR[ra] = PC;
    }
    ACTIVATION { Branch_Logic }
    DOCUMENTATION("BRANCH LINK") { Branch to rb if rc satisfies c3,
        and save PC in ra }
}

OPERATION brlnv {
    DECLARE {
        REFERENCE ra,rb,rc;
    }
    // cond = 0
    CODING { 0b000 }
    SYNTAX { "brlnv" ~" " ra }
    EXPRESSION { BRNV }
    DOCUMENTATION { Do not branch, but save PC in ra }
}

OPERATION brl {
    DECLARE {
        REFERENCE ra,rb,rc;
    }
    // cond = 1
    CODING { 0b001 }
    SYNTAX { "brl" ~" " ra "," rb }
    EXPRESSION { BR }
    DOCUMENTATION { Branch unconditionally to rb, and save PC in ra }
}

OPERATION brlzs {
    DECLARE {
        REFERENCE ra,rb,rc;
    }
    // cond = 2
    CODING { 0b010 }
    SYNTAX { "brlzs" ~" " ra "," rb "," rc }
}

```

```

EXPRESSION { BRZR }
DOCUMENTATION { Branch to rb if rc is zero, and save PC in ra }
}

OPERATION brlnz {
DECLARE {
    REFERENCE ra,rb,rc;
}
// cond = 3
CODING { 0b011 }
SYNTAX { "brlnz" ~" " ra "," rb "," rc }
EXPRESSION { BRNZ }
DOCUMENTATION { Branch to rb if rc is non-zero, and save PC in ra }
}

OPERATION brlpl {
DECLARE {
    REFERENCE ra,rb,rc;
}
// cond = 4
CODING { 0b100 }
SYNTAX { "brlpl" ~" " ra "," rb "," rc }
EXPRESSION { BRPL }
DOCUMENTATION { Branch to rb if rc is positive, and save PC in ra }
}

OPERATION brlmi {
DECLARE {
    REFERENCE ra,rb,rc;
}
// cond = 5
CODING { 0b101 }
SYNTAX { "brlmi" ~" " ra "," rb "," rc }
EXPRESSION { BRMI }
DOCUMENTATION { Branch to rb if rc is negative, and save PC in ra }
}

/*****************
*               *
*      BRANCH LOGIC      *
*               *
*****************/
OPERATION Branch_Logic IN pipe.ONE {
DECLARE {
    REFERENCE c3;
    REFERENCE rb,rc;
}
BEHAVIOR {
    char br_cond;

    br_cond = c3;

    switch(br_cond) {
        case BRNV: // branch never
            BSET = false;
            break;
    }
}
}

```

```

        case BR: // branch always
            BPC = (unsigned int)GPR[rb];
            BSET = true;
            break;
        case BRZR: // branch when condition(rc) is zero
            if((signed int)GPR[rc] == 0) {
                BPC = (unsigned int)GPR[rb];
                BSET = true;
            }
            else { BSET = false; }
            break;
        case BRNZ: // branch when condition(rc) is non-zero
            if((signed int)GPR[rc] != 0) {
                BPC = (unsigned int)GPR[rb];
                BSET = true;
            }
            else { BSET = false; }
            break;
        case BRPL: // branch when condition(rc) is positive (>= 0)
            if((signed int)GPR[rc] >= 0) {
                BPC = (unsigned int)GPR[rb];
                BSET = true;
            }
            else { BSET = false; }
            break;
        case BRMI: // branch when condition(rc) is negative (< 0)
            if((signed int)GPR[rc] < 0) {
                BPC = (unsigned int)GPR[rb];
                BSET = true;
            }
            else { BSET = false; }
            break;
        default:
            BSET = false;
            break;
    }
}
}

```

load_store.lisa

```

*****
*
*          Loads and Stores
*
*****
#include "defines.h"
#include "memory_if.h"

*****
*          LDR / STR / LAR
*      31..27  26..22 21           ...           0   *

```

```

* ----- *  

* | Op | ra | c1 | *  

* ----- *  

*----- *  

*****/*  

OPERATION ldr IN pipe.ONE {  

    DECLARE {  

        GROUP ra = { reg };  

        LABEL c1;  

        INSTANCE alu, mem_access_load, writeback;  

    }  

    // ldr opdcode := 2  

    CODING { 0b00010 ra c1=0bx[22] }  

    SYNTAX { "ldr" ~" " ra "," SYMBOL(((c1=#$22))+CURRENT_ADDRESS)=$X32)  

    }  

    BEHAVIOR {  

        src1 = PC;  

        src2 = SIGN_EXTEND_10(c1);  

        alu_mode = ALU_ADD;  

    }  

    ACTIVATION { alu, mem_access_load, writeback }  

    DOCUMENTATION("LDR") { Load from a relative address }  

}  

OPERATION lar IN pipe.ONE {  

    DECLARE {  

        GROUP ra = { reg };  

        LABEL c1;  

        INSTANCE alu, writeback;  

    }  

    // lar opdcode := 6  

    CODING { 0b00110 ra c1=0bx[22] }  

    SYNTAX { "lar" ~" " ra "," SYMBOL(((c1=#$22))+CURRENT_ADDRESS)=$X32)  

    }  

    BEHAVIOR {  

        src1 = PC;  

        src2 = SIGN_EXTEND_10(c1);  

        alu_mode = ALU_ADD;  

    }  

    ACTIVATION { alu, writeback }  

    DOCUMENTATION ("LAR") { Load value of relative address into ra }  

}  

OPERATION str IN pipe.ONE {  

    DECLARE {  

        GROUP ra = { reg };  

        LABEL c1;  

        INSTANCE alu, mem_access_store;  

    }  

    // str opdcode := 4  

    CODING { 0b00100 ra c1=0bx[22] }  

    SYNTAX { "str" ~" " ra "," SYMBOL(((c1=#$22))+CURRENT_ADDRESS)=$X32)  

    }  

    BEHAVIOR {  

        src1 = PC;  

        src2 = SIGN_EXTEND_10(c1);  

    }
}

```

```

        alu_mode = ALU_ADD;
        // MD = GPR[ra];
    }
ACTIVATION { alu, mem_access_store }
DOCUMENTATION ("STR") { Store into relative address }
}

/*********************************************
*                                     *
*   31..27  26..22 21..17 16          ...      0   *
*   -----|-----|-----|-----|-----|-----|-----*
*   | Op  | ra   | rb   |           c2           |   *
*   -----|-----|-----|-----|-----|-----|-----*
*                                     *
*****************************************/
OPERATION ld IN pipe.ONE {
DECLARE
{
    GROUP ra,rb = { reg };
    GROUP c2 = { imm17 };
    INSTANCE alu, mem_access_load, writeback;
}
// ld opcode := 1
CODING { 0b00001 ra rb c2 }
SYNTAX { "ld" ~" " ra "," c2 "(" rb ")" }
BEHAVIOR {
    if(rb) { src2 = GPR[rb]; }
    else { src2 = 0; }
    src1 = SIGN_EXTEND_15(c2);
    alu_mode = ALU_ADD;
}
ACTIVATION { alu, mem_access_load, writeback }
DOCUMENTATION("LD") { Load from displacement address }
}

ALIAS OPERATION ld_simple IN pipe.ONE {
DECLARE {
    GROUP ra = { reg };
    GROUP c2 = { imm17 };
    INSTANCE alu, mem_access_load, writeback;
}
CODING { 0b00001 ra 0b0[5] c2 }
SYNTAX { "ld" ~" " ra "," c2 }
BEHAVIOR {
    src2 = 0;
    src1 = SIGN_EXTEND_15(c2);
    alu_mode = ALU_ADD;
}
ACTIVATION { alu, mem_access_load, writeback }
DOCUMENTATION("LD") { Load from absolute address; rb is register 0 }
}

/*********************************************
*                                     *
*   31..27  26..22 21..17 16          ...      0   *
*   -----|-----|-----|-----|-----|-----|-----*
*   |-----|-----|-----|-----|-----|-----|-----*
*                                     *
*****************************************/

```

```

* ----- *  

* | Op | ra | rb | c2 | *  

* ----- *  

* ----- *  

*****  

OPERATION la IN pipe.ONE {
    DECLARE
    {
        GROUP ra,rb = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu, writeback;
    }
    // la opcode := 5
    CODING { 0b00101 ra rb c2 }
    SYNTAX { "la" ~" " ra "," c2 "(" rb ")" }
    BEHAVIOR {
        if(rb) { src2 = GPR[rb]; }
        else { src2 = 0; }
        src1 = SIGN_EXTEND_15(c2);
        alu_mode = ALU_ADD;
    }
    ACTIVATION { alu, writeback }
    DOCUMENTATION("LA") { Load value of displacement address into ra }
}

ALIAS OPERATION la_simple IN pipe.ONE {
    DECLARE {
        GROUP ra = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu, writeback;
    }
    CODING { 0b00101 ra 0b0[5] c2 }
    SYNTAX { "la" ~" " ra "," c2 }
    BEHAVIOR {
        src2 = 0;
        src1 = SIGN_EXTEND_15(c2);
        alu_mode = ALU_ADD;
    }
    ACTIVATION { alu, writeback }
    DOCUMENTATION("LA") { Load value of absolute address into ra;
        rb is register 0 }
}

*****  

* ST *  

* 31..27 26..22 21..17 16 ... 0 *  

* ----- *  

* | Op | ra | rb | c2 | *  

* ----- *  

*****  

OPERATION st IN pipe.ONE {
    DECLARE
    {
        GROUP ra,rb = { reg };
        GROUP c2 = { imm17 };

```

```

        INSTANCE alu, mem_access_store;
    }
    // ld opcode := 3
    CODING { 0b00011 ra rb c2 }
    SYNTAX { "st" ~" " ra "," c2 "(" rb ")" }
    BEHAVIOR {
        if(rb) { src2 = GPR[rb]; }
        else { src2 = 0; }

        src1 = SIGN_EXTEND_15(c2);
        alu_mode = ALU_ADD;
    }
    ACTIVATION { alu, mem_access_store }
    DOCUMENTATION("ST") { Store into displacement address }
}

ALIAS OPERATION st_simple IN pipe.ONE {
    DECLARE {
        GROUP ra = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu, mem_access_store;
    }
    CODING { 0b00011 ra 0b0[5] c2 }
    SYNTAX { "st" ~" " ra "," c2 }
    BEHAVIOR {
        src2 = 0;
        src1 = SIGN_EXTEND_15(c2);
        alu_mode = ALU_ADD;
    }
    ACTIVATION { alu, mem_access_store }
    DOCUMENTATION("ST") { Store into absolute address; rb is register 0 }
}

/********************* *
 *                      MEM ACCESS Definition
 * ********************/
OPERATION mem_access_load IN pipe.ONE {
    DECLARE { REFERENCE ra; }
    BEHAVIOR {
        uint32 addr;
        uint32 data;

        addr = result;
        DMEM_LD(data,addr);
        MD = data;
    }
}

OPERATION mem_access_store IN pipe.ONE {
    DECLARE { REFERENCE ra; }
    BEHAVIOR {
        uint32 addr;
        uint32 data;

```

```

        addr = result;
        data = GPR[ ra ];
        DMEM_ST(data,addr);
    }
}

```

miscellaneous.lisa

```

/*********************************************
*
*          Miscellaneous Instructions
*
*****************************************/
#include "defines.h"

/*********************************************
*
*          NOP/STOP
*      31..27  26          ...
*  -----|   Op   |           Unused   |  *
*  -----|-----|-----|-----|-----|-----|
*          *
*****************************************/
OPERATION nop IN pipe.ONE {
    // binary image: 0b0[32] means 32 zeros
    CODING { 0b0[32] }
    SYNTAX { "nop" }
    BEHAVIOR
    {
        // nothing to do here
#pragma analyze(off)
        fprintf(stdout,"Nothing to do here\n");
#pragma analyze(on)
    }
    DOCUMENTATION("NOP") { No operation. Used to insert pipeline bubble }
}

OPERATION stop IN pipe.ONE {
    // opcode = 31
    CODING { 0b11111 0b0[27] }
    SYNTAX { "stop" }
    BEHAVIOR
    {
#pragma analyze(off)
        fprintf(stdout,"Im in the stop function now what\n");
#pragma analyze(on)
        // set Run to zero, halting the machine
        RUN = false;
    }
    DOCUMENTATION("STOP") { Set RUN to zero, halting the machine }
}

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

```

```

*           Interrupt Instructions      *
*
***** ****
OPERATION een IN pipe.ONE {
    // opcode = 10
    CODING { 0b01010 0b0[27] }
    SYNTAX { "een" }
    BEHAVIOR
    {
        // Exception enable. Set overall exception enable
        IE = true;
    }
    DOCUMENTATION("EEN") { Exception enable. Set overall exception enable
        bit }
}

OPERATION edi IN pipe.ONE {
    // opcode = 11
    CODING { 0b01011 0b0[27] }
    SYNTAX { "edi" }
    BEHAVIOR
    {
        // Exception disable. Clear overall exception enable
        IE = false;
    }
    DOCUMENTATION("EDI") { Exception disable. Clear overall exception
        enable }
}

OPERATION rfi IN pipe.ONE {
    // opcode = 30
    CODING { 0b11110 0b0[27] }
    SYNTAX { "rfi" }
    BEHAVIOR
    {
        // Return from interrupt. PC <- IPC; enable exceptions
        PC = IPC;
    }
    DOCUMENTATION("RFI") { Return from interrupt. PC <- IPC; enable
        exceptions }
}

OPERATION svi IN pipe.ONE {
    DECLARE
    {
        GROUP ra,rb = { reg };
    }
    // opcode = 16
    CODING { 0b10000 ra rb 0b0[17] }
    SYNTAX { "svi" ~" " ra "," rb }
    BEHAVIOR
    {
        // Save II and IPC in ra and rb respectively
        // GPR[ra] <- II && GPR[rb] <- IPC
        GPR[ra] = II;
        GPR[rb] = IPC;
    }
}

```

```

}

DOCUMENTATION("SVI") { Save II and IPC in ra and rb, respectively }

}

OPERATION ri IN pipe.ONE {
    DECLARE
    {
        GROUP ra,rb = { reg };
    }
    // opcode = 17
    CODING { 0b10001 ra rb 0b0[17] }
    SYNTAX { "ri" ~" " ra "," rb }
    BEHAVIOR
    {
        // Restore II and IPC from ra and rb, respectively
        // II <- GPR[rb] && IPC <- GPR[rb]
        II = GPR[ra];
        IPC = GPR[rb];
    }
    DOCUMENTATION("RI") { Restore II and IPC from ra and rb,
        respectively }
}

```

immediate.lisa

```

/*********************************************
*
*           Immediate Implementation
*
*****************************************/
#include "defines.h"

IMMEDIATE imm5 {
    // 5-bit immediate value
    DECLARE { LABEL value; }
    CODING { value=0bx[5] }
    SYNTAX { SYMBOL( value=#U5 ) }
    EXPRESSION { value }
    DOCUMENTATION { 5-bit unsigned immediate value }
}

// X makes it not care about sign(#X17).
// If you want the value to hold 0xfffffe0
// you can define the symbol as -32 or 4,294,967,264.
IMMEDIATE imm17 {
    // 17-bit immediate value
    DECLARE { LABEL value; }
    CODING { value=0bx[17] }
    SYNTAX { SYMBOL( value=#X17 ) }
    EXPRESSION { value }
    DOCUMENTATION { 17-bit value/(label) sign extended to 31-bits }
}

// This operation describes a simple register

```

```
IMMEDIATE reg {
    DECLARE { LABEL idx; } // A label (=variable) for the reg addr
    SYNTAX { "r" ~idx=#U5 } // The assembly is made up by the reg addr
    CODING { idx=0bx[5] } // The reg addr is encoded within 5 bits
    EXPRESSION {idx} // This operation returns the reg addr
    DOCUMENTATION { General Purpose Register r0 - r31 }
}
```

APPENDIX B

Pipelined SRC Code

main.lisa

```
*****  
*  
*          SRC Implementation  
*  
*****  
#include "defines.h"  
#include "memory_if.h"  
#include "memory_cfg.h"  
  
VERSION("Pipelined_SRC, 2006.1.1")  
  
*****  
*  
*          Processor Resource Description  
*  
*****  
RESOURCE  
{  
    // Harvard memory map for program and data memory  
    MEMORY_MAP  
    {  
        RANGE(PMEM_START, PMEM_END) -> prog_mem[(31..0)];  
        RANGE(DMEM_START, DMEM_END) -> data_mem[(31..0)];  
    }  
  
    // 0x1000 32bit words of program memory  
    // FLAGS are set to R|X meaning that prog_mem is readable and  
    // executable  
    RAM uint32 prog_mem  
    {  
        SIZE(PMEM_SIZE);  
        BLOCKSIZE(32);  
        FLAGS(R|X);  
        ENDIANESS(BIG);  
    };  
  
    // 0x1000 32bit words of data memory  
    // FLAGS are set to R|W meaning that data mem is  
    // readable and writeable  
    RAM uint32 data_mem  
    {  
        SIZE(DMEM_SIZE);  
        BLOCKSIZE(32);  
        FLAGS(R|W);  
    };
```

```

        ENDIANESS(BIG);
    };

    // Branch Program Counter (BPC) Register
    // and Status Flag (BSET)
REGISTER TClocked<uint32> BPC;
REGISTER TClocked<bool> BSET;
bool branch_set;

    // Return from Interrupt Program Counter (RFIPC) Register
REGISTER TClocked<bool> RFIPC;

    // Register file ( 32 General Purpose Registers )
REGISTER TClocked<int32> GPR[0..31];

    // Fetch program counter register
REGISTER TClocked<uint32> FPC;

    // Program Counter ( PC ) Register
PROGRAM_COUNTER uint32 PC;

    // 32-bit Interrupt Program Counter ( IPC ) Register
REGISTER TClocked<uint32> IPC;
    // 32-bit Interrupt Information ( II ) Register
REGISTER TClocked<uint32> II;
    // 1-bit interrupt enable ( IE ) flag
REGISTER TClocked<bool> IE;
    // 1-bit RUN flag
REGISTER TClocked<bool> RUN;

    // Interrupt Request and Acknowledge PINS
PIN IN bool ireq;
PIN OUT bool iack;

/******5 Stage pipelined architecture:*****
* -----|-----|-----|-----|-----*
* | Fetch | Decode | Execute | Mem Access | Write Back | *
* | (FE)  | (DC)   | (EX)    | (MEM)      | (WB)       | *
* -----|-----|-----|-----|-----*
*****/PIPELINE pipe = { FE; DC; EX; MEM; WB };

PIPELINE_REGISTER IN pipe
{
    // Program Counter 2 ( PC2 ) Register
PROGRAM_COUNTER uint32 PC2;

    // 32-bit instruction register ( IR )
uint32 IR;

    // ALU resources
int32 X; // ALU input operand 1
int32 Y; // ALU input operand 2
int32 Z;

```

```

    unsigned char ALU_MODE; // selects ALU operation

    uint32 MD;
};

// UNIT declarations for RTL generation
UNIT U_FETCH IN pipe.FE {
    OPERATIONS (fetch);
    REGISTERS (PC);
};

UNIT ALU_DC IN pipe.DC {
    OPERATIONS (alu1, alur, aluu, shift);
};

UNIT MISC_DC IN pipe.DC {
    OPERATIONS (stop, ri, svi, rfi, edi, een);
    REGISTERS (RUN, IE, II, IPC);
};

UNIT MEM_DC IN pipe.DC {
    OPERATIONS (ld, la, st, ldr, lar, str);
};

UNIT BYPASS_DC IN pipe.DC {
    OPERATIONS (bypass_X, bypass_Y, bypass_MD);
};

UNIT U_BRANCH IN pipe.DC {
    OPERATIONS (branchl, Branch_Logic);
};

UNIT ALU_EX IN pipe.EX {
    OPERATIONS (alu);
    REGISTERS (src1, src2, dest);
};

UNIT U_WRITEBACK IN pipe.WB {
    OPERATIONS (writeback);
};
}

```

```

/***********************
*                      *
*      Processor Instruction Set Description      *
*                      *
***********************/
OPERATION reset {
    BEHAVIOR {
        // C Behavior Code to reset all the processor
        // resources to a well defined state

        // Zero out General Purpose Register
        int i;
        for(i = 0; i < 32; i++) {
            GPR[i] = 0;
        }

        // Set program counter to the program entry point
        FPC = LISA_PROGRAM_COUNTER;
        PC = 0;
        BPC = 0;
    }
}

```

```

BSET = false;
branch_set = false;
RFIPC = false;

RUN = true;
IE = true;
IPC = 0;
II = 0;

PIPELINE(pipe).flush();
#pragma analyze(off)
// declare an text-output channel with the name "debug",
// this channel can be used to print debug messages into
// a window that is created for each channel
decl_out_channel(DBGOUT,"debug");
#pragma analyze(on)
}

OPERATION main {
// Operation main is triggered every control step (clock)
DECLARE {
INSTANCE fetch, update_pc;
}
BEHAVIOR {
// Causes simulator to execute the pipeline
PIPELINE(pipe).execute();
// Causes pipeline to shift on cycle
PIPELINE(pipe).shift();
}
ACTIVATION {
// Always update current pc
update_pc
if(!pipe.DC.IN.stalled()) { fetch }
}
}

OPERATION fetch IN pipe.FE {
DECLARE {
INSTANCE decode;
}
BEHAVIOR {

// Check RUN bit to see if Machine has been halted
if(!RUN) { FPC += 0; }
else {
uint32 temp_FPC;

if(RFIPC) {
temp_FPC = IPC;
RFIPC = false;
}
else if(BSET) {
temp_FPC = BPC;
}
}
}
}

```

```

        BSET = false;
    }
    else { temp_FPC = FPC; }

    // If we have an interrupt request and if interrupts are enabled
    // Save the PC and load the Interrupt Vector
    if(ireq && IE) {
        if(!branch_set) {
            uint32 idata;
            iack = true;

            PMEM_LD(idata,temp_FPC);
            II = 0x0000ffff & idata;
            FPC = 0x000000ff & (idata >> 16);
            IPC = temp_FPC;
            OUT.IR = 0;
            OUT.PC2 = temp_FPC;
        }
        else {
            OUT.IR = 0;
            OUT.PC2 = temp_FPC;
            FPC = FPC;
        }
    }
    else { // Else run normally...
        iack = false;

        // Get an instruction from the program memory at address
        // temp_PC;
        PMEM_LD(OUT.IR,temp_FPC);
        OUT.PC2 = temp_FPC;

        // Now update the program counter to the next address
        if(temp_FPC >= 0xffff)
        {
#pragma analyze(off)
            printf(stderr,"Possible Model Fault --
                           PC goes beyond valid range and wraps to 0\n");
#pragma analyze(on)
            temp_FPC = 0;
        }
        else
        {
            // Increment the program counter
#ifndef LT_PROCESSOR_GENERATOR
            temp_FPC += 4; // for RTL generation
#else
            temp_FPC += 1; // for Processor Debugger
#endif
        }
        // Now we put the instruction word and the address into
        // our pipeline register between FE and DC stage
        FPC = temp_FPC;
    }
}
}

```

```

// Activate the operation "decode". This will put "decode" into the
// pipeline. The pipeline scheduler will automatically execute it
// when the time is appropriate
ACTIVATION { decode }
}

OPERATION decode IN pipe.DC {
    DECLARE {
        GROUP instruction = { nop || stop ||
                                ri || svi || rfi || edi || een ||
                                alui || alur || aluu ||
                                ld || ld_simple ||
                                st || st_simple ||
                                la || la_simple ||
                                ldr || str || lar ||
                                branch || branchl ||
                                shift || shift_reg || shift_count };
    }
}

// The current instruction word is in "IR"
// The current instruction word is at "PC"
CODING AT ( IN.PC2 ) { IN.IR == instruction }

// The complete syntax description is implemented in "instruction"
SYNTAX { instruction }
ACTIVATION { instruction }
}

// Update the pc only when MEM stage is not stalled
OPERATION update_pc IN pipe.DC
{
    BEHAVIOR {
        if (DC.IN.PC2) {
            // PC2 in pipe reg is valid. This PC update is only important for
            // the debugger display and GDB.
            PC = DC.IN.PC2;
        }
    }
}

```

alu.lisa

```

*****
*
*          ALU Implementation
*
*****
#include "defines.h"

RESOURCE {
    int32 src1;
    int32 src2;
    int32 dest;

```

```

}

/***** ALU Immediate *****
*      31..27  26..22 21..17 16      ...          0  *
* -----|-----|-----|-----|-----|-----|-----|
* | Op   | ra    | rb    |           c2           |   *
* -----|-----|-----|-----|-----|-----|-----|
*      *      *
***** */

OPERATION alui IN pipe.DC
{
    DECLARE
    {
        GROUP opcode = { addi || ori || andi };
        GROUP ra,rb = { reg };
        GROUP c2      = { imm17 };
        INSTANCE alu, writeback, bypass_X;
    }
    CODING { opcode ra rb c2 }
    SYNTAX { opcode ~" " ra "," rb "," c2 }
    BEHAVIOR
    {
        OUT.Y = SIGN_EXTEND_15(c2); // Put immediate into src2
        OUT.ALU_MODE = opcode;
    }
    ACTIVATION { alu, writeback, bypass_X }
    DOCUMENTATION("IMMEDIATE") { ALU IMMEDIATE }
}

OPERATION addi
{
    // addi opcode := 13
    CODING { 0b01101 }
    SYNTAX { "addi" }
    EXPRESSION { ALU_ADD }
    DOCUMENTATION("ADD") { Add rb to immediate constant, and put result
                           into ra }
}

OPERATION ori
{
    // ori opcode := 23
    CODING { 0b10111 }
    SYNTAX { "ori" }
    EXPRESSION { ALU_OR }
    DOCUMENTATION("OR") { "OR" rb and immediate constant, and put result
                           into ra }
}

OPERATION andi
{
    // andi opcode := 21
    CODING { 0b10101 }
    SYNTAX { "andi" }
    EXPRESSION { ALU_AND }
}

```

```

DOCUMENTATION("AND") { "AND" rb and immediate constnat, and put
                      result into ra }
}

/***** ALU Register *****/
*   31..27  26..22 21..17 16..12 11      ...      0  *
* -----|-----|-----|-----|-----|-----|-----|
* | Op  |  ra  |  rb  |  rc  |      unused      |  *
* -----|-----|-----|-----|-----|-----|-----|
*                                     *
***** /



OPERATION alur IN pipe.DC
{
    DECLARE
    {
        GROUP opcode = { add || sub || and || or };
        GROUP ra,rb,rc = { reg };
        INSTANCE alu, writeback, bypass_X, bypass_Y;
    }
    CODING { opcode ra rb rc 0b0[12] }
    SYNTAX { opcode ~" " ra "," rb "," rc }
    BEHAVIOR
    {
        OUT.ALU_MODE = opcode;
    }
    ACTIVATION { alu, writeback, bypass_X, bypass_Y }
    DOCUMENTATION("REGISTER") { ALU REGISTER }
}

OPERATION add
{
    // add opcode := 12
    CODING { 0b01100 }
    SYNTAX { "add" }
    EXPRESSION { ALU_ADD }
    DOCUMENTATION("ADD") { Add rb to rc, and put result into ra }
}

OPERATION sub
{
    // sub opcode := 14
    CODING { 0b01110 }
    SYNTAX { "sub" }
    EXPRESSION { ALU_SUB }
    DOCUMENTATION("SUB") { Subtract rc from rb, and put result into ra }
}

OPERATION and
{
    // and opcode := 20
    CODING { 0b10100 }
    SYNTAX { "and" }
    EXPRESSION { ALU_AND }
    DOCUMENTATION("AND") { "AND" rb and rc, and put result into ra }
}

```

```

}

OPERATION or
{
    // or opcode := 22
    CODING { 0b10110 }
    SYNTAX { "or" }
    EXPRESSION { ALU_OR }
    DOCUMENTATION("OR") { "OR" rb and rc, and put result into ra }
}

/********************* Neg/Not Register *****/
*           Neg/Not Register          *
*   31..27  26..22 21 ... 17 16..12 11   ...      0  *
*   -----|-----|-----|-----|-----|-----|-----*
*   | Op  |  ra  | unused | rc  |      unused   |   *
*   -----|-----|-----|-----|-----|-----|-----*
*           *
*****
```

OPERATION aluu IN pipe.DC

```

{
    DECLARE
    {
        GROUP opcode = { neg || not };
        GROUP ra,rc = { reg };
        INSTANCE alu, writeback, bypass_Y;
    }
    CODING { opcode ra 0b0[5] rc 0b0[12] }
    SYNTAX { opcode ~" " ra "," rc }
    BEHAVIOR
    {
        OUT.ALU_MODE = opcode;
    }
    ACTIVATION { alu, writeback, bypass_Y }
    DOCUMENTATION ("UNARY") { ALU UNARY }
}
```

OPERATION neg

```

{
    // neg opcode := 15
    CODING { 0b01111 }
    SYNTAX { "neg" }
    EXPRESSION { ALU_NEG }
    DOCUMENTATION("NEG") { Place 2's compliment negative of rc into ra }
}
```

OPERATION not

```

{
    // not opcode := 24
    CODING { 0b11000 }
    SYNTAX { "not" }
    EXPRESSION { ALU_NOT }
    DOCUMENTATION("NOT") { Place logical "NOT" of rc into ra }
}
```

```
*****
```

```

*
*                               ALU Definition
*
*****
OPERATION alu POLL IN pipe.EX {
    BEHAVIOR {
        char ex_rb = EX.IN.IR >> 17 & 0x1f;
        char ex_rc = EX.IN.IR >> 12 & 0x1f;
        char ex_ra = EX.IN.IR >> 22 & 0x1f;
        char ex_op = EX.IN.IR >> 27 & 0x1f;

        char mem_ra = MEM.IN.IR >> 22 & 0x1f;
        char mem_op = MEM.IN.IR >> 27 & 0x1f;

        char wb_ra = WB.IN.IR >> 22 & 0x1f;
        char wb_op = WB.IN.IR >> 27 & 0x1f;

// =====
//           MD fowarding
// =====
if(ex_op == ST || ex_op == STR) {
    // nop, branch, stop, een, edi, rfi, ri, svi, st, and str
    // dont effect ra in path 0 (or have already been written to in
    // DC stage)
    if((ex_ra == mem_ra) && // if(ra3==ra4)
       (mem_op != NOP && mem_op != BRANCH && mem_op != STOP &&
        mem_op != EEN && mem_op != EDI && mem_op != RFI &&
        mem_op != RI && mem_op != SVI && mem_op != ST &&
        mem_op != STR))
    {
        // if(op4==LD(R)) then stall the pipeline one stage
        if(mem_op == LD || mem_op == LDR)
        {
            pipe.DC.IN.stall();
            pipe.EX.IN.stall();
            pipe.MEM.IN.stall();
        }
        else { OUT.MD = MEM.IN.Z; } // else update MD <- Z4
    }
    else if((ex_ra == wb_ra) &&
             (wb_op != NOP && wb_op != NOP && wb_op != BRANCH &&
              wb_op != STOP && wb_op != EEN && wb_op != EDI &&
              wb_op != RFI && wb_op != RI && wb_op != SVI &&
              wb_op != ST && wb_op != STR)) // else if(ra3==ra5) then
                                         // MD <- Z5
        { OUT.MD = WB.IN.Z; }
        else { OUT.MD = IN.MD; }
    }

// =====
//           MP6 multiplexer
// =====
// nop, branch, stop, een, edi, rfi, ri, svi, st, and str
// dont effect ra in path 0 (or have already been written to in DC
// stage)
if((mem_ra == ex_rb) &&

```

```

(mem_op != NOP && mem_op != BRANCH && mem_op != STOP &&
mem_op != EEN && mem_op != EDI && mem_op != RFI &&
mem_op != RI && mem_op != SVI && mem_op != ST && mem_op != STR
&& ex_op != LDR && ex_op != LAR && ex_op != STR))
{
    if((ex_rb == 0) &&
        (ex_op == LD || ex_op == LA || ex_op == ST))
    { src1 = EX.IN.X; }
    else if(mem_op == LD || mem_op == LDR)
    {
        pipe.DC.IN.stall();
        pipe.EX.IN.stall();
        pipe.MEM.IN.stall();
    }
    else { src1 = MEM.IN.Z; }
}
else if((wb_ra == ex_rb) &&
        (wb_op != NOP && wb_op != BRANCH && wb_op != STOP &&
        wb_op != EEN && wb_op != EDI && wb_op != RFI &&
        wb_op != RI && wb_op != SVI && wb_op != ST && wb_op != STR
        && ex_op != LDR && ex_op != LAR && ex_op != STR))
{
    if((ex_rb == 0) &&
        (ex_op == LD || ex_op == LA || ex_op == ST))
    { src1 = EX.IN.X; }
    else { src1 = WB.IN.Z; }
}
else { src1 = EX.IN.X; }

// =====
//           MP7 multiplexer
// =====
// nop, branch, stop, een, edi, rfi, ri, svi, st, and str
// dont effect ra in path 0 (or have already been written to in DC
// stage)
if(mem_ra == ex_rc &&
    (mem_op != NOP && mem_op != BRANCH && mem_op != STOP &&
    mem_op != EEN && mem_op != EDI && mem_op != RFI &&
    mem_op != RI && mem_op != SVI && mem_op != ST && mem_op != STR
    &&
    ex_op != LD && ex_op != LA && ex_op != ST &&
    ex_op != LDR && ex_op != LAR && ex_op != STR &&
    ex_op != ADDI && ex_op != ANDI && ex_op != ORI))
{
    if((ex_rc == 0) &&
        (ex_op == SHR || ex_op == SHRA ||
        ex_op == SHL || ex_op == SCH))
    { src2 = EX.IN.Y; }
    else if(mem_op == LD || mem_op == LDR)
    {
        pipe.DC.IN.stall();
        pipe.EX.IN.stall();
        pipe.MEM.IN.stall();
    }
    else { src2 = MEM.IN.Z; }
}
else if(wb_ra == ex_rc &&

```

```

(wb_op != NOP && wb_op != BRANCH && wb_op != STOP &&
wb_op != EEN && wb_op != EDI && wb_op != RFI &&
wb_op != RI && wb_op != SVI && wb_op != ST && wb_op != STR
&& ex_op != LD && ex_op != LA && ex_op != ST &&
ex_op != LDR && ex_op != LAR && ex_op != STR &&
ex_op != ADDI && ex_op != ANDI && ex_op != ORI))
{
    if((ex_rc == 0) &&
       (ex_op == SHR || ex_op == SHRA ||
        ex_op == SHL || ex_op == SHC))
    { src2 = EX.IN.Y; }
    else { src2 = WB.IN.Z; }
}
else { src2 = EX.IN.Y; }

// =====
//          ALU Logic
// =====
switch(EX.IN.ALU_MODE)
{
    case ALU_ADD:
        dest = src1 + src2;
        break;
    case ALU_SUB:
        dest = src1 - src2;
        break;
    case ALU_AND:
        dest = src1 & src2;
        break;
    case ALU_OR:
        dest = src1 | src2;
        break;
    case ALU_NEG:
        dest = -src2;
        break;
    case ALU_NOT:
        dest = ~src2;
        break;
    case ALU_SHR:
        dest = (uint32)src1 >> src2;
        break;
    case ALU_SHRA:
        dest = src1 >> src2;
        break;
    case ALU_SHL:
        dest = src1 << src2;
        break;
    case ALU_SHC:
        dest = ((src1 << src2) | ((uint32)src1 >> (32-src2)));
        break;
    default:
        dest = 0;
        break;
}
EX.OUT.Z = dest;
}
}

```

```

OPERATION writeback IN pipe.WB
{
    BEHAVIOR
    {
        char ra = WB.IN.IR >> 22 & 0x1f;
        GPR[ ra ] = WB.IN.Z;
    }
}

```

shift.lisa

```

/***** ****
*
*          SHIFT Implementation
*
***** */
#include "defines.h"

/***** ****
*
*          shift immediate
*  31..27 26..22 21..17 16           ...      5 4 ... 0 *
*  -----|-----|-----|-----|-----|-----|-----|-----|-----|
*  | Op   | ra    | rb    | (c3)  unused   | count  | *
*  -----|-----|-----|-----|-----|-----|-----|-----|-----|
*  |
*  |
*          shift immediate
*  31..27 26..22 21..17 16..12 11   ...      5 4 ... 0 *
*  -----|-----|-----|-----|-----|-----|-----|-----|-----|
*  | Op   | ra    | rb    | rc    |(c3) unused| 00000 | *
*  -----|-----|-----|-----|-----|-----|-----|-----|-----|
***** */

OPERATION shift IN pipe.DC {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP c3 = { imm5 };
        GROUP ra,rb,rc = { reg };
        INSTANCE alu, writeback, bypass_X, bypass_Y;
    }
    CODING { opcode ra rb rc 0bx[7] c3 }
    SYNTAX { opcode ~" " ra "," rb "," rc "," c3 }
    BEHAVIOR {
        if(c3!=0) { OUT.Y = c3; }
        OUT.ALU_MODE = opcode;
    }
    ACTIVATION { alu, writeback, bypass_X, if(c3==0) { bypass_Y } }
}

ALIAS OPERATION shift_count IN pipe.DC {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP c3 = { imm5 };
        GROUP ra,rb = { reg };

```

```

        INSTANCE alu, writeback, bypass_X;
    }
    // opcode = 26
    CODING { opcode ra rb 0b0[5] 0bx[7] c3 }
    SYNTAX { opcode ~" " ra "," rb "," c3 }
    BEHAVIOR {
        OUT.Y = c3;
        OUT.ALU_MODE = opcode;
    }
    ACTIVATION { alu, writeback, bypass_X }
    SWITCH(opcode) {
        CASE shr:
            { DOCUMENTATION("SHR") { shift rb right into ra by constant
                c3 } }
        CASE shra:
            { DOCUMENTATION("SHRA") { shift rb right with sign-extend into ra
                by constant c3 } }
        CASE shl:
            { DOCUMENTATION("SHL") { shift rb left into ra by constant c3 } }
        CASE shc:
            { DOCUMENTATION("SHC") { shift rb left circularly into ra by
                constant c3 } }
    }
}

ALIAS OPERATION shift_reg IN pipe.DC {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP ra,rb,rc = { reg };
        INSTANCE alu, writeback, bypass_X, bypass_Y;
    }
    // opcode = 26
    CODING { opcode ra rb rc 0bx[7] 0b0[5] }
    SYNTAX { opcode ~" " ra "," rb "," rc }
    BEHAVIOR {
        OUT.ALU_MODE = opcode;
    }
    ACTIVATION { alu, writeback, bypass_X, bypass_Y }
    SWITCH(opcode) {
        CASE shr:
            { DOCUMENTATION("SHR") { shift rb right into ra by count in rc;
                c3 is 0 } }
        CASE shra:
            { DOCUMENTATION("SHRA") { shift rb right with sign-extend into ra
                by count in rc; c3 is 0 } }
        CASE shl:
            { DOCUMENTATION("SHL") { shift rb left into ra by count in rc; c3
                is 0 } }
        CASE shc:
            { DOCUMENTATION("SHC") { shift rb left circularly into ra by
                count in rc; c3 is 0 } }
    }
}

OPERATION shr {
    // opcode = 26

```

```

CODING { 0b11010 }
SYNTAX { "shr" }
EXPRESSION { ALU_SHR }
}

OPERATION shra {
// opcode = 27
CODING { 0b11011 }
SYNTAX { "shra" }
EXPRESSION { ALU_SHRA }
}

OPERATION shl {
// opcode = 28
CODING { 0b11100 }
SYNTAX { "shl" }
EXPRESSION { ALU_SHL }
}

OPERATION shc {
// opcode = 29
CODING { 0b11101 }
SYNTAX { "shc" }
EXPRESSION { ALU_SHC }
}

```

load_store.lisa

```

***** *
*           Loads and Stores      *
*                                     *
***** /
#include "defines.h"
#include "memory_if.h"

***** *
*           LDR / STR / LAR      *
*   31..27  26..22  21          ...      0  *
*   -----                         -----   *
*   |   Op   |   ra    |           c1        |   *
*   -----                         -----   *
*                                     *
***** /
OPERATION ldr IN pipe.DC
{
    DECLARE
    {
        GROUP ra = { reg };
        LABEL c1;
        INSTANCE alu, mem_access_load, writeback;
    }
    // ldr opdcode := 2
    CODING { 0b00010 ra c1=0bx[22] }
}
```

```

SYNTAX   { "ldr" ~" " ra ","
           SYMBOL(((c1=#$22))+CURRENT_ADDRESS)=#X32) }
BEHAVIOR
{
    // Load Data Relative "GPR[ra] <- M[PC + c1]
    OUT.X = IN.PC2;
    OUT.Y = SIGN_EXTEND_10(c1);
    OUT.ALU_MODE = ALU_ADD;
}
ACTIVATION { alu, mem_access_load, writeback }
DOCUMENTATION("LDR") { Load from a relative address }
}

OPERATION lar IN pipe.DC
{
DECLARE
{
    GROUP ra = { reg };
    LABEL c1;
    INSTANCE alu, writeback;
}
// lar opdcode := 6
CODING   { 0b00110 ra c1=0bx[22] }
SYNTAX   { "lar" ~" " ra ","
           SYMBOL(((c1=#$22))+CURRENT_ADDRESS)=#X32) }
BEHAVIOR
{
    // Load Address Relative "GPR[ra] <- PC + c1
    OUT.X = IN.PC2;
    OUT.Y = SIGN_EXTEND_10(c1);
    OUT.ALU_MODE = ALU_ADD;
}
ACTIVATION { alu, writeback }
DOCUMENTATION ("LAR") { Load value of relative address into ra }
}

OPERATION str IN pipe.DC
{
DECLARE
{
    GROUP ra = { reg };
    LABEL c1;
    INSTANCE alu, mem_access_store, bypass_MD;
}
// str opdcode := 4
CODING   { 0b00100 ra c1=0bx[22] }
SYNTAX   { "str" ~" " ra ","
           SYMBOL(((c1=#$22))+CURRENT_ADDRESS)=#X32) }
BEHAVIOR
{
    // Store Relative "M[c1] <- GPR[ra]
    OUT.X = IN.PC2;
    OUT.Y = SIGN_EXTEND_10(c1);
    OUT.ALU_MODE = ALU_ADD;
}
ACTIVATION { alu, mem_access_store, bypass_MD }

```

```

DOCUMENTATION ("STR") { Store into relative address }
}

/***** LD *****
*          LD
* 31..27  26..22 21..17 16      ...          0  *
* -----|-----|-----|-----|-----|-----|-----|
* | Op   | ra    | rb    |           c2           |   *
* -----|-----|-----|-----|-----|-----|-----|
*          *
***** */

OPERATION ld IN pipe.DC {
    DECLARE {
        GROUP ra,rb = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu, mem_access_load, writeback, bypass_X;
    }
    // ld opdcode := 1
    CODING { 0b00001 ra rb c2 }
    SYNTAX { "ld" ~" " ra "," c2 "(" rb ")" }
    BEHAVIOR {
        if(rb==0) { OUT.X = 0; }
        OUT.Y = SIGN_EXTEND_15(c2);
        OUT.ALU_MODE = ALU_ADD;
    }
    ACTIVATION { alu, mem_access_load, writeback, if(rb!=0) { bypass_X } }
    DOCUMENTATION("LD") { Load from displacement address }
}

// This ALIAS operation will not get translated into HDL
// it is here for the assembler and the two syntaxes for
// the ld instruction
ALIAS OPERATION ld_simple IN pipe.DC {
    DECLARE {
        GROUP ra = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu, mem_access_load, writeback;
    }
    // ld opdcode := 1
    CODING { 0b00001 ra 0b0[5] c2 }
    SYNTAX { "ld" ~" " ra "," c2 }
    BEHAVIOR {
        OUT.X = 0;
        OUT.Y = SIGN_EXTEND_15(c2);
        OUT.ALU_MODE = ALU_ADD;
    }
    ACTIVATION { alu, mem_access_load, writeback }
    DOCUMENTATION("LD") { Load from absolute address; rb is register 0 }
}

/***** LA *****
*          LA
* 31..27  26..22 21..17 16      ...          0  *
* -----|-----|-----|-----|-----|-----|-----|
*          *
***** */

```

```

* | Op | ra | rb | c2 | *
* ----- * *
* ***** /*****
OPERATION la IN pipe.DC {
    DECLARE {
        GROUP ra,rb = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu, writeback, bypass_X;
    }
    // la opcode := 5
    CODING { 0b00101 ra rb c2 }
    SYNTAX { "la" ~" " ra "," c2 "(" rb ")" }
    BEHAVIOR {
        if(rb==0) { OUT.X = 0; }
        OUT.Y = SIGN_EXTEND_15(c2);
        OUT.ALU_MODE = ALU_ADD;
    }
    ACTIVATION { alu, writeback, if(rb!=0) { bypass_X } }
    DOCUMENTATION("LA") { Load value of displacement address into ra }
}

ALIAS OPERATION la_simple IN pipe.DC {
    DECLARE {
        GROUP ra = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu, writeback;
    }
    // la opcode := 5
    CODING { 0b00101 ra 0b0[5] c2 }
    SYNTAX { "la" ~" " ra "," c2 }
    BEHAVIOR {
        OUT.X = 0;
        OUT.Y = SIGN_EXTEND_15(c2);
        OUT.ALU_MODE = ALU_ADD;
    }
    ACTIVATION { alu, writeback }
    DOCUMENTATION("LA") { Load value of absolute address into ra; rb is
        register 0 }
}

/***** ST *****
* 31..27 26..22 21..17 16 ... 0 *
* ----- * *
* | Op | ra | rb | c2 | *
* ----- * *
* ***** /*****
OPERATION st IN pipe.DC {
    DECLARE {
        GROUP ra,rb = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu, mem_access_store, bypass_MD, bypass_X;
    }
}

```

```

// st opdcode := 3
CODING { 0b00011 ra rb c2 }
SYNTAX { "st" ~" " ra "," c2 "(" rb ")" }
BEHAVIOR {
    if(rb==0) { OUT.X = 0; }
    OUT.Y = SIGN_EXTEND_15(c2);
    OUT.ALU_MODE = ALU_ADD;
}
ACTIVATION { alu, mem_access_store, bypass_MD, if(rb!=0) {bypass_X} }
DOCUMENTATION("ST") { Store into displacement address }
}

ALIAS OPERATION st_simple IN pipe.DC {
DECLARE {
    GROUP ra = { reg };
    GROUP c2 = { imm17 };
    INSTANCE alu, mem_access_store, bypass_MD;
}
CODING { 0b00011 ra 0b0[5] c2 }
SYNTAX { "st" ~" " ra "," c2 }
BEHAVIOR {
    OUT.X = 0;
    OUT.Y = SIGN_EXTEND_15(c2);
    OUT.ALU_MODE = ALU_ADD;
}
ACTIVATION { alu, mem_access_store, bypass_MD }
DOCUMENTATION("ST") { Store into absolute address; rb is register 0 }
}

/********************* *
 *                      *
 *      MEM ACCESS Definition      *
 *                      *
******************/
```

OPERATION mem_access_load IN pipe.MEM {
 BEHAVIOR { DMEM_LD(MEM.OUT.Z, (uint32)MEM.IN.Z); }
}

OPERATION mem_access_store IN pipe.MEM {
 BEHAVIOR { DMEM_ST(MEM.IN.MD, (uint32)MEM.IN.Z); }
}

branch.lisa

```

/********************* *
 *                      *
 *      Branch Instructions      *
 *                      *
******************/
```

#include "defines.h"

```

/********************* *
 *                      *
 *      Branch          *
 *                      *
******************/
```

```

*   31..27  26..22 21..17 16..12 11      ...      3 2..0    *
* -----
* | Op   |           | rb   | rc   | (c3) unused | c3  |    *
* -----
*                                     *  

******/  

OPERATION branch IN pipe.DC {  

    DECLARE {  

        GROUP rb,rc = { reg };  

        GROUP c3  = { brnv || br || brzr || brnz || brpl || brmi };  

        INSTANCE Branch_Logic;  

    }  

    // opcode = 8  

    CODING { 0b01000 0b0[5] rb rc 0b0[9] c3 }  

    SYNTAX { c3 }  

    ACTIVATION { Branch_Logic }  

    DOCUMENTATION("BRANCH") { Branch to target in rb if rc satisfies  

                                condition c3 }  

}  

OPERATION brnv {  

    DECLARE { REFERENCE rb,rc; }  

    // cond = 0  

    CODING { 0b000 }  

    SYNTAX { "brnv" }  

    EXPRESSION { BRNV }  

    DOCUMENTATION { branch never }  

}  

OPERATION br {  

    DECLARE { REFERENCE rb,rc; }  

    // cond = 1  

    CODING { 0b001 }  

    SYNTAX { "br" ~" " rb }  

    EXPRESSION { BR }  

    DOCUMENTATION { branch unconditionally to rb }  

}  

OPERATION brzr {  

    DECLARE { REFERENCE rb,rc; }  

    // cond = 2  

    CODING { 0b010 }  

    SYNTAX { "brzr" ~" " rb "," rc }  

    EXPRESSION { BRZR }  

    DOCUMENTATION { branch to rb if rc is zero }  

}  

OPERATION brnz {  

    DECLARE { REFERENCE rb,rc; }  

    // cond = 3  

    CODING { 0b011 }  

    SYNTAX { "brnz" ~" " rb "," rc }  

    EXPRESSION { BRNZ }  

    DOCUMENTATION { branch to rb if rc is non-zero }  

}

```

```

OPERATION brpl {
    DECLARE { REFERENCE rb,rc; }
    // cond = 4
    CODING { 0b100 }
    SYNTAX { "brpl" ~" " rb "," rc }
    EXPRESSION { BRPL }
    DOCUMENTATION { branch to rb if rc is positive or zero (sign is
                    plus) }
}

OPERATION brmi {
    DECLARE { REFERENCE rb,rc; }
    // cond = 5
    CODING { 0b101 }
    SYNTAX { "brmi" ~" " rb "," rc }
    EXPRESSION { BRMI }
    DOCUMENTATION { branch to rb if rc is negative (sign is minus) }
}

/***** Branch Link *****/
*           Branch Link          *
*   31..27  26..22 21..17 16..12 11   ...   3 2..0   *
*   ----- -----
*   | Op   | ra   | rb   | rc   | (c3) unused | c3   | *
*   ----- -----
*   |
*   |
***** Branch Link *****/
OPERATION branchl IN pipe.DC {
    DECLARE {
        GROUP ra,rb,rc = { reg };
        GROUP c3 = { brlnv || brl || brlzs || brlnz || brlpl || brlmi };
        INSTANCE writeback;
        INSTANCE Branch_Logic;
    }
    // opcode = 9
    CODING { 0b01001 ra rb rc 0b0[9] c3 }
    SYNTAX { c3 }
    BEHAVIOR {
        OUT.Z = IN.PC2;
    }
    ACTIVATION { Branch_Logic, writeback }
    DOCUMENTATION("BRANCH LINK") { Branch to rb if rc satisfies c3, and
                                    save PC in ra  }
}

OPERATION brlnv {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 0
    CODING { 0b000 }
    SYNTAX { "brlnv" ~" " ra }
    EXPRESSION { BRNV }
    DOCUMENTATION { Do not branch, but save PC in ra }
}

```

```

OPERATION brl {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 1
    CODING { 0b001 }
    SYNTAX { "brl" ~" " ra "," rb }
    EXPRESSION { BR }
    DOCUMENTATION { Branch unconditionally to rb, and save PC in ra }
}

OPERATION brlzs {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 2
    CODING { 0b010 }
    SYNTAX { "brlzs" ~" " ra "," rb "," rc }
    EXPRESSION { BRZR }
    DOCUMENTATION { Branch to rb if rc is zero, and save PC in ra }
}

OPERATION brlnz {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 3
    CODING { 0b011 }
    SYNTAX { "brlnz" ~" " ra "," rb "," rc }
    EXPRESSION { BRNZ }
    DOCUMENTATION { Branch to rb if rc is non-zero, and save PC in ra }
}

OPERATION brlpl {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 4
    CODING { 0b100 }
    SYNTAX { "brlpl" ~" " ra "," rb "," rc }
    EXPRESSION { BRPL }
    DOCUMENTATION { Branch to rb if rc is positive, and save PC in ra }
}

OPERATION brlmi {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 5
    CODING { 0b101 }
    SYNTAX { "brlmi" ~" " ra "," rb "," rc }
    EXPRESSION { BRMI }
    DOCUMENTATION { Branch to rb if rc is negative, and save PC in ra }
}

```

```

*****
*
*           Branch Logic
*
*****
*/
```

```

OPERATION Branch_Logic POLL IN pipe.DC {
    DECLARE {
        REFERENCE c3;
        REFERENCE rb,rc;
    }
    BEHAVIOR {
        char br_cond;

        uint32 rb2;
        int32 rc2;

        char ex_ra = EX.IN.IR >> 22 & 0x1f;
        char ex_op = EX.IN.IR >> 27 & 0x1f;

        char mem_ra = MEM.IN.IR >> 22 & 0x1f;
        char mem_op = MEM.IN.IR >> 27 & 0x1f;

        char wb_ra = WB.IN.IR >> 22 & 0x1f;
        char wb_op = WB.IN.IR >> 27 & 0x1f;

        // =====
        //
        // =====
        // st, str, branch, nop, stop, een, edi, svi, ri and rfi
        // dont use ra (or are written to in DC stage)
        if((rc == ex_ra) &&
            (ex_op != ST && ex_op != STR && ex_op != BRANCH && ex_op != NOP
             && ex_op != STOP && ex_op != EEN && ex_op != EDI && ex_op != SVI
             && ex_op != RI && ex_op != RFI))
        {
            pipe.DC.IN.stall();
            pipe.EX.IN.stall();
        }
        else if((rc == mem_ra) &&
                 (mem_op != ST && mem_op != STR && mem_op != BRANCH &&
                  mem_op != NOP && mem_op != STOP && mem_op != EEN &&
                  mem_op != EDI && mem_op != SVI && mem_op != RI &&
                  mem_op != RFI))
        { rc2 = MEM.IN.Z; }
        else if((rc == wb_ra) &&
                 (wb_op != ST && wb_op != STR && wb_op != BRANCH &&
                  wb_op != NOP && wb_op != STOP && wb_op != EEN &&
                  wb_op != EDI && wb_op != SVI && wb_op != RI &&
                  wb_op != RFI))
        { rc2 = WB.IN.Z; }
        else { rc2 = GPR[rc]; }

        // =====
        //
        // =====
        // st, str, branch, nop, stop, een, edi, svi, ri and rfi
        // dont use ra (or are written to in DC stage)
        if((rb == ex_ra) &&
            (ex_op != ST && ex_op != STR && ex_op != BRANCH && ex_op != NOP
             && ex_op != STOP && ex_op != EEN && ex_op != EDI && ex_op != SVI
             && ex_op != RI && ex_op != RFI))
    }
}

```

```

{
    pipe.DC.IN.stall();
    pipe.EX.IN.stall();
}
else if((rb == mem_ra) &&
        (mem_op != ST && mem_op != STR && mem_op != BRANCH &&
         mem_op != NOP && mem_op != STOP && mem_op != EEN &&
         mem_op != EDI && mem_op != SVI && mem_op != RI &&
         mem_op != RFI))
{ rb2 = MEM.IN.Z; }
else if((rb == wb_ra) &&
        (wb_op != ST && wb_op != STR && wb_op != BRANCH &&
         wb_op != NOP && wb_op != STOP && wb_op != EEN &&
         wb_op != EDI && wb_op != SVI && wb_op != RI &&
         wb_op != RFI))
{ rb2 = WB.IN.Z; }
else { rb2 = GPR[rb]; }

// =====
//          BRANCH Logic
// =====
br_cond = c3 & 0x1f;

switch(br_cond) {
    case BRNV: // Branch never
        BSET = false;
        branch_set = false;
        break;
    case BR: // Branch always
        BPC = rb2;
        BSET = true;
        branch_set = true;
        pipe.FE.OUT.flush();
        break;
    case BRZR: // Branch when rb is zero
        if(rc2 == 0) {
            BPC = rb2;
            BSET = true;
            branch_set = true;
            pipe.FE.OUT.flush();
        }
        else { BSET = false; }
        break;
    case BRNZ: // branch when rb != 0
        if(rc2 != 0) {
            BPC = rb2;
            BSET = true;
            branch_set = true;
            pipe.FE.OUT.flush();
        }
        else { BSET = false; }
        break;
    case BRPL: // Branch when rb > 0
        if(rc2 > 0) {
            BPC = rb2;
            BSET = true;
        }
}

```

```

        branch_set = true;
        pipe.FE.OUT.flush();
    }
    else { BSET = false; }
    break;
case BRMI: // Branch when rb < 0
    if(rc2 < 0) {
        BPC = rb2;
        BSET = true;
        branch_set = true;
        pipe.FE.OUT.flush();
    }
    else { BSET = false; }
    break;
default: // DEFAULT
    BSET = false;
    branch_set = false;
    break;
}
}
}
}

```

miscellaneous.lisa

```

*****
*
*          Miscellaneous Instructions
*
*****
#include "defines.h"

*****
*
*          NOP/STOP
*      31..27  26          ...
*      -----|-----|-----|-----|
*      |   Op   |           Unused
*      -----|-----|-----|-----|
*
*****
OPERATION nop IN pipe.DC {
    // binary image: 0b0[32] means 32 zeros
    CODING   { 0b0[32] }
    SYNTAX   { "nop" }
    BEHAVIOR
    {
#pragma analyze(off)
        // nothing to do here
        fprintf(stderr,"Nothing to do here\n");
#pragma analyze(on)
    }
    DOCUMENTATION("NOP") { No operation. Used to insert pipeline bubble }
}

```

```

OPERATION stop IN pipe.DC {
    // opcode = 31
    CODING { 0b11111 0b0[27] }
    SYNTAX { "stop" }
    BEHAVIOR
    {
        // set Run to zero, halting the machine
        RUN = false;
    }
    DOCUMENTATION("STOP") { Set RUN to zero, halting the machine }
}

/********************* Interrupt Instructions *****/
*
*                         Interrupt Instructions
*
***** /


OPERATION een IN pipe.DC {
    // opcode = 10
    CODING { 0b01010 0b0[27] }
    SYNTAX { "een" }
    BEHAVIOR
    {
        // Exception enable. Set overall exception enable
        IE = true;
    }
    DOCUMENTATION("EEN") { Exception enable. Set overall exception enable
        bit }
}

OPERATION edi IN pipe.DC {
    // opcode = 11
    CODING { 0b01011 0b0[27] }
    SYNTAX { "edi" }
    BEHAVIOR
    {
        // Exception disable. Clear overall exception enable
        IE = false;
    }
    DOCUMENTATION("EDI") { Exception disable. Clear overall exception
        enable }
}

OPERATION rfi IN pipe.DC {
    // opcode = 30
    CODING { 0b11110 0b0[27] }
    SYNTAX { "rfi" }
    BEHAVIOR
    {
        // Return from interrupt. FPC <- IPC; enable exceptions
        RFIPC = true;
        pipe.FE.OUT.flush();
    }
    DOCUMENTATION("RFI") { Return from interrupt. PC <- IPC; enable
        exceptions }
}

```

```

OPERATION svi IN pipe.DC {
    DECLARE {
        GROUP ra,rb = { reg };
    }
    // opcode = 16
    CODING { 0b10000 ra rb 0b0[17] }
    SYNTAX { "svi" ~" " ra "," rb }
    BEHAVIOR
    {
        // Save II and IPC in ra and rb respectively
        // GPR[ra] <- II && GPR[rb] <- IPC
        GPR[ra] = II;
        GPR[rb] = IPC;
    }
    DOCUMENTATION("SVI") { Save II and IPC in ra and rb, respectively }
}

OPERATION ri IN pipe.DC {
    DECLARE {
        GROUP ra,rb = { reg };
    }
    // opcode = 17
    CODING { 0b10001 ra rb 0b0[17] }
    SYNTAX { "ri" ~" " ra "," rb }
    BEHAVIOR
    {
        char wb_ra = WB.IN.IR >> 22 & 0x1f;
        char wb_op = WB.IN.IR >> 27 & 0x1f;

        // Restore II and IPC from ra and rb, respectively
        // II <- GPR[rb] && IPC <- GPR[rb]
        //-----
        // nop, branch, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra (or have already been written to in DC stage)
        if((wb_ra == ra) &&
            (wb_op != NOP && wb_op != BRANCH && wb_op != STOP &&
            wb_op != EEN && wb_op != EDI && wb_op != RFI && wb_op != RI &&
            wb_op != SVI && wb_op != ST && wb_op != STR))
        { II = WB.IN.Z; }
        else { II = GPR[ra]; }
        // II = GPR[ra];

        //-----
        // nop, branch, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra (or have already been written to in DC stage)
        if((wb_ra == rb) &&
            (wb_op != NOP && wb_op != BRANCH && wb_op != STOP &&
            wb_op != EEN && wb_op != EDI && wb_op != RFI && wb_op != RI &&
            wb_op != SVI && wb_op != ST && wb_op != STR))
        { IPC = WB.IN.Z; }
        else { IPC = GPR[rb]; }
        // IPC = GPR[rb];
    }
    DOCUMENTATION("RI") { Restore II and IPC from ra and rb,
        respectively }
}

```

```
}
```

immediate.lisa

```
*****  
*  
*           Immediate Implementation  
*  
*****  
#include "defines.h"  
  
IMMEDIATE imm5  
{  
    // 5-bit immediate value  
    DECLARE { LABEL value; }  
    CODING { value=0bx[5] }  
    SYNTAX { SYMBOL( value=#U5 ) }  
    EXPRESSION { value }  
    DOCUMENTATION { 5-bit unsigned immediate value }  
}  
  
IMMEDIATE imm17  
{  
    // 17-bit immediate value  
    DECLARE { LABEL value; }  
    CODING { value=0bx[17] }  
    SYNTAX { SYMBOL( value=#S17 ) }  
    EXPRESSION { value }  
    DOCUMENTATION { 17-bit signed value/(label) }  
}  
  
// This operation describes a simple register  
IMMEDIATE reg  
{  
    DECLARE { LABEL idx; } // A label (=variable) for the reg addr  
    SYNTAX { "r" ~idx=#U } // The assembly is made up by the reg addr  
    CODING { idx=0bx[5] } // The reg addr is encoded within 5 bits  
    EXPRESSION { idx } // This operation returns the reg addr  
    DOCUMENTATION { General Purpose Register r0 - r31 }  
}
```

bypass.lisa

```
*****  
*  
*           Bypass X, Y, and MD from DC stage  
*  
*****  
#include "defines.h"  
  
OPERATION bypass_X IN pipe.DC {  
    DECLARE { REFERENCE rb; }
```

```

BEHAVIOR {
    char wb_ra = WB.IN.IR >> 22 & 0x1f;
    char wb_op = WB.IN.IR >> 27 & 0x1f;

    //-----
    // nop, branch, stop, een, edi, rfi, ri, svi, st, and str
    // dont effect ra (or have already been written to in DC stage)
    if((wb_ra == rb) &&
        (wb_op != NOP && wb_op != BRANCH && wb_op != STOP &&
         wb_op != EEN && wb_op != EDI && wb_op != RFI && wb_op != RI &&
         wb_op != SVI && wb_op != ST && wb_op != STR))
    { OUT.X = WB.IN.Z; }
    else { OUT.X = GPR[rb]; }
}
}

OPERATION bypass_Y IN pipe.DC {
    DECLARE { REFERENCE rc; }
    BEHAVIOR {
        char wb_ra = WB.IN.IR >> 22 & 0x1f;
        char wb_op = WB.IN.IR >> 27 & 0x1f;

        //-----
        // nop, branch, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra (or have already been written to in DC stage)
        if((wb_ra == rc) &&
            (wb_op != NOP && wb_op != BRANCH && wb_op != STOP &&
             wb_op != EEN && wb_op != EDI && wb_op != RFI && wb_op != RI &&
             wb_op != SVI && wb_op != ST && wb_op != STR))
        { OUT.Y = WB.IN.Z; }
        else { OUT.Y = GPR[rc]; }
    }
}

OPERATION bypass_MD IN pipe.DC {
    DECLARE { REFERENCE ra; }
    BEHAVIOR {
        char wb_ra = WB.IN.IR >> 22 & 0x1f;
        char wb_op = WB.IN.IR >> 27 & 0x1f;

        //-----
        // if WB has a newer version of MD to be stored than the GPR
        // load it into MD out else load the GPR value
        if((wb_ra == ra) &&
            (wb_op != NOP && wb_op != BRANCH && wb_op != STOP &&
             wb_op != EEN && wb_op != EDI && wb_op != RFI && wb_op != RI &&
             wb_op != SVI && wb_op != ST && wb_op != STR))
        { OUT.MD = WB.IN.Z; }
        else { OUT.MD = GPR[ra]; }
    }
}

```

APPENDIX C

VLIW SRC Code

main.lisa

```
*****SRC Implementation*****
*
// IMPORTANT !!
// Documentation Generator is not yet supported for VLIW architectures

#include "defines.h"
#include "memory_if.h"
#include "memory_cfg.h"

// Specify a version string which is contained later
// in the generated tools
VERSION("VLIW SRC, 2006.1.1")

*****Processor Resource Description*****
*
RESOURCE
{
    // Havard memory map for program and data memory
MEMORY_MAP
{
    RANGE(PMEM_START, PMEM_END) -> prog_mem[(31..0)];
    RANGE(DMEM_START, DMEM_END) -> data_mem[(31..0)];
}

    // 0x10000 64bit words of program memory
    // FLAGS are set to R|X meaning that prog_mem is readable and
    // executable
    RAM uint64 prog_mem
    {
        SIZE(PMEM_SIZE);
        BLOCKSIZE(64);
        FLAGS(R|X);
        ENDIANESS(BIG);
    };

    // 0x10000 32bit words of data memory
    // FLAGS are set to R|W meaning that data mem is
    // readable and writeable
    RAM uint32 data_mem
}
```

```

{
    SIZE(DMEM_SIZE);
    BLOCKSIZE(32);
    FLAGS(R|W);
    ENDIANESS(BIG);
};

// Register file ( 32 General Purpose Registers )
REGISTER TClocked<int32> GPR[0..31];

// Program Counter ( PC ) Register
PROGRAM_COUNTER TClocked<uint32> PC;

// 32-bit instruction register ( IR )
uint32 IRO;
uint32 IR1;

// Branch Program Counter (BPC) Register
// and Status Flag (BSET)
REGISTER TClocked<uint32> BPC;
REGISTER TClocked<bool> BSET;

// ALU and shifter resources
int32 src1<0..1>; // ALU input operand 1
int32 src2<0..1>; // ALU input operand 2
int32 result<0..1>; // ALU output result
unsigned char alu_mode<0..1>; // selects ALU operation
unsigned char shift_mode;

// Interface to Memory Registers
uint32 MD; // Data Register to Memory

// 32-bit Interrupt Program Counter ( IPC ) Register
REGISTER TClocked<uint32> IPC;
// 32-bit Interrupt Information ( II ) Register
REGISTER TClocked<uint32> II;
// 1-bit interrupt enable ( IE ) flag
REGISTER TClocked<bool> IE;
REGISTER TClocked<bool> RUN;

// Interrupt Request and Acknowledge PINS
PIN IN bool ireq;
PIN OUT bool iack;

PIPELINE pipe = { ONE };

// UNIT declarations for RTL generation
UNIT U_FETCH IN pipe.ONE {
    OPERATIONS (fetch);
    REGISTERS (IRO, IR1, PC);
};
UNIT U_EXECUTE IN pipe.ONE {

```

```

OPERATIONS (alu, alui, alur, aluu, shift, shifter);
REGISTERS (src1<0>, src1<1>, src2<0>, src2<1>, result<0>,
           result<1>, alu_mode<0>, alu_mode<1>);
};

UNIT U_BRANCH IN pipe.ONE {
    OPERATIONS (Branch_Logic);
    REGISTERS (BPC, BSET);
};

UNIT U_MEMORY IN pipe.ONE {
    OPERATIONS (ld, la, st, ldr, lar, str);
    REGISTERS (MD);
};

UNIT U_WRITEBACK IN pipe.ONE {
    OPERATIONS (writeback);
};

UNIT U_MISCCELLANEOUS IN pipe.ONE {
    OPERATIONS (stop);
    REGISTERS (RUN);
};

UNIT U_INTR IN pipe.ONE {
    OPERATIONS (een, edi, rfi, svi, ri);
    REGISTERS (IPC, II, IE);
};

}

/*****
*
*          Processor Instruction Set Description
*
*****/
OPERATION reset {
    BEHAVIOR
    {
        // C Behavior Code to reset all the processor
        // resources to a well defined state

        // Zero out General Purpose Register
        int i;
        for(i = 0; i < 32; i++)
        {
            GPR[i] = 0;
        }

        // Set program counter to the program entry point
        PC = LISA_PROGRAM_COUNTER;

        // Set Branch Program Counter and Branch Set Flag to defined states
        BPC = 0;
        BSET = false;
        RUN = true;
        IE = false;
        iack = false;

        PIPELINE(pipe).flush();
    #pragma analyze(off)
    }
}

```

```

// declare an text-output channel with the name "debug",
// this channel can be used to print debug messages into
// a window that is created for each channel
decl_out_channel(DBGOUT,"debug");
#pragma analyze(on)
}
}

OPERATION main {
    // Operation main is triggered every control step (clock)
    DECLARE {
        INSTANCE fetch;
    }
    BEHAVIOR {
        PIPELINE(pipe).execute();
        PIPELINE(pipe).shift();
    }
    ACTIVATION { fetch }
}

OPERATION fetch IN pipe.ONE {
    DECLARE
    {
        INSTANCE decode;
    }
    BEHAVIOR {
        uint32 temp_PC;
        uint64 temp;

        // Check RUN bit to see if Machine has been halted
        if(!RUN) { PC += 0; }
        else {
            if(BSET) { temp_PC = BPC; }
            else { temp_PC = PC; }

            // If we have an interrupt request and if interrupts are enabled
            // Save the PC and load the Interrupt Vector
            if(ireq && IE)
            {
                uint32 iack_data;
                iack = true;

                PMEM_LD(temp,temp_PC); // data mem read from highest word
                // prog_bus.read(temp_PC,&iack_data);
                iack_data = temp;
                II = 0x0000ffff & iack_data;
                PC = 0x000000ff & (iack_data >> 16);
                IPC = temp_PC;
                IRO = 0;
                IR1 = 0;
            }
            else
            {
                iack = false;
                // Get an instruction from the program memory at address
                // temp_PC;
            }
        }
    }
}

```

```

        PMEM_LD(temp,temp_PC);
        // PMEM_LD(IR0,temp_PC+4);
        IR1 = temp;
        IR0 = temp >> 32;

        // No, update the program counter to the next address
        if (PC >= 0xffff) {
#pragma analyze(off)
            fprintf(stderr,"Possible Model Fault --
                        PC goes beyond valid range and wraps to 0\n");
#pragma analyze(on)
            temp_PC = 0;
        }
        else
        {
            // Increment the program counter to the next address
#ifndef LT_PROCESSOR_GENERATOR
            temp_PC += 8; // For RTL Generation
#else
            temp_PC += 1; // For Processor Debugger
#endif
        }
        PC = temp_PC;
    }
}

ACTIVATION { decode }

OPERATION decode IN pipe.ONE {
    DECLARE {
        // insn1 = ld, ldr, st, str, een
        // insn2 = shr, shra, shc, shl, br, brl
        // both = alu, alui, alur, nop, stop, la, lar
        GROUP insn1 = { nop<0> || stop<0> ||
                        alur<0> || alui<0> || aluu<0> ||
                        la<0> || la_simple<0> || lar<0> ||
                        ld || ld_simple || ldr ||
                        st || st_simple || str ||
                        een || edi || rfi || svi || ri };
        GROUP insn2 = { nop<1> || stop<1> ||
                        alur<1> || alui<1> || aluu<1> ||
                        la<1> || la_simple<1> || lar<1> ||
                        shift || shift_reg || shift_count ||
                        branch || branchl };
    }

    // The current instruction word is in "IR"
    // The current instruction word is at "PC"
    CODING AT ( PC ) { (IR1 == insn2) && (IR0 == insn1) }

    // The complete syntax description is implemented in "instruction"
    SYNTAX { insn1 "|" insn2 }
    // Execute the instruction that was selected by
    // the decoder.
    BEHAVIOR {

```

```

        char op = IR1 >> 27 & 0x1f;
        if(op != BRANCH && op != BRANCHL) {
            BSET = false;
            BPC = 0;
        }
    }
    ACTIVATION { insn1, insn2 }
}

```

alu.lisa

```

/*
*          ALU Implementation
*
*****/
#include "defines.h"

/*
*          ALU Immediate
*  31..27  26..22 21..17 16      ...
*  -----|-----|-----|-----|-----|-----|
*  | Op   | ra    | rb    |           c2           |   |
*  -----|-----|-----|-----|-----|-----|-----|
*
*****/
INSTRUCTION alui<id> IN pipe.ONE {
    DECLARE {
        GROUP opcode = { addi || ori || andi };
        GROUP ra,rb = { reg };
        GROUP c2      = { imm17 };
        INSTANCE alu<id>;
        INSTANCE writeback<id>;
    }
    CODING   { opcode ra rb c2 }
    SYNTAX   { opcode ~" " ra "," rb "," c2 }
    BEHAVIOR {
        src1<id> = GPR[rb]; // Load GPR[rb] into src1
        src2<id> = SIGN_EXTEND_15(c2); // Put immediate into src2
        alu_mode<id> = opcode; // Set the ALU mode
    }
    ACTIVATION { alu, writeback } // Activate the ALU & writeback
    DOCUMENTATION("ALU IMMEDIATE") { ALU Immediate Instructions }
}

OPERATION addi {
    // addi opcode := 13
    CODING   { 0b01101 }
    SYNTAX   { "addi" }
    EXPRESSION { ALU_ADD }
    DOCUMENTATION { Add rb to immediate constant, and put result into
                    ra }
}

```

```

OPERATION ori {
    // ori opcode := 23
    CODING { 0b10111 }
    SYNTAX { "ori" }
    EXPRESSION { ALU_OR }
    DOCUMENTATION { "OR" rb and immediate constant, and put result into
        ra }
}

OPERATION andi {
    // andi opcode := 21
    CODING { 0b10101 }
    SYNTAX { "andi" }
    EXPRESSION { ALU_AND }
    DOCUMENTATION { "AND" rb and immediate constnat, and put result into
        ra }
}

/********************* ALU Register ********************
*   31..27 26..22 21..17 16..12 11      ...      0  *
* -----|-----|-----|-----|-----|-----|-----*
* | Op | ra | rb | rc | unused |      |      *
* -----|-----|-----|-----|-----|-----|-----*
*                                     *
***** ALU Register *****

INSTRUCTION alur<id> IN pipe.ONE {
    DECLARE {
        GROUP opcode = { add || sub || and || or };
        GROUP ra,rb,rc = { reg };
        INSTANCE alu<id>;
        INSTANCE writeback<id>;
    }

    CODING { opcode ra rb rc 0b0[12] }
    SYNTAX { opcode ~" " ra "," rb "," rc }
    BEHAVIOR {
        // Load register into resource operand 1 and 2
        src1<id> = GPR[rb];
        src2<id> = GPR[rc];
        alu_mode<id> = opcode;
    }
    ACTIVATION { alu, writeback } // Activate the alu & writeback
    DOCUMENTATION("ALU REGISTER") { ALU Register Instructions }
}

OPERATION add {
    // add opcode := 12
    CODING { 0b01100 }
    SYNTAX { "add" }
    EXPRESSION { ALU_ADD }
    DOCUMENTATION { Add rb to rc, and put result into ra }
}

OPERATION sub {

```

```

// sub opcode := 14
CODING { 0b01110 }
SYNTAX { "sub" }
EXPRESSION { ALU_SUB }
DOCUMENTATION { Subtract rc from rb, and put result into ra }
}

OPERATION and {
// and opcode := 20
CODING { 0b10100 }
SYNTAX { "and" }
EXPRESSION { ALU_AND }
DOCUMENTATION { "AND" rb and rc, and put result into ra }
}

OPERATION or {
// or opcode := 22
CODING { 0b10110 }
SYNTAX { "or" }
EXPRESSION { ALU_OR }
DOCUMENTATION { "OR" rb and rc, and put result into ra }
}

/*****************
*           Neg/Not Register
*   31..27  26..22 21 ... 17 16..12 11      ...
*   -----|-----|-----|-----|-----|-----|-----|
*   | Op   |   ra  | unused |   rc  |      unused |      |
*   -----|-----|-----|-----|-----|-----|-----|
*   *
***** */
INSTRUCTION aluu<id> IN pipe.ONE {
DECLARE {
GROUP opcode = { neg || not };
GROUP ra,rc = { reg };
INSTANCE alu<id>;
INSTANCE writeback<id>;
}
CODING { opcode ra 0b0[5] rc 0b0[12] }
SYNTAX { opcode ~" " ra "," rc }
BEHAVIOR {
src1<id> = 0;
src2<id> = GPR[rc];
alu_mode<id> = opcode;
}
ACTIVATION { alu, writeback } // Activate the alu & writeback
DOCUMENTATION ("ALU UNARY") { ALU Unary Instructions }
}

OPERATION neg {
// neg opcode := 15
CODING { 0b01111 }
SYNTAX { "neg" }
EXPRESSION { ALU_NEG }
DOCUMENTATION { Place 2's compliment negative of rc into ra }

```

```

}

OPERATION not {
    // not opcode := 24
    CODING { 0b11000 }
    SYNTAX { "not" }
    EXPRESSION { ALU_NOT }
    DOCUMENTATION { Place logical "NOT" of rc into ra }
}

/********************* ALU Definition ********************/
*                                                       *
*          ALU Definition                           *
*                                                       *
*****/



OPERATION alu<id> IN pipe.ONE {
    BEHAVIOR {
        switch(alu_mode<id>) {
            case ALU_ADD: // Add
                result<id> = src1<id> + src2<id>;
                break;
            case ALU_SUB: // Sub
                result<id> = src1<id> - src2<id>;
                break;
            case ALU_AND: // And
                result<id> = src1<id> & src2<id>;
                break;
            case ALU_OR: // Or
                result<id> = src1<id> | src2<id>;
                break;
            case ALU_NEG: // Neg
                result<id> = -src2<id>;
                break;
            case ALU_NOT: // Not
                result<id> = ~src2<id>;
                break;
            default:
                result<id> = 0;
                break;
        }
    }
}

OPERATION writeback<id> IN pipe.ONE {
    DECLARE { REFERENCE ra; }
    BEHAVIOR {
        GPR[ra] = result<id>;
    }
}

OPERATION writeback_ld IN pipe.ONE {
    DECLARE { REFERENCE ra; }
    BEHAVIOR {
        GPR[ra] = MD;
    }
}

```

```

OPERATION writeback_brl IN pipe.ONE {
    DECLARE { REFERENCE ra; }
    BEHAVIOR {
        GPR[ra] = PC;
    }
}

```

shift.lisa

```

/*********************************************
*
*          SHIFT Implementation
*
*****************************************/
#include "defines.h"

/*********************************************
*
*          shift immediate
*
*  31..27 26..22 21..17 16      ...      5 4 ... 0
*  -----|-----|-----|-----|-----|-----|-----|
*  | Op   |   ra  |   rb  | (c3)  unused   | count  |
*  -----|-----|-----|-----|-----|-----|-----|
*
*          shift register
*
*  31..27 26..22 21..17 16..12 11   ...   5 4 ... 0
*  -----|-----|-----|-----|-----|-----|-----|
*  | Op   |   ra  |   rb  | rc  |(c3) unused| 00000 | 00000
*  -----|-----|-----|-----|-----|-----|-----|
*****************************************/
INSTRUCTION shift IN pipe.ONE {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP ra,rb,rc = { reg };
        GROUP c3 = { imm5 };
        INSTANCE shifter, writeback<1>;
    }
    CODING { opcode ra rb rc 0bx[7] c3 }
    SYNTAX { opcode ~" " ra "," rb "," rc "," c3 }
    BEHAVIOR {
        if(c3) { src2<1> = c3; }
        else { src2<1> = GPR[rc]; }
        src1<1> = GPR[rb];
        shift_mode = opcode;
    }
    ACTIVATION { shifter, writeback }
}

ALIAS INSTRUCTION shift_count IN pipe.ONE {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP ra,rb = { reg };
        GROUP c3 = { imm5 };
    }
}

```

```

        INSTANCE shifter, writeback<1>;
    }
CODING { opcode ra rb 0b0[5] 0bx[7] c3 }
SYNTAX { opcode ~" " ra "," rb "," c3 }
BEHAVIOR {
    src2<1> = c3;
    src1<1> = GPR[rb];
    shift_mode = opcode;
}
ACTIVATION { shifter, writeback }
SWITCH(opcode) {
    CASE shr:
        { DOCUMENTATION("SHR") { shift rb right into ra by constant
            c3 } }
    CASE shra:
        { DOCUMENTATION("SHRA") { shift rb right with sign-extend into ra
            by constant c3 } }
    CASE shl:
        { DOCUMENTATION("SHL") { shift rb left into ra by constant c3 } }
    CASE shc:
        { DOCUMENTATION("SHC") { shift rb left circularly into ra by
            constant c3 } }
}
}

ALIAS INSTRUCTION shift_reg IN pipe.ONE {
DECLARE {
    GROUP opcode = { shr || shra || shl || shc };
    GROUP ra,rb,rc = { reg };
    INSTANCE shifter, writeback<1>;
}
CODING { opcode ra rb rc 0bx[7] 0b0[5] }
SYNTAX { opcode ~" " ra "," rb "," rc }
BEHAVIOR {
    src2<1> = GPR[rc];
    src1<1> = GPR[rb];
    shift_mode = opcode;
}
ACTIVATION { shifter, writeback }
SWITCH(opcode) {
    CASE shr:
        { DOCUMENTATION("SHR") { shift rb right into ra by count in rc;
            c3 is 0 } }
    CASE shra:
        { DOCUMENTATION("SHRA") { shift rb right with sign-extend into ra
            by count in rc; c3 is 0 } }
    CASE shl:
        { DOCUMENTATION("SHL") { shift rb left into ra by count in rc; c3
            is 0 } }
    CASE shc:
        { DOCUMENTATION("SHC") { shift rb left circularly into ra by
            count in rc; c3 is 0 } }
}
}

OPERATION shr {

```

```

// shr opcode := 26
CODING { 0b11010 }
SYNTAX { "shr" }
EXPRESSION { ALU_SHR }
DOCUMENTATION("SHR") { Shift Right with all variables given }
}

OPERATION shra {
// shra opcode := 27
CODING { 0b11011 }
SYNTAX { "shra" }
EXPRESSION { ALU_SHRA }
DOCUMENTATION("SHRA") { Shift Right Arithmetic with all variables
given }
}

OPERATION shl {
// shl opcode := 28
CODING { 0b11100 }
SYNTAX { "shl" }
EXPRESSION { ALU_SHL }
DOCUMENTATION("SHL") { Shift Left with all variables given }
}

OPERATION shc {
// shc opcode := 29
CODING { 0b11101 }
SYNTAX { "shc" }
EXPRESSION { ALU_SHC }
DOCUMENTATION("SHC") { Shift Circular with all variables given }
}

OPERATION shifter IN pipe.ONE {
BEHAVIOR {
switch(shift_mode) {
    case ALU_SHR: // Shift Right
        result<1> = (uint32)src1<1> >> src2<1>;
        break;
    case ALU_SHRA: // Shift Right Arithmetic
        result<1> = src1<1> >> src2<1>;
        break;
    case ALU_SHL: // Shift Left
        result<1> = src1<1> << src2<1>;
        break;
    case ALU_SHC: // Shift Circular
        result<1> = (src1<1> << src2<1>) | ((uint32)src1<1> >> (32-
src2<1>));
        break;
    default:
        result<1> = 0;
        break;
}
}
}
}

```

load_store.lisa

```

*****  

* * Loads and Stores * *  

* *****  

#include "defines.h"  

#include "memory_if.h"  

*****  

* * LDR / STR / LAR * *  

* 31..27 26..22 21 ... 0 *  

* -----  

* | Op | ra | c1 | *  

* -----  

* *****  

INSTRUCTION ldr IN pipe.ONE {  

    DECLARE {  

        GROUP ra = { reg };  

        LABEL c1;  

        INSTANCE alu<0>, mem_access_load, writeback_ld;  

    }  

    // ldr opdcode := 2  

    CODING { 0b00010 ra c1=0bx[22] }  

    SYNTAX { "ldr" ~" " ra "," SYMBOL(((c1=#$S22))+CURRENT_ADDRESS)  

            =#X32) }  

    BEHAVIOR {  

        src1<0> = PC;  

        src2<0> = SIGN_EXTEND_10(c1);  

        alu_mode<0> = ALU_ADD;  

    }  

    ACTIVATION { alu, mem_access_load, writeback_ld }  

    DOCUMENTATION("LDR") { Load from a relative address }  

}  

INSTRUCTION lar<id> IN pipe.ONE {  

    DECLARE {  

        GROUP ra = { reg };  

        LABEL c1;  

        INSTANCE alu<id>, writeback<id>;  

    }  

    // lar opdcode := 6  

    CODING { 0b00110 ra c1=0bx[22] }  

    SYNTAX { "lar" ~" " ra "," SYMBOL(((c1=#$S22))+CURRENT_ADDRESS)  

            =#X32) }  

    BEHAVIOR {  

        src1<id> = PC;  

        src2<id> = SIGN_EXTEND_10(c1);  

        alu_mode<id> = ALU_ADD;  

    }  

    ACTIVATION { alu, writeback }  

    DOCUMENTATION ("LAR") { Load value of relative address into ra }  

}

```

```

INSTRUCTION str IN pipe.ONE {
    DECLARE {
        GROUP ra = { reg };
        LABEL c1;
        INSTANCE alu<0>, mem_access_store;
    }
    // str opdcode := 4
    CODING { 0b00100 ra c1=0bx[22] }
    SYNTAX { "str" ~" " ra "," SYMBOL(((c1=#S22))+CURRENT_ADDRESS)
             =#X32) }
    BEHAVIOR {
        src1<0> = PC;
        src2<0> = SIGN_EXTEND_10(c1);
        // MD = GPR[ra];
        alu_mode<0> = ALU_ADD;
    }
    ACTIVATION { alu, mem_access_store }
    DOCUMENTATION ("STR") { Store into relative address }
}

/*********************************************
*                                     LD
*   31..27  26..22 21..17 16          ...
*   -----|-----|-----|-----|-----|-----|-----|
*   | Op  |   ra |   rb |           c2           |   *
*   -----|-----|-----|-----|-----|-----|-----|
*                                     *
*****************************************/
INSTRUCTION ld IN pipe.ONE {
    DECLARE {
        GROUP ra,rb = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu<0>, mem_access_load, writeback_ld;
    }
    //ld opcode := 1
    CODING { 0b00001 ra rb c2 }
    SYNTAX { "ld" ~" " ra "," c2 "(" rb ")" }
    BEHAVIOR {
        if(rb) { src2<0> = GPR[rb]; }
        else { src2<0> = 0; }
        src1<0> = SIGN_EXTEND_15(c2);
        alu_mode<0> = ALU_ADD;
    }
    ACTIVATION { alu, mem_access_load, writeback_ld }
    DOCUMENTATION("LD") { Load from displacement address }
}

ALIAS INSTRUCTION ld_simple IN pipe.ONE {
    DECLARE {
        GROUP ra = { reg };
        GROUP c2 = { imm17 };
        INSTANCE alu<0>, mem_access_load, writeback_ld;
    }
    //ld opcode := 1
    CODING { 0b00001 ra 0b0[5] c2 }

```

```

SYNTAX { "ld" ~" " ra "," c2 }
BEHAVIOR {
    src2<0> = 0;
    src1<0> = SIGN_EXTEND_15(c2);
    alu_mode<0> = ALU_ADD;
}
ACTIVATION { alu, mem_access_load, writeback_ld }
DOCUMENTATION("LD") { Load from absolute address; rb is register 0 }
}

/********************* LA ********************/
*          LA
* 31..27 26..22 21..17 16      ...
* -----|-----|-----|-----|-----|-----|-----|
* | Op | ra | rb |           c2 |           |
* -----|-----|-----|-----|-----|-----|-----|
*
***** INSTRUCTION la<id> IN pipe.ONE {
DECLARE {
    GROUP ra,rb = { reg };
    GROUP c2 = { imm17 };
    INSTANCE alu<id>, writeback<id>;
}
// la opcode := 5
CODING { 0b00101 ra rb c2 }
SYNTAX { "la" ~" " ra "," c2 "(" rb ")" }
BEHAVIOR {
    if(rb) { src2<id> = GPR[ rb ]; }
    else { src2<id> = 0; }
    src1<id> = SIGN_EXTEND_15(c2);
    alu_mode<id> = ALU_ADD;
}
ACTIVATION { alu, writeback }
DOCUMENTATION("LA") { Load value of displacement address into ra }
}

ALIAS INSTRUCTION la_simple<id> IN pipe.ONE {
DECLARE {
    GROUP ra = { reg };
    GROUP c2 = { imm17 };
    INSTANCE alu<id>, writeback<id>;
}
// la opcode := 5
CODING { 0b00101 ra 0b0[5] c2 }
SYNTAX { "la" ~" " ra "," c2 }
BEHAVIOR {
    src2<id> = 0;
    src1<id> = SIGN_EXTEND_15(c2);
    alu_mode<id> = ALU_ADD;
}
ACTIVATION { alu, writeback }
DOCUMENTATION("LA") { Load value of absolute address into ra; rb is
register 0 }
}

```

```

*****
*                      ST
*  31..27  26..22 21..17 16      ...          0  *
* -----
* | Op   | ra    | rb    |           c2           | *
* -----
* *
*****
```

INSTRUCTION st IN pipe.ONE {
 DECLARE {
 GROUP ra,rb = { reg };
 GROUP c2 = { imm17 };
 INSTANCE alu<0>, mem_access_store;
 }
 // st opcode := 3
 CODING { 0b00011 ra rb c2 }
 SYNTAX { "st" ~ " ra ,," c2 "(" rb ")" }
 BEHAVIOR {
 if(rb) { src2<0> = GPR[rb]; }
 else { src2<0> = 0; }
 src1<0> = SIGN_EXTEND_15(c2);
 alu_mode<0> = ALU_ADD;
 }
 ACTIVATION { alu, mem_access_store }
 DOCUMENTATION("ST") { Store into displacement address }
}

ALIAS INSTRUCTION st_simple IN pipe.ONE {
 DECLARE {
 GROUP ra = { reg };
 GROUP c2 = { imm17 };
 INSTANCE alu<0>, mem_access_store;
 }
 // st opcode := 3
 CODING { 0b00011 ra 0b0[5] c2 }
 SYNTAX { "st" ~ " ra ,," c2 }
 BEHAVIOR {
 src2<0> = 0;
 src1<0> = SIGN_EXTEND_15(c2);
 alu_mode<0> = ALU_ADD;
 }
 ACTIVATION { alu, mem_access_store }
 DOCUMENTATION("ST") { Store into absolute address; rb is register 0 }
}

```

*****
*                      MEM ACCESS Definition
* *****
```

OPERATION mem_access_load IN pipe.ONE {
 BEHAVIOR {
 uint32 addr;
 uint32 data;

 addr = result<0>;

```

        DMEM_LD(data,addr);
        // data_bus.read(result,&data);
        MD = data;
    }
}

OPERATION mem_access_store IN pipe.ONE {
    DECLARE { REFERENCE ra; }
    BEHAVIOR {
        uint32 addr;
        uint32 data;

        addr = result<0>;
        data = GPR[ra];
        DMEM_ST(data,addr);
        // data_bus.write(result,&data);
    }
}

```

branch.lisa

```

/********************* *
 *                      *
 *      Branch Instructions      *
 *                      *
 *******************/ 
#include "defines.h"

/********************* *
 *                      *
 *      Branch          *
 *  31..27  26..22 21..17 16..12 11      ...      3 2..0  *
 *  -----  -----
 *  |   Op   |           |   rb   |   rc   | (c3) unused  |Cond|  *
 *  -----  -----
 *                      *
 *******************/ 
INSTRUCTION branch IN pipe.ONE {
    DECLARE {
        GROUP rb,rc = { reg };
        GROUP c3 = { brnv || br || brzr || brnz || brpl || brmi };
        INSTANCE Branch_Logic;
    }
    // opcode = 8
    CODING { 0b01000 0b0[5] rb rc 0b0[9] c3 }
    SYNTAX { c3 }
    ACTIVATION { Branch_Logic }
    DOCUMENTATION("BRANCH") { Branch to target in rb if rc satisfies
        condition c3  }
}

OPERATION brnv {
    DECLARE {
        REFERENCE rb,rc;
    }
}
```

```

// cond = 0
CODING { 0b000 }
SYNTAX { "brnv" }
EXPRESSION { BRNV }
DOCUMENTATION { branch never }
}

OPERATION br {
DECLARE {
    REFERENCE rb,rc;
}
// cond = 1
CODING { 0b001 }
SYNTAX { "br" ~" " rb }
EXPRESSION { BR }
DOCUMENTATION { branch unconditionally to rb }
}

OPERATION brzr {
DECLARE {
    REFERENCE rb,rc;
}
// cond = 2
CODING { 0b010 }
SYNTAX { "brzr" ~" " rb "," rc }
EXPRESSION { BRZR }
DOCUMENTATION { branch to rb if rc is zero }
}

OPERATION brnz {
DECLARE {
    REFERENCE rb,rc;
}
// cond = 3
CODING { 0b011 }
SYNTAX { "brnz" ~" " rb "," rc }
EXPRESSION { BRNZ }
DOCUMENTATION { branch to rb if rc is non-zero }
}

OPERATION brpl {
DECLARE {
    REFERENCE rb,rc;
}
// cond = 4
CODING { 0b100 }
SYNTAX { "brpl" ~" " rb "," rc }
EXPRESSION { BRPL }
DOCUMENTATION { branch to rb if rc is positive or zero (sign is
plus) }
}

OPERATION brmi {
DECLARE {
    REFERENCE rb,rc;
}

```

```

// cond = 5
CODING { 0b101 }
SYNTAX { "brmi" ~" " rb "," rc }
EXPRESSION { BRMI }
DOCUMENTATION { branch to rb if rc is negative (sign is minus) }
}

/***** Branch Link *****/
*           Branch Link          *
*   31..27  26..22 21..17 16..12 11    ...    3 2..0  *
*   -----|-----|-----|-----|-----|-----|-----|-----*
*   | Op  | ra   | rb   | rc   | (c3) unused |Cond | *
*   -----|-----|-----|-----|-----|-----|-----|-----*
*           *
***** INSTRUCTION branchl IN pipe.ONE {
DECLARE {
    GROUP ra,rb,rc = { reg };
    GROUP c3 = { brlnv || brl || brlzs || brlnz || brlpl || brlmi };
    INSTANCE Branch_Logic, writeback_brl;
}
// opcode = 9
CODING { 0b01001 ra rb rc 0b0[9] c3 }
SYNTAX { c3 }
ACTIVATION { Branch_Logic, writeback_brl }
DOCUMENTATION("BRANCH LINK") { Branch to rb if rc satisfies c3, and
    save PC in ra }
}

OPERATION brlnv {
DECLARE {
    REFERENCE ra,rb,rc;
}
// cond = 0
CODING { 0b000 }
SYNTAX { "brlnv" ~" " ra }
EXPRESSION { BRNV }
DOCUMENTATION { Do not branch, but save PC in ra }
}

OPERATION brl {
DECLARE {
    REFERENCE ra,rb,rc;
}
// cond = 1
CODING { 0b001 }
SYNTAX { "brl" ~" " ra "," rb }
EXPRESSION { BR }
DOCUMENTATION { Branch unconditionally to rb, and save PC in ra }
}

OPERATION brlzs {
DECLARE {
    REFERENCE ra,rb,rc;
}
}

```

```

// cond = 2
CODING { 0b010 }
SYNTAX { "brlzr" ~" " ra "," rb "," rc }
EXPRESSION { BRZR }
DOCUMENTATION { Branch to rb if rc is zero, and save PC in ra }
}

OPERATION brlnz {
    DECLARE {
        REFERENCE ra,rb,rc;
    }
    // cond = 3
    CODING { 0b011 }
    SYNTAX { "brlnz" ~" " ra "," rb "," rc }
    EXPRESSION { BRLNZ }
    DOCUMENTATION { Branch to rb if rc is non-zero, and save PC in ra }
}

OPERATION brlpl {
    DECLARE {
        REFERENCE ra,rb,rc;
    }
    // cond = 4
    CODING { 0b100 }
    SYNTAX { "brlpl" ~" " ra "," rb "," rc }
    EXPRESSION { BRPL }
    DOCUMENTATION { Branch to rb if rc is positive, and save PC in ra }
}

OPERATION brlmi {
    DECLARE {
        REFERENCE ra,rb,rc;
    }
    // cond = 5
    CODING { 0b101 }
    SYNTAX { "brlmi" ~" " ra "," rb "," rc }
    EXPRESSION { BRMI }
    DOCUMENTATION { Branch to rb if rc is negative, and save PC in ra }
}

/*
*****
*          BRANCH LOGIC
*
*****
*/
OPERATION Branch_Logic IN pipe.ONE {
    DECLARE {
        REFERENCE c3;
        REFERENCE rb,rc;
    }
    BEHAVIOR {
        char br_cond;

        br_cond = c3;
    }
}

```

```

switch(br_cond) {
    case BRNV: // branch never
        BSET = false;
        break;
    case BR: // branch always
        BPC = (unsigned int)GPR[rb];
        BSET = true;
        break;
    case BRZR: // branch when condition(rc) is zero
        if((signed int)GPR[rc] == 0) {
            BPC = (unsigned int)GPR[rb];
            BSET = true;
        }
        else { BSET = false; }
        break;
    case BRNZ: // branch when condition(rc) is non-zero
        if((signed int)GPR[rc] != 0) {
            BPC = (unsigned int)GPR[rb];
            BSET = true;
        }
        else { BSET = false; }
        break;
    case BRPL: // branch when condition(rc) is positive (> 0)
        if((signed int)GPR[rc] > 0) {
            BPC = (unsigned int)GPR[rb];
            BSET = true;
        }
        else { BSET = false; }
        break;
    case BRMI: // branch when condition(rc) is negative (< 0)
        if((signed int)GPR[rc] < 0) {
            BPC = (unsigned int)GPR[rb];
            BSET = true;
        }
        else { BSET = false; }
        break;
    default:
        BSET = false;
        break;
}
}
}

```

miscellaneous.lisa

```

*****
*
*          Miscellaneous Instructions
*
*****
#include "defines.h"
*****

```

```

*                               NOP/STOP
*      31..27   26           ...          0   *
*
* | _____| |
* |   Op  |           Unused        | *
* | _____| |
* |
***** /  

INSTRUCTION nop<id> IN pipe.ONE {
    // binary image: 0b0[32] means 32 zeros
    CODING { 0b0[32] }
    SYNTAX { "nop" }
    BEHAVIOR
    {
        // nothing to do here
    #pragma analyze(off)
        fprintf(stdout,"Nothing to do here\n");
    #pragma analyze(on)
    }
    DOCUMENTATION("NOP") { No operation. Used to insert pipeline bubble }
}  

INSTRUCTION stop<id> IN pipe.ONE {
    // opcode = 31
    CODING { 0b11111 0b0[27] }
    SYNTAX { "stop" }
    BEHAVIOR
    {
    #pragma analyze(off)
        fprintf(stdout,"Im in the stop function now what\n");
    #pragma analyze(on)
        // set Run to zero, halting the machine
        RUN = false;
    }
    DOCUMENTATION("STOP") { Set RUN to zero, halting the machine }
}  

***** /  

*                               Interrupt Instructions
* |
***** /  

INSTRUCTION een IN pipe.ONE {
    // opcode = 10
    CODING { 0b01010 0b0[27] }
    SYNTAX { "een" }
    BEHAVIOR
    {
        // Exception enable. Set overall exception enable
        IE = true;
    }
    DOCUMENTATION("EEN") { Exception enable. Set overall exception enable
        bit }
}  

INSTRUCTION edi IN pipe.ONE {
```

```

// opcode = 11
CODING { 0b01011 0b0[27] }
SYNTAX { "edi" }
BEHAVIOR
{
    // Exception disable. Clear overall exception enable
    IE = false;
}
DOCUMENTATION("EDI") { Exception disable. Clear overall exception
    enable }
}

INSTRUCTION rfi IN pipe.ONE {
// opcode = 30
CODING { 0b11110 0b0[27] }
SYNTAX { "rfi" }
BEHAVIOR
{
    // Return from interrupt. PC <- IPC; enable exceptions
    PC = IPC;
}
DOCUMENTATION("RFI") { Return from interrupt. PC <- IPC; enable
    exceptions }
}

INSTRUCTION svi IN pipe.ONE {
DECLARE
{
    GROUP ra,rb = { reg };
}
// opcode = 16
CODING { 0b10000 ra rb 0b0[17] }
SYNTAX { "svi" ~" " ra "," rb }
BEHAVIOR
{
    // Save II and IPC in ra and rb respectively
    // GPR[ra] <- II && GPR[rb] <- IPC
    GPR[ra] = II;
    GPR[rb] = IPC;
}
DOCUMENTATION("SVI") { Save II and IPC in ra and rb, respectively }
}

INSTRUCTION ri IN pipe.ONE {
DECLARE
{
    GROUP ra,rb = { reg };
}
// opcode = 17
CODING { 0b10001 ra rb 0b0[17] }
SYNTAX { "ri" ~" " ra "," rb }
BEHAVIOR
{
    // Restore II and IPC from ra and rb, respectively
    // II <- GPR[rb] && IPC <- GPR[rb]
    II = GPR[ra];
}

```

```

    IPC = GPR[rb];
}
DOCUMENTATION("RI") { Restore II and IPC from ra and rb,
    respectively }
}

```

immediate.lisa

```

/*********************************************
*
*           Immediate Implementation
*
*****************************************/
#include "defines.h"

IMMEDIATE imm5
{
    // 5-bit immediate value
    DECLARE { LABEL value; }
    CODING { value=0bx[5] }
    SYNTAX { SYMBOL( value=#U5 ) }
    EXPRESSION { value }
    DOCUMENTATION { 5-bit unsigned immediate value }
}

// X makes it not care about sign(#X17).
// If you want the value to hold 0xffffffe0
// you can define the symbol as -32 or 4,294,967,264.
IMMEDIATE imm17
{
    // 17-bit immediate value
    DECLARE { LABEL value; }
    CODING { value=0bx[17] }
    SYNTAX { SYMBOL( value=#X17 ) }
    EXPRESSION { value }
    DOCUMENTATION { 17-bit value/(label) later sign extended }
}

// This operation describes a simple register
IMMEDIATE reg
{
    DECLARE { LABEL idx; } // A label (=variable) for the reg addr
    SYNTAX { "r" ~idx=#U5 } // The assembly is made up by the reg addr
    CODING { idx=0bx[5] } // The reg addr is encoded within 5 bits
    EXPRESSION { idx } // This operation returns the reg addr
    DOCUMENTATION { General Purpose Register r0 - r31 }
}

```

APPENDIX D

Pipelined VLIW SRC Code

main.lisa

```
*****SRC Implementation*****
*
*          Processor Resource Description
*
*****/
```

```
#include "defines.h"
#include "memory_if.h"
#include "memory_cfg.h"

VERSION("VLIW_PipelinedSRC, 2006.1.1")

*****Processor Resource Description*****
*
*          Processor Resource Description
*
*****/
```

```
RESOURCE
{
    // Havard memory map for program and data memory
    MEMORY_MAP
    {
        RANGE(PMEM_START, PMEM_END) -> prog_mem[(31..0)];
        RANGE(DMEM_START, DMEM_END) -> data_mem[(31..0)];
    }

    // 0x1000 32bit words of program memory
    // FLAGS are set to R|X meaning that prog_mem is readable and
    // executable
    RAM uint64 prog_mem
    {
        SIZE(PMEM_SIZE);
        BLOCKSIZE(64);
        FLAGS(R|X);
        ENDIANESS(BIG);
    };

    // 0x1000 32bit words of data memory
    // FLAGS are set to R|W meaning that data mem is
    // readable and writeable
    RAM uint32 data_mem
    {
        SIZE(DMEM_SIZE);
        BLOCKSIZE(32);
        FLAGS(R|W);
        ENDIANESS(BIG);
    };
}
```

```

};

// Branch Program Counter (BPC) Register
// and Status Flag (BSET)
REGISTER TClocked<uint32> BPC;
REGISTER TClocked<bool> BSET;
bool branch_set;

// Return from Interrupt Program Counter (RFIPC) Register
REGISTER TClocked<bool> RFIPC;

// Register file ( 32 General Purpose Registers )
REGISTER TClocked<int32> GPR[0..31];

// Fetch program counter register
REGISTER TClocked<uint32> FPC;

// Program Counter ( PC ) Register
PROGRAM_COUNTER uint32 PC;

// 32-bit Interrupt Program Counter ( IPC ) Register
REGISTER TClocked<uint32> IPC;
// 32-bit Interrupt Information ( II ) Register
REGISTER TClocked<uint32> II;
// 1-bit interrupt enable ( IE ) flag
REGISTER TClocked<bool> IE;
// 1-bit RUN flag
REGISTER TClocked<bool> RUN;

// Interrupt Request and Acknowledge PINS
PIN IN bool ireq;
PIN OUT bool iack;

/*
 *      5 Stage pipelined architecture:
 * -----
 * | Fetch | Decode | Execute | Mem Access | Write Back |
 * | (FE)  | (DC)   | (EX)    | (MEM)     | (WB)      |
 * -----
 */
PIPELINE pipe = { FE; DC; EX; MEM; WB };

PIPELINE_REGISTER IN pipe
{
    // Program Counter 2 ( PC2 ) Register
    PROGRAM_COUNTER uint32 PC2;

    // 32-bit instruction register ( IR ) (2 for VLIW implementation)
    uint32 IR0;
    uint32 IR1;

    // ALU resources
    int32 X<0..1>; // ALU input operand 1
    int32 Y<0..1>; // ALU input operand 2
    int32 Z<0..1>; // Writeback value
}

```

```

unsigned char ALU_MODE<0..1>; // selects ALU operation

char op0<0..1>;
char ra0<0..1>;
char rb0<0..1>;
char rc0<0..1>;

uint32 MD;
};

// UNIT declarations for RTL generation
UNIT U_FETCH IN pipe.FE {
    OPERATIONS (fetch);
    REGISTERS (PC);
};

UNIT ALU_DC IN pipe.DC {
    OPERATIONS (alui, alur, aluu, shift);
};

UNIT ALU_EX IN pipe.EX {
    OPERATIONS (alu);
    REGISTERS (src1<0>, src1<1>, src2<0>, src2<1>, dest<0>, dest<1>);
};

UNIT MEM_DC IN pipe.DC {
    OPERATIONS (ld, la, st, ldr, lar, str);
};

UNIT BR_DC IN pipe.DC {
    OPERATIONS (branchl, Branch_Logic);
    REGISTERS (BPC, BSET);
};

UNIT INTR_DC IN pipe.DC {
    OPERATIONS (ri, svi, rfi, edi, een, nop, stop);
    REGISTERS (IPC, IE, II, RUN);
};

}

*******/

*
*          Processor Instruction Set Description
*
*****OPERATION reset {
BEHAVIOR {
// C Behavior Code to reset all the processor
// resources to a well defined state

// Zero out General Purpose Register
int i;
for(i = 0; i < 32; i++) {
    GPR[i] = 0;
}

// Set program counter to the program entry point
FPC = LISA_PROGRAM_COUNTER;
PC = 0;
}

```

```

BPC = 0;
BSET = false;
branch_set = false;
RFIPC = false;

RUN = true;
IE = true;
IPC = 0;
II = 0;

PIPELINE(pipe).flush();
#pragma analyze(off)
// declare an text-output channel with the name "debug",
// this channel can be used to print debug messages into
// a window that is created for each channel
decl_out_channel(DBGOUT,"debug");
#pragma analyze(on)
}
}

OPERATION main {
// Operation main is triggered every control step (clock)
DECLARE {
INSTANCE fetch, update_pc;
}
BEHAVIOR {
// Causes simulator to execute the pipeline
PIPELINE(pipe).execute();
// Causes pipeline to shift on cycle
PIPELINE(pipe).shift();
}
ACTIVATION {
// Always update current pc
update_pc
if(!pipe.DC.IN.stalled()) { fetch }
}
}

OPERATION fetch IN pipe.FE {
DECLARE {
INSTANCE decode;
}
BEHAVIOR {

// Check RUN bit to see if Machine has been halted
if(!RUN) { FPC += 0; }
else {
uint32 temp_FPC;
uint64 temp;

if(RFIPC) {
temp_FPC = IPC;
RFIPC = false;
}
}
}
}

```

```

    else if(BSET) {
        temp_FPC = BPC;
        BSET = false;
    }
    else { temp_FPC = FPC; }

    // If we have an interrupt request and if interrupts are enabled
    // Save the PC and load the Interrupt Vector
    if(ireq && IE) {
        if(!branch_set) {
            uint32 idata;
            iack = true;

            PMEM_LD(temp,temp_FPC);
            idata = temp;
            II = 0x0000ffff & idata;
            FPC = 0x000000ff & (idata >> 16);
            IPC = temp_FPC;
            OUT.IR0 = 0;
            OUT.IR1 = 0;
            OUT.PC2 = temp_FPC;
        }
        else {
            OUT.IR0 = 0;
            OUT.IR1 = 0;
            OUT.PC2 = temp_FPC;
            FPC = FPC;
        }
    }
    else { // Else run normally...
        uint32 ir0;
        uint32 ir1;
        iack = false;

        // Get an instruction from the program memory at address
        // temp_PC;
        PMEM_LD(temp,temp_FPC);
        OUT.IR1 = ir1 = temp;
        OUT.IR0 = ir0 = temp >> 32;
        OUT.PC2 = temp_FPC;

        OUT.op0<0> = ir0 >> 27 & 0x1f;
        OUT.ra0<0> = ir0 >> 22 & 0x1f;
        OUT.rb0<0> = ir0 >> 17 & 0x1f;
        OUT.rc0<0> = ir0 >> 12 & 0x1f;
        OUT.op0<1> = ir1 >> 27 & 0x1f;
        OUT.ra0<1> = ir1 >> 22 & 0x1f;
        OUT.rb0<1> = ir1 >> 17 & 0x1f;
        OUT.rc0<1> = ir1 >> 12 & 0x1f;

        // Now update the program counter to the next address
        if(temp_FPC >= 0xffff)
        {
#pragma analyze(off)
            fprintf(stderr,"Possible Model Fault -- PC goes beyond

```

```

        valid range and wraps to 0\n");
#pragma analyze(on)
    temp_FPC = 0;
}
else
{
    // Increment the program counter
    temp_FPC += 8;
    // temp_FPC += 1;
}
// Now we put the instruction word and the address into
// our pipeline register between FE and DC stage
FPC = temp_FPC;
}
}
}

// Activate the operation "decode". This will put "decode" into the
// pipeline. The pipeline scheduler will automatically execute it
// when the time is appropriate
ACTIVATION { decode }
}

OPERATION decode IN pipe.DC {
DECLARE {
    // insn1 instructions := ld, ldr, st, str
    // insn2 instructions := shr, shra, shc, shl, br, brl
    // instructions in both := alu, alui, alur, nop, stop
    GROUP insn1 = { nop<0> || stop<0> ||
                    alui<0> || alur<0> || aluu<0> ||
                    la<0> || la_simple<0> || lar<0> ||
                    ld || ld_simple || ldr ||
                    st || st_simple || str ||
                    ri || svf || rfi || edi || een };
    GROUP insn2 = { nop<1> || stop<1> ||
                    alur<1> || alui<1> || aluu<1> ||
                    la<1> || la_simple<1> || lar<1> ||
                    shift || shift_reg || shift_count ||
                    branch || branchl };
}
}

// The current instruction word is in "IRO && IR1"
// The current instruction word is at "PC2"
CODING AT ( IN.PC2 ) { (IN.IR1 == insn2) && (IN.IRO == insn1) }

// The complete syntax description is implemented in "instruction"
SYNTAX { insn1 "|" insn2 }
ACTIVATION { insn1, insn2 }
}

// Update the pc only when MEM stage is not stalled
OPERATION update_pc IN pipe.DC
{
BEHAVIOR {
    if (DC.IN.PC2) {
        // PC2 in pipe reg is valid. This PC update is only important for
        // the debugger display and GDB.
    }
}
}
```

```

        PC = DC.IN.PC2;
    }
}
}
```

alu.lisa

```

/*********************************************
*
*          ALU Implementation
*
*****************************************/
#include "defines.h"

RESOURCE {
    int32 src1<0..1>;
    int32 src2<0..1>;
    int32 dest<0..1>;
}

/*********************************************
*           ALU Immediate
*   31..27  26..22 21..17 16      ...
*   -----|-----|-----|-----|-----|-----|
*   | Op   | ra    | rb    |           c2           |   |
*   -----|-----|-----|-----|-----|-----|-----|
*
*****************************************/
OPERATION alui<id> IN pipe.DC
{
    DECLARE
    {
        GROUP opcode = { addi || ori || andi };
        GROUP ra,rb = { reg };
        GROUP c2      = { imm17 };
        INSTANCE alu<id>, writeback<id>;
    }
    CODING   { opcode ra rb c2 }
    SYNTAX   { opcode ~" " ra "," rb "," c2 }
    BEHAVIOR
    {
        // nop, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra in path 0 (or have already been written to in DC
        // stage)
        if((WB.IN.ra0<0> == rb) &&
           (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
            WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
            WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
            WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
            WB.IN.op0<0> != STR))
        { OUT.X<id> = WB.IN.Z<0>; }
        // nop and stop and branch dont effect ra in path 1
        else if((WB.IN.ra0<1> == rb) &&
                (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
```

```

        WB.IN.op0<0> != BRANCH) )
{ OUT.X<id> = WB.IN.Z<1>; }
else { OUT.X<id> = GPR[rb]; }

OUT.Y<id> = SIGN_EXTEND_15(c2); // Put immediate into src2
OUT.ALU_MODE<id> = opcode;
}
ACTIVATION { alu, writeback }
DOCUMENTATION("ALU IMMEDIATE") { ALU Immediate Instructions }
}

OPERATION addi
{
// addi opcode := 13
CODING { 0b01101 }
SYNTAX { "addi" }
EXPRESSION { ALU_ADD }
DOCUMENTATION { Add rb to immediate constant, and put result into ra
}
}

OPERATION ori
{
// ori opcode := 23
CODING { 0b10111 }
SYNTAX { "ori" }
EXPRESSION { ALU_OR }
DOCUMENTATION { "OR" rb and immediate constant, and put result into
ra }
}

OPERATION andi
{
// andi opcode := 21
CODING { 0b10101 }
SYNTAX { "andi" }
EXPRESSION { ALU_AND }
DOCUMENTATION { "AND" rb and immediate constnat, and put result into
ra }
}

/*
*          ALU Register
* 31..27  26..22 21..17 16..12 11      ...
* -----|-----|-----|-----|-----|-----|-----|
* | Op   |   ra  |   rb  |   rc  |      unused    |   *
* -----|-----|-----|-----|-----|-----|-----|
* |-----|-----|-----|-----|-----|-----|-----|
* *
***** */

OPERATION alur<id> IN pipe.DC
{
DECLARE
{
GROUP opcode = { add || sub || and || or };
GROUP ra,rb,rc = { reg };

```

```

        INSTANCE alu<id>, writeback<id>;
    }
    CODING { opcode ra rb rc 0b0[12] }
    SYNTAX { opcode ~" " ra "," rb "," rc }
    BEHAVIOR
    {
        // nop, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra in path 0 (or have already been written to in DC
        // stage)
        if((WB.IN.ra0<0> == rb) &&
           (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
            WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
            WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
            WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
            WB.IN.op0<0> != STR))
        { OUT.X<id> = WB.IN.Z<0>; }
        // nop and stop and branch dont effect ra in path 1
        else if((WB.IN.ra0<1> == rb) &&
                 (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
                  WB.IN.op0<0> != BRANCH))
        { OUT.X<id> = WB.IN.Z<1>; }
        else { OUT.X<id> = GPR[rb]; }

        // nop, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra in path 0 (or have already been written to in DC
        stage)
        if((WB.IN.ra0<0> == rc) &&
           (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP && WB.IN.op0<0> != EEN &&
            WB.IN.op0<0> != EDI && WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
            WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST && WB.IN.op0<0> != STR))
        { OUT.Y<id> = WB.IN.Z<0>; }
        // nop and stop and branch dont effect ra in path 1
        else if((WB.IN.ra0<1> == rc) &&
                 (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
                  WB.IN.op0<0> != BRANCH))
        { OUT.Y<id> = WB.IN.Z<1>; }
        else { OUT.Y<id> = GPR[rc]; }

        OUT.ALU_MODE<id> = opcode;
    }
    ACTIVATION { alu, writeback }
    DOCUMENTATION("ALU REGISTER") { ALU Register Instructions }
}

OPERATION add
{
    // add opcode := 12
    CODING { 0b01100 }
    SYNTAX { "add" }
    EXPRESSION { ALU_ADD }
    DOCUMENTATION { Add rb to rc, and put result into ra }
}

```

```

OPERATION sub
{
    // sub opcode := 14
    CODING { 0b01110 }
    SYNTAX { "sub" }
    EXPRESSION { ALU_SUB }
    DOCUMENTATION { Subtract rc from rb, and put result into ra }
}

OPERATION and
{
    // and opcode := 20
    CODING { 0b10100 }
    SYNTAX { "and" }
    EXPRESSION { ALU_AND }
    DOCUMENTATION { "AND" rb and rc, and put result into ra }
}

OPERATION or
{
    // or opcode := 22
    CODING { 0b10110 }
    SYNTAX { "or" }
    EXPRESSION { ALU_OR }
    DOCUMENTATION { "OR" rb and rc, and put result into ra }
}

/*********************************************
*           Neg/Not Register          *
*   31..27  26..22 21 ... 17 16..12 11   ...      0  *
*   -----  -----
*   | Op   |   ra  |   unused |   rc  |   unused   |   *
*   -----  -----
*   |
*   *
*****************************************/
OPERATION aluu<id> IN pipe.DC
{
    DECLARE
    {
        GROUP opcode = { neg || not };
        GROUP ra,rc = { reg };
        INSTANCE alu<id>, writeback<id>;
    }
    CODING { opcode ra 0b0[5] rc 0b0[12] }
    SYNTAX { opcode ~" " ra "," rc }
    BEHAVIOR
    {
        // nop, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra in path 0 (or have already been written to in DC
        stage)
        if((WB.IN.ra0<0> == rc) &&
           (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
            WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
            WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
            WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
            WB.IN.op0<0> != STR))
    }
}

```



```

        if(MEM.IN.op0<0> == LD || MEM.IN.op0<0> == LDR)
        {
            pipe.DC.IN.stall();
            pipe.EX.IN.stall();
            pipe.MEM.IN.stall();
        }
        else { OUT.MD = MEM.IN.Z<0>; } // else update MD <- Z4
    }

    // nop, stop and branch dont effect ra in path 1
    else if((EX.IN.ra0<id> == MEM.IN.ra0<1>) &&
             (MEM.IN.op0<1> != NOP && MEM.IN.op0<1> != STOP &&
              MEM.IN.op0<1> != BRANCH)) // if(ra3==ra4)
    { OUT.MD = MEM.IN.Z<1>; } // else update MD <- Z4

    else {

        // nop, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra in path 0 (or have already been written to in
        // DC stage)
        if((EX.IN.ra0<id> == WB.IN.ra0<0>) &&
           (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
            WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
            WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
            WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
            WB.IN.op0<0> != STR)) // else if(ra3==ra5) then MD <- Z5
        { OUT.MD = WB.IN.Z<0>; }

        // nop, stop, and branches dont effect ra in path 1
        else if((EX.IN.ra0<id> == WB.IN.ra0<1>) &&
                 (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
                  WB.IN.op0<1> != BRANCH)) // else if(ra3==ra5) then MD
                                         // <- Z5
        { OUT.MD = WB.IN.Z<1>; }

        else { OUT.MD = IN.MD; }
    }
}

// =====
//          MP6 multiplexer
// =====
// Checks for read after write hazards against EX stage and MEM
// stage
// nop, stop, een, edi, rfi, ri, svi, st, and str
// dont effect ra in path 0 (or have already been written to in DC
// stage)
if(EX.IN.rb0<id> == MEM.IN.ra0<0> &&
   MEM.IN.op0<0> != NOP && MEM.IN.op0<0> != STOP &&
   MEM.IN.op0<0> != EEN && MEM.IN.op0<0> != EDI &&
   MEM.IN.op0<0> != RFI && MEM.IN.op0<0> != RI &&
   MEM.IN.op0<0> != SVI && MEM.IN.op0<0> != ST &&
   MEM.IN.op0<0> != STR && EX.IN.op0<id> != LDR &&
   EX.IN.op0<id> != LAR && EX.IN.op0<id> != STR)
{
    if((EX.IN.rb0<id> == 0) &&

```

```

        (EX.IN.op0<id> == LD || EX.IN.op0<id> == LA ||
         EX.IN.op0<id> == ST))
    { src1<id> = EX.IN.X<id>; }
    else if(MEM.IN.op0<0> == LD || MEM.IN.op0<0> == LDR)
    {
        pipe.DC.IN.stall();
        pipe.EX.IN.stall();
        pipe.MEM.IN.stall();
    }
    else { src1<id> = MEM.IN.Z<0>; }
}

// nop and stop and branch dont effect ra in path 1
else if(EX.IN.rb0<id> == MEM.IN.ra0<1> &&
        MEM.IN.op0<1> != NOP && MEM.IN.op0<1> != STOP &&
        MEM.IN.op0<1> != BRANCH && EX.IN.op0<id> != LDR &&
        EX.IN.op0<id> != LAR && EX.IN.op0<id> != STR)
{
    if((EX.IN.rb0<id> == 0) &&
       (EX.IN.op0<id> == LD || EX.IN.op0<id> == LA ||
        EX.IN.op0<id> == ST))
    { src1<id> = EX.IN.X<id>; }
    else { src1<id> = MEM.IN.Z<1>; }
}

else {
    // nop, stop, een, edi, rfi, ri, svi, st, and str
    // dont effect ra in path 0 (or have already been written to in
    // DC stage)
    if((EX.IN.rb0<id> == WB.IN.ra0<0>) &&
       (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
        WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
        WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
        WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
        WB.IN.op0<0> != STR && EX.IN.op0<id> != LDR &&
        EX.IN.op0<id> != LAR && EX.IN.op0<id> != STR))
    {
        if((EX.IN.rb0<id> == 0) &&
           (EX.IN.op0<id> == LD || EX.IN.op0<id> == LA ||
            EX.IN.op0<id> == ST))
        { src1<id> = EX.IN.X<id>; }
        else { src1<id> = WB.IN.Z<0>; }
    }
    // nop and stop and branch dont effect ra in path 1
    else if((EX.IN.rb0<id> == WB.IN.ra0<1>) &&
            (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
             WB.IN.op0<1> != BRANCH && EX.IN.op0<id> != LDR &&
             EX.IN.op0<id> != LAR && EX.IN.op0<id> != STR))
    {
        if((EX.IN.rb0<id> == 0) &&
           (EX.IN.op0<id> == LD || EX.IN.op0<id> == LA ||
            EX.IN.op0<id> == ST))
        { src1<id> = EX.IN.X<id>; }
        else { src1<id> = WB.IN.Z<1>; }
    }
}

```

```

        else { src1<id> = EX.IN.X<id>; }

    // =====
    //           MP7 multiplexer
    // =====
    // nop, stop, een, edi, rfi, ri, svi, st, and str
    // dont effect ra in path 0 (or have already been written to in DC
    // stage)
    if(EX.IN.rc0<id> == MEM.IN.ra0<0> &&
       (MEM.IN.op0<0> != NOP && MEM.IN.op0<0> != STOP &&
        MEM.IN.op0<0> != EEN && MEM.IN.op0<0> != EDI &&
        MEM.IN.op0<0> != RFI && MEM.IN.op0<0> != RI &&
        MEM.IN.op0<0> != SVI && MEM.IN.op0<0> != ST &&
        MEM.IN.op0<0> != STR && EX.IN.op0<id> != LD &&
        EX.IN.op0<id> != LA && EX.IN.op0<id> != ST &&
        EX.IN.op0<id> != LDR && EX.IN.op0<id> != LAR &&
        EX.IN.op0<id> != STR && EX.IN.op0<id> != ADDI &&
        EX.IN.op0<id> != ANDI && EX.IN.op0<id> != ORI))
    {
        if((EX.IN.rc0<id> == 0) &&
           (EX.IN.op0<id> == SHR || EX.IN.op0<id> == SHRA ||
            EX.IN.op0<id> == SHL || EX.IN.op0<id> == SHC))
        { src2<id> = EX.IN.Y<id>; }
        else if(MEM.IN.op0<0> == LD || MEM.IN.op0<0> == LDR)
        {
            pipe.DC.IN.stall();
            pipe.EX.IN.stall();
            pipe.MEM.IN.stall();
        }
        else { src2<id> = MEM.IN.Z<0>; }
    }

    // nop and stop and branch dont effect ra in path 1
    else if(EX.IN.rc0<id> == MEM.IN.ra0<1> &&
            (MEM.IN.op0<1> != NOP && MEM.IN.op0<1> != STOP &&
             MEM.IN.op0<1> != BRANCH && EX.IN.op0<id> != LD &&
             EX.IN.op0<id> != LA && EX.IN.op0<id> != ST &&
             EX.IN.op0<id> != LDR && EX.IN.op0<id> != LAR &&
             EX.IN.op0<id> != STR && EX.IN.op0<id> != ADDI &&
             EX.IN.op0<id> != ANDI && EX.IN.op0<id> != ORI))
    {
        if((EX.IN.rc0<id> == 0) &&
           (EX.IN.op0<id> == SHR || EX.IN.op0<id> == SHRA ||
            EX.IN.op0<id> == SHL || EX.IN.op0<id> == SHC))
        { src2<id> = EX.IN.Y<id>; }
        else { src2<id> = MEM.IN.Z<1>; }
    }

    else {
        // nop, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra in path 0 (or have already been written to in
        // DC stage)
        if(EX.IN.rc0<id> == WB.IN.ra0<0> &&
           (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
            WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&

```

```

WB.IN.op0<0> != RFI    && WB.IN.op0<0> != RI      &&
WB.IN.op0<0> != SVI    && WB.IN.op0<0> != ST      &&
WB.IN.op0<0> != STR    && EX.IN.op0<id> != LD      &&
EX.IN.op0<id> != LA     && EX.IN.op0<id> != ST      &&
EX.IN.op0<id> != LDR    && EX.IN.op0<id> != LAR     &&
EX.IN.op0<id> != STR    && EX.IN.op0<id> != ADDI    &&
EX.IN.op0<id> != ANDI   && EX.IN.op0<id> != ORI))
{
    if((EX.IN.rc0<id> == 0) &&
        (EX.IN.op0<id> == SHR || EX.IN.op0<id> == SHRA ||
         EX.IN.op0<id> == SHL || EX.IN.op0<id> == SHC))
    { src2<id> = EX.IN.Y<id>; }
    else { src2<id> = WB.IN.Z<0>; }
}

// nop and stop and branch dont effect ra in path 1
else if(EX.IN.rc0<id> == WB.IN.ra0<1> &&
        (WB.IN.op0<1> != NOP    && WB.IN.op0<1> != STOP    &&
         WB.IN.op0<1> != BRANCH && EX.IN.op0<id> != LD      &&
         EX.IN.op0<id> != LA     && EX.IN.op0<id> != ST      &&
         EX.IN.op0<id> != LDR    && EX.IN.op0<id> != LAR     &&
         EX.IN.op0<id> != STR    && EX.IN.op0<id> != ADDI    &&
         EX.IN.op0<id> != ANDI   && EX.IN.op0<id> != ORI))
{
    if((EX.IN.rc0<id> == 0) &&
        (EX.IN.op0<id> == SHR || EX.IN.op0<id> == SHRA ||
         EX.IN.op0<id> == SHL || EX.IN.op0<id> == SHC))
    { src2<id> = EX.IN.Y<id>; }
    else { src2<id> = WB.IN.Z<1>; }
}

else { src2<id> = EX.IN.Y<id>; }
}

// =====
//          ALU Logic
// =====
switch(EX.IN.ALU_MODE<id>)
{
    case ALU_ADD:
        dest<id> = src1<id> + src2<id>;
        break;
    case ALU_SUB:
        dest<id> = src1<id> - src2<id>;
        break;
    case ALU_AND:
        dest<id> = src1<id> & src2<id>;
        break;
    case ALU_OR:
        dest<id> = src1<id> | src2<id>;
        break;
    case ALU_NEG:
        dest<id> = -src2<id>;
        break;
    case ALU_NOT:
        dest<id> = ~src2<id>;
}

```

```

        break;
    case ALU_SHR:
        dest<id> = (uint32)src1<id> >> src2<id>;
        break;
    case ALU_SHRA:
        dest<id> = src1<id> >> src2<id>;
        break;
    case ALU_SHL:
        dest<id> = src1<id> << src2<id>;
        break;
    case ALU_SHC:
        dest<id> = ((src1<id> << src2<id>) | ((uint32)src1<id> >> (32-
            src2<id>)));
        break;
    default:
        dest<id> = 0;
        break;
    }
    EX.OUT.Z<id> = dest<id>;
}
}

OPERATION writeback<id> IN pipe.WB
{
    DECLARE { REFERENCE ra; }
    BEHAVIOR
    {
        // char ra = WB.IN.IR >> 22 & 0x1f;
        GPR[ ra ] = WB.IN.Z<id>;
    }
}

```

Shift.lisa

```

/*********************************************
*
*          SHIFT Implementation
*
*****************************************/
#include "defines.h"

/*********************************************
*
*          shift immediate
*
*  31..27  26..22 21..17 16      ...      5 4 ... 0
*  -----|-----|-----|-----|-----|-----|-----|
*  | Op   |   ra  |   rb  | (c3)  unused   | count  |
*  -----|-----|-----|-----|-----|-----|-----|
*
*          shift immediate
*
*  31..27  26..22 21..17 16..12 11   ...   5 4 ... 0
*  -----|-----|-----|-----|-----|-----|-----|
*  | Op   |   ra  |   rb  |   rc  | (c3)  unused| 00000 |
*  -----|-----|-----|-----|-----|-----|-----|

```

```

***** ****
OPERATION shift IN pipe.DC {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP ra,rb,rc = { reg };
        GROUP c3 = { imm5 };
        INSTANCE alu<1>, writeback<1>;
    }
    CODING { opcode ra rb rc 0bx[7] c3 }
    SYNTAX { opcode ~" " ra "," rb "," rc "," c3 }
    BEHAVIOR {
        if(c3) { OUT.Y<1> = c3; }
        else {
            // nop, stop, een, edi, rfi, ri, svi, st, and str
            // dont effect ra in path 0 (or have already been written to in
            // DC stage)
            if((WB.IN.ra0<0> == rc) &&
                (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
                WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
                WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
                WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
                WB.IN.op0<0> != STR))
            { OUT.Y<1> = WB.IN.Z<0>; }
            // nop and stop and branch dont effect ra in path 1
            else if((WB.IN.ra0<1> == rc) &&
                    (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
                    WB.IN.op0<0> != BRANCH))
            { OUT.Y<1> = WB.IN.Z<1>; }
            else { OUT.Y<1> = GPR[rc]; }
        }
    }

    OUT.ALU_MODE<1> = opcode;

    // nop, stop, een, edi, rfi, ri, svi, st, and str
    // dont effect ra in path 0 (or have already been written to in DC
    // stage)
    if((WB.IN.ra0<0> == rb) &&
        (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
        WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
        WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
        WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
        WB.IN.op0<0> != STR))
    { OUT.X<1> = WB.IN.Z<0>; }
    // nop and stop and branch dont effect ra in path 1
    else if((WB.IN.ra0<1> == rb) &&
            (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
            WB.IN.op0<0> != BRANCH))
    { OUT.X<1> = WB.IN.Z<1>; }
    else { OUT.X<1> = GPR[rb]; }
}

ACTIVATION { alu, writeback }
}

ALIAS OPERATION shift_count IN pipe.DC {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };

```

```

GROUP ra,rb = { reg };
GROUP c3 = { imm5 };
INSTANCE alu<1>, writeback<1>;
}
CODING { opcode ra rb 0b0[5] 0bx[7] c3 }
SYNTAX { opcode ~" " ra "," rb "," c3 }
BEHAVIOR {
    OUT.Y<1> = c3;
    OUT.ALU_MODE<1> = opcode;

    // nop, stop, een, edi, rfi, ri, svi, st, and str
    // dont effect ra in path 0 (or have already been written to in DC
    // stage)
    if((WB.IN.ra0<0> == rb) &&
        (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
         WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
         WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
         WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
         WB.IN.op0<0> != STR))
    { OUT.X<1> = WB.IN.Z<0>; }
    // nop and stop and branch dont effect ra in path 1
    else if((WB.IN.ra0<1> == rb) &&
            (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
             WB.IN.op0<0> != BRANCH))
    { OUT.X<1> = WB.IN.Z<1>; }
    else { OUT.X<1> = GPR[rb]; }
}
ACTIVATION { alu, writeback }
SWITCH(opcode) {
    CASE shr:
        { DOCUMENTATION("SHR") { shift rb right into ra by constant
                                c3 } }
    CASE shra:
        { DOCUMENTATION("SHRA") { shift rb right with sign-extend into ra
                                by constant c3 } }
    CASE shl:
        { DOCUMENTATION("SHL") { shift rb left into ra by constant c3 } }
    CASE shc:
        { DOCUMENTATION("SHC") { shift rb left circularly into ra by
                                constant c3 } }
}
}

ALIAS OPERATION shift_reg IN pipe.DC {
    DECLARE {
        GROUP opcode = { shr || shra || shl || shc };
        GROUP ra,rb,rc = { reg };
        INSTANCE alu<1>, writeback<1>;
    }
    CODING { opcode ra rb rc 0bx[7] 0b0[5] }
    SYNTAX { opcode ~" " ra "," rb "," rc }
    BEHAVIOR {
        // nop, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra in path 0 (or have already been written to in DC
        // stage)
        if((WB.IN.ra0<0> == rc) &&

```

```

(WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
WB.IN.op0<0> != STR))
{ OUT.Y<1> = WB.IN.Z<0>; }
// nop and stop and branch dont effect ra in path 1
else if((WB.IN.ra0<1> == rc) &&
(WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
WB.IN.op0<0> != BRANCH))
{ OUT.Y<1> = WB.IN.Z<1>; }
else { OUT.Y<1> = GPR[rc]; }

OUT.ALU_MODE<1> = opcode;

// nop, stop, een, edi, rfi, ri, svi, st, and str
// dont effect ra in path 0 (or have already been written to in DC
// stage)
if((WB.IN.ra0<0> == rb) &&
(WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
WB.IN.op0<0> != STR))
{ OUT.X<1> = WB.IN.Z<0>; }
// nop and stop and branch dont effect ra in path 1
else if((WB.IN.ra0<1> == rb) &&
(WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
WB.IN.op0<0> != BRANCH))
{ OUT.X<1> = WB.IN.Z<1>; }
else { OUT.X<1> = GPR[rb]; }
}

ACTIVATION { alu, writeback }
SWITCH(opcode) {
CASE shr:
{ DOCUMENTATION("SHR") { shift rb right into ra by count in rc;
c3 is 0 } }
CASE shra:
{ DOCUMENTATION("SHRA") { shift rb right with sign-extend into ra
by count in rc; c3 is 0 } }
CASE shl:
{ DOCUMENTATION("SHL") { shift rb left into ra by count in rc; c3
is 0 } }
CASE shc:
{ DOCUMENTATION("SHC") { shift rb left circularly into ra by
count in rc; c3 is 0 } }
}

OPERATION shr {
// shr opcode := 26
CODING { 0b11010 }
SYNTAX { "shr" }
EXPRESSION { ALU_SHR }
DOCUMENTATION("SHR") { SHIFT RIGHT }
}

```

```
OPERATION shra {
    // shra opcode := 27
    CODING { 0b11011 }
    SYNTAX { "shra" }
    EXPRESSION { ALU_SHRA}
    DOCUMENTATION("SHRA") { SHIFT RIGHT ARITHMETIC }
}

OPERATION shl {
    // opcode = 28
    CODING { 0b11100 }
    SYNTAX { "shl" }
    EXPRESSION { ALU_SHL }
    DOCUMENTATION("SHL") { SHIFT LEFT }
}

OPERATION shc {
    // opcode = 29
    CODING { 0b11101 }
    SYNTAX { "shc" }
    EXPRESSION { ALU_SHC }
    DOCUMENTATION("SHC") { SHIFT CIRCULAR LEFT }
}
```

load_store.lisa

```
*****
*                                     *
*             Loads and Stores          *
*                                     *
*****/
#include "defines.h"
#include "memory_if.h"

*****
*                                     LDR / STR / LAR      *
*   31..27  26..22  21           ...          0      *
*   -----|-----|-----|-----|-----|-----|-----*
*   |     Op    |     ra   |           c1           |      *
*   -----|-----|-----|-----|-----|-----|-----*
*                                     *
*****/
OPERATION ldr IN pipe.DC
{
    DECLARE
    {
        GROUP ra = { reg };
        LABEL c1;
        INSTANCE alu<0>, mem_access_load, writeback<0>;
    }
    // ldr opdcode := 2
    CODING { 0b00010 ra c1=0bx[22] }
    SYNTAX { "ldr" ~" " ra ","
}
```

```

        SYMBOL(((c1=#$22)+CURRENT_ADDRESS)=#X32) }
BEHAVIOR
{
    // Load Data Relative "GPR[ra] <- M[PC + c1]
    OUT.X<0> = IN.PC2;
    OUT.Y<0> = SIGN_EXTEND_10(c1);
    OUT.ALU_MODE<0> = ALU_ADD;
}
ACTIVATION { alu, mem_access_load, writeback }
DOCUMENTATION("LDR") { Load from a relative address }
}

OPERATION lar<id> IN pipe.DC
{
DECLARE
{
    GROUP ra = { reg };
    LABEL c1;
    INSTANCE alu<id>, writeback<id>;
}
// lar opdcode := 6
CODING   { 0b00110 ra c1=0bx[22] }
SYNTAX   { "lar" ~" " ra "," }
        SYMBOL(((c1=#$22)+CURRENT_ADDRESS)=#X32) }
BEHAVIOR
{
    // Load Address Relative "GPR[ra] <- PC + c1
    OUT.X<id> = IN.PC2;
    OUT.Y<id> = SIGN_EXTEND_10(c1);
    OUT.ALU_MODE<id> = ALU_ADD;
}
ACTIVATION { alu, writeback }
DOCUMENTATION ("LAR") { Load value of relative address into ra }
}

OPERATION str IN pipe.DC
{
DECLARE
{
    GROUP ra = { reg };
    LABEL c1;
    INSTANCE alu<0>, mem_access_store;
}
// str opdcode := 4
CODING   { 0b00100 ra c1=0bx[22] }
SYNTAX   { "str" ~" " ra "," }
        SYMBOL(((c1=#$22)+CURRENT_ADDRESS)=#X32) }
BEHAVIOR
{
    // Store Relative "M[c1] <- GPR[ra]
    OUT.X<0> = IN.PC2;
    OUT.Y<0> = SIGN_EXTEND_10(c1);

    // if WB has a newer version of MD to be stored than the GPR
    // load it into MD out else load the GPR value
    // nop, stop, een, edi, rfi, ri, svi, st, and str
}

```

```

// dont effect ra in path 0 (or have already been written to in DC
// stage)
if((WB.IN.ra0<0> == ra) &&
(WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
WB.IN.op0<0> != STR))
{ OUT.MD = WB.IN.Z<0>; }
// nop and stop dont effect ra in path 1
else if((WB.IN.ra0<1> == ra) &&
(WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
WB.IN.op0<0> != BRANCH))
{ OUT.MD = WB.IN.Z<1>; }
else { OUT.MD = GPR[ra]; }

OUT.ALU_MODE<0> = ALU_ADD;
}
ACTIVATION { alu, mem_access_store }
DOCUMENTATION ("STR") { Store into relative address }
}

/*********************************************
*                                     *
*      31..27 26..22 21..17 16          ...      0   *
* -----|-----|-----|-----|-----|-----|-----*
* | Op | ra | rb |           c2 |           |   *
* -----|-----|-----|-----|-----|-----|-----*
*                                     *
*****************************************/
OPERATION ld IN pipe.DC {
DECLARE {
    GROUP ra,rb = { reg };
    GROUP c2 = { imm17 };
    INSTANCE alu<0>, mem_access_load, writeback<0>;
}
// ld opcode := 1
CODING { 0b00001 ra rb c2 }
SYNTAX { "ld" ~" " ra "," c2 "(" rb ")" }
BEHAVIOR {
    if(rb != 0) {
        // nop, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra in path 0 (or have already been written to in
        // DC stage)
        if((WB.IN.ra0<0> == rb) &&
(WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
WB.IN.op0<0> != STR))
        { OUT.X<0> = WB.IN.Z<0>; }
        // nop and stop and branch dont effect ra in path 1
        else if((WB.IN.ra0<1> == rb) &&
(WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
WB.IN.op0<0> != BRANCH))
        { OUT.X<0> = WB.IN.Z<1>; }
    }
}

```

```

        else { OUT.X<0> = GPR[rb]; }
    }
    else { OUT.X<0> = 0; }

    // If rb is non-zero then use displacement addressing
    OUT.Y<0> = SIGN_EXTEND_15(c2);

    OUT.ALU_MODE<0> = ALU_ADD;
}
ACTIVATION { alu, mem_access_load, writeback }
DOCUMENTATION("LD") { Load from displacement address }
}

ALIAS OPERATION ld_simple IN pipe.DC {
DECLARE {
    GROUP ra = { reg };
    GROUP c2 = { imm17 };
    INSTANCE alu<0>, mem_access_load, writeback<0>;
}
// ld opcode := 1
CODING { 0b00001 ra 0b0[5] c2 }
SYNTAX { "ld" ~" " ra "," c2 }
BEHAVIOR {
    // If rb is non-zero then use displacement addressing
    OUT.X<0> = 0;
    OUT.Y<0> = SIGN_EXTEND_15(c2);
    OUT.ALU_MODE<0> = ALU_ADD;
}
ACTIVATION { alu, mem_access_load, writeback }
DOCUMENTATION("LD") { Load from absolute address; rb is register 0 }
}

/********************* LA ********************/
*      31..27 26..22 21..17 16          ...          0  *
* -----|-----|-----|-----|-----|-----|-----|-----*
* | Op  |   ra  |   rb  |           c2           |   |
* -----|-----|-----|-----|-----|-----|-----|-----*
*                                              *
***** OPERATION la<id> IN pipe.DC {
DECLARE {
    GROUP ra,rb = { reg };
    GROUP c2 = { imm17 };
    INSTANCE alu<id>, writeback<id>;
}
// la opcode := 5
CODING { 0b00101 ra rb c2 }
SYNTAX { "la" ~" " ra "," c2 "(" rb ")" }
BEHAVIOR {
    if(rb != 0) {
        // nop, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra in path 0 (or have already been written to in
        // DC stage)
        if((WB.IN.ra0<0> == rb) &&
            (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&

```

```

        WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
        WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
        WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
        WB.IN.op0<0> != STR))
    { OUT.X<0> = WB.IN.Z<0>; }
    // nop and stop and branch dont effect ra in path 1
    else if((WB.IN.ra0<1> == rb) &&
            (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
             WB.IN.op0<0> != BRANCH))
    { OUT.X<0> = WB.IN.Z<1>; }
    else { OUT.X<0> = GPR[rb]; }
}
else { OUT.X<0> = 0; }

OUT.Y<id> = SIGN_EXTEND_15(c2);
OUT.ALU_MODE<id> = ALU_ADD;
}
ACTIVATION { alu, writeback }
DOCUMENTATION("LA") { Load value of displacement address into ra }
}

ALIAS OPERATION la_simple<id> IN pipe.DC {
DECLARE {
    GROUP ra = { reg };
    GROUP c2 = { imm17 };
    INSTANCE alu<id>, writeback<id>;
}
// la opcode := 5
CODING { 0b00101 ra 0b0[5] c2 }
SYNTAX { "la" ~" " ra "," c2 }
BEHAVIOR {
    OUT.X<id> = 0;
    OUT.Y<id> = SIGN_EXTEND_15(c2);
    OUT.ALU_MODE<id> = ALU_ADD;
}
ACTIVATION { alu, writeback }
DOCUMENTATION("LA") { Load value of absolute address into ra; rb is
register
    0 }
}

/***** ST *****/
*           ST
*   31..27  26..22 21..17 16          ...          0   *
*   ----- -----
*   | Op   |   ra   |   rb   |           c2           |   *
*   ----- -----
*   *
***** OPERATION st IN pipe.DC {
DECLARE {
    GROUP ra,rb = { reg };
    GROUP c2 = { imm17 };
    INSTANCE alu<0>, mem_access_store;
}
// st opdcode := 3

```

```

CODING { 0b00011 ra rb c2 }
SYNTAX { "st" ~" " ra "," c2 "(" rb ")" }
BEHAVIOR {
    if(rb != 0) {
        // nop, stop, een, edi, rfi, ri, svi, st, and str
        // dont effect ra in path 0 (or have already been written to in
        // DC stage)
        if((WB.IN.ra0<0> == rb) &&
            (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
            WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
            WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
            WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
            WB.IN.op0<0> != STR))
            { OUT.X<0> = WB.IN.Z<0>; }
        // nop and stop and branch dont effect ra in path 1
        else if((WB.IN.ra0<1> == rb) &&
            (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
            WB.IN.op0<0> != BRANCH))
            { OUT.X<0> = WB.IN.Z<1>; }
        else { OUT.X<0> = GPR[rb]; }
    }
    else { OUT.X<0> = 0; }

    OUT.Y<0> = SIGN_EXTEND_15(c2);

    // if WB has a newer version of MD to be stored than the GPR
    // load it into MD out else load the GPR value
    // nop, stop, een, edi, rfi, ri, svi, st, and str
    // dont effect ra in path 0 (or have already been written to in DC
    // stage)
    if((WB.IN.ra0<0> == ra) &&
        (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
        WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
        WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
        WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
        WB.IN.op0<0> != STR))
        { OUT.MD = WB.IN.Z<0>; }
    // nop and stop dont effect ra in path 1
    else if((WB.IN.ra0<1> == ra) &&
        (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
        WB.IN.op0<0> != BRANCH))
        { OUT.MD = WB.IN.Z<1>; }
    else { OUT.MD = GPR[ra]; }

    OUT.ALU_MODE<0> = ALU_ADD;
}
ACTIVATION { alu, mem_access_store }
DOCUMENTATION("ST") { Store into displacement address }
}

ALIAS OPERATION st_simple IN pipe.DC {
DECLARE {
    GROUP ra = { reg };
    GROUP c2 = { imm17 };
    INSTANCE alu<0>, mem_access_store;
}

```

```

// st opdcode := 3
CODING { 0b00011 ra 0b0[5] c2 }
SYNTAX { "st" ~" " ra "," c2 }
BEHAVIOR {
    OUT.X<0> = 0;
    OUT.Y<0> = SIGN_EXTEND_15(c2);

    // if WB has a newer version of MD to be stored than the GPR
    // load it into MD out else load the GPR value
    // nop, stop, een, edi, rfi, ri, svi, st, and str
    // dont effect ra in path 0 (or have already been written to in DC
    // stage)
    if((WB.IN.ra0<0> == ra) &&
        (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
         WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
         WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
         WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
         WB.IN.op0<0> != STR))
    { OUT.MD = WB.IN.Z<0>; }
    // nop and stop dont effect ra in path 1
    else if((WB.IN.ra0<1> == ra) &&
            (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
             WB.IN.op0<0> != BRANCH))
    { OUT.MD = WB.IN.Z<1>; }
    else { OUT.MD = GPR[ra]; }

    OUT.ALU_MODE<0> = ALU_ADD;
}
ACTIVATION { alu, mem_access_store }
DOCUMENTATION("ST") { Store into absolute address; rb is register 0 }
}

/******************
*
*          MEM ACCESS Definition
*
*****************/
OPERATION mem_access_load IN pipe.MEM {
    BEHAVIOR { DMEM_LD(MEM.OUT.Z<0>, (uint32)MEM.IN.Z<0>); }
}

OPERATION mem_access_store IN pipe.MEM {
    BEHAVIOR { DMEM_ST(MEM.IN.MD, (uint32)MEM.IN.Z<0>); }
}

```

branch.lisa

```

/******************
*
*          Branch Instructions
*
*****************/
#include "defines.h"

```

```

/*
*      Branch
* 31..27 26..22 21..17 16..12 11      ...      3 2..0
* -----|-----|-----|-----|-----|-----|-----|
* | Op   |       | rb    | rc   | (c3) unused |Cond  |
* -----|-----|-----|-----|-----|-----|-----|
*                                         *
*/
OPERATION branch IN pipe.DC {
    DECLARE {
        GROUP rb,rc = { reg };
        GROUP c3 = { brnv || br || brzr || brnz || brpl || brmi };
        INSTANCE Branch_Logic;
    }
    // opcode = 8
    CODING { 0b01000 0b0[5] rb rc 0b0[9] c3 }
    SYNTAX { c3 }
    ACTIVATION { Branch_Logic }
    DOCUMENTATION("BRANCH") { Branch to target in rb if rc satisfies
        condition c3 }
}

OPERATION brnv {
    DECLARE { REFERENCE rb,rc; }
    // cond = 0
    CODING { 0b000 }
    SYNTAX { "brnv" }
    EXPRESSION { BRNV }
    DOCUMENTATION { branch never }
}

OPERATION br {
    DECLARE { REFERENCE rb,rc; }
    // cond = 1
    CODING { 0b001 }
    SYNTAX { "br" ~" " rb}
    EXPRESSION { BR }
    DOCUMENTATION { branch unconditionally to rb }
}

OPERATION brzr {
    DECLARE { REFERENCE rb,rc; }
    // cond = 2
    CODING { 0b010 }
    SYNTAX { "brzr" ~" " rb "," rc }
    EXPRESSION { BRZR }
    DOCUMENTATION { branch to rb if rc is zero }
}

OPERATION brnz {
    DECLARE { REFERENCE rb,rc; }
    // cond = 3
    CODING { 0b011 }
    SYNTAX { "brnz" ~" " rb "," rc }
}

```

```

EXPRESSION { BRNZ }
DOCUMENTATION { branch to rb if rc is non-zero }
}

OPERATION brpl {
    DECLARE { REFERENCE rb,rc; }
    // cond = 4
    CODING { 0b100 }
    SYNTAX { "brpl" ~" " rb "," rc }
    EXPRESSION { BRPL }
    DOCUMENTATION { branch to rb if rc is positive or zero
        (sign is plus) }
}

OPERATION brmi {
    DECLARE { REFERENCE rb,rc; }
    // cond = 5
    CODING { 0b101 }
    SYNTAX { "brmi" ~" " rb "," rc }
    EXPRESSION { BRMI }
    DOCUMENTATION { branch to rb if rc is negative (sign is minus) }
}

/*
*****
*          Branch Link
*  31..27  26..22 21..17 16..12 11      ...      3 2..0  *
*  -----|-----|-----|-----|-----|-----|-----|-----|-----*
*  | Op   | ra   | rb   | rc   | (c3) unused |Cond | *
*  -----|-----|-----|-----|-----|-----|-----|-----|-----*
*  *
*****/
OPERATION branchl IN pipe.DC {
    DECLARE {
        GROUP ra,rb,rc = { reg };
        GROUP c3 = { brlnv || brl || brlzs || brlnz || brlpl || brlmi };
        INSTANCE writeback<1>, Branch_Logic;
    }
    // opcode = 9
    CODING { 0b01001 ra rb rc 0b0[9] c3 }
    SYNTAX { c3 }
    BEHAVIOR {
        OUT.Z<1> = IN.PC2;
    }
    ACTIVATION { Branch_Logic, writeback }
    DOCUMENTATION("BRANCH LINK") { Branch to rb if rc satisfies c3, and
        save PC in ra }
}

OPERATION brlnv {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 0
    CODING { 0b000 }
    SYNTAX { "brlnv" ~" " ra }
    EXPRESSION { BRNV }
}

```

```

DOCUMENTATION { Do not branch, but save PC in ra }
}

OPERATION brl {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 1
    CODING { 0b001 }
    SYNTAX { "brl" ~" " ra "," rb }
    EXPRESSION { BR }
    DOCUMENTATION { Branch unconditionally to rb, and save PC in ra }
}

OPERATION brlzs {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 2
    CODING { 0b010 }
    SYNTAX { "brlzs" ~" " ra "," rb "," rc }
    EXPRESSION { BRZR }
    DOCUMENTATION { Branch to rb if rc is zero, and save PC in ra }
}

OPERATION brlnz {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 3
    CODING { 0b011 }
    SYNTAX { "brlnz" ~" " ra "," rb "," rc }
    EXPRESSION { BRNZ }
    DOCUMENTATION { Branch to rb if rc is non-zero, and save PC in ra }
}

OPERATION brlpl {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 4
    CODING { 0b100 }
    SYNTAX { "brlpl" ~" " ra "," rb "," rc }
    EXPRESSION { BRPL }
    DOCUMENTATION { Branch to rb if rc is positive, and save PC in ra }
}

OPERATION brlmi {
    DECLARE { REFERENCE ra,rb,rc; }
    // cond = 5
    CODING { 0b101 }
    SYNTAX { "brlmi" ~" " ra "," rb "," rc }
    EXPRESSION { BRMI }
    DOCUMENTATION { Branch to rb if rc is negative, and save PC in ra }
}

```

```
*****
*                                *
*          Branch Logic          *
*                                *
*****
```

```

OPERATION Branch_Logic POLL IN pipe.DC {
    DECLARE {
        REFERENCE c3;
        REFERENCE rb,rc;
    }
    BEHAVIOR {
        uint32 rb2;
        int32 rc2;

        char br_cond;
        br_cond = c3 & 0x1f;

        // =====
        //
        // =====
        // st, str, nop, stop, een, edi, svi, ri and rfi
        // dont use ra in path 0
        if((rc == EX.IN.ra0<0>) &&
           (EX.IN.op0<0> != ST && EX.IN.op0<0> != STR &&
            EX.IN.op0<0> != NOP && EX.IN.op0<0> != STOP &&
            EX.IN.op0<0> != EEN && EX.IN.op0<0> != EDI &&
            EX.IN.op0<0> != SVI && EX.IN.op0<0> != RI &&
            EX.IN.op0<0> != RFI))
        {
            pipe.DC.IN.stall();
            pipe.EX.IN.stall();
        }

        // only nops, stops and branches dont use ra in path 1
        else if((rc == EX.IN.ra0<1>) &&
                 (EX.IN.op0<1> != NOP && EX.IN.op0<1> != STOP &&
                  EX.IN.op0<1> != BRANCH))
        {
            pipe.DC.IN.stall();
            pipe.EX.IN.stall();
        }

        else {
            // st, str, nop, stop, een, edi, svi, ri and rfi
            // dont use ra in path 0
            if((rc == MEM.IN.ra0<0>) &&
               (MEM.IN.op0<0> != ST && MEM.IN.op0<0> != STR &&
                MEM.IN.op0<0> != NOP && MEM.IN.op0<0> != STOP &&
                MEM.IN.op0<0> != EEN && MEM.IN.op0<0> != EDI &&
                MEM.IN.op0<0> != SVI && MEM.IN.op0<0> != RI &&
                MEM.IN.op0<0> != RFI))
            { rc2 = MEM.IN.Z<0>; }

            // only nops, stops, branches dont use ra in path 1
            else if((rc == MEM.IN.ra0<1>) &&
                     (MEM.IN.op0<1> != NOP && MEM.IN.op0<1> != STOP &&
                      MEM.IN.op0<1> != BRANCH))
            { rc2 = MEM.IN.Z<1>; }
        }
    }
}

```

```

// st, str, nop, stop, een, edi, svi, ri and rfi
// dont use ra in path 0
else {
    if((rc == WB.IN.ra0<0>) &&
       (WB.IN.op0<0> != ST && WB.IN.op0<0> != STR &&
        WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
        WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
        WB.IN.op0<0> != SVI && WB.IN.op0<0> != RI &&
        WB.IN.op0<0> != RFI))
    { rc2 = WB.IN.Z<0>; }

    // only nops, stops, and branches dont use ra in path 1
    else if((rc == WB.IN.ra0<1>) &&
            (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
             WB.IN.op0<1> != BRANCH))
    { rc2 = WB.IN.Z<1>; }

    else { rc2 = GPR[rc]; }
}
}

// =====
//
// =====
// st, str, nop, stop, een, edi, svi, ri and rfi
// dont use ra in path 0
if((rb == EX.IN.ra0<0>) &&
   (EX.IN.op0<0> != ST && EX.IN.op0<0> != STR &&
    EX.IN.op0<0> != NOP && EX.IN.op0<0> != STOP &&
    EX.IN.op0<0> != EEN && EX.IN.op0<0> != EDI &&
    EX.IN.op0<0> != SVI && EX.IN.op0<0> != RI &&
    EX.IN.op0<0> != RFI))
{
    pipe.DC.IN.stall();
    pipe.EX.IN.stall();
}

// only nops, stops, and branches dont use ra in path 1
else if((rb == EX.IN.ra0<1>) &&
        (EX.IN.op0<1> != NOP && EX.IN.op0<1> != STOP &&
         EX.IN.op0<1> != BRANCH))
{
    pipe.DC.IN.stall();
    pipe.EX.IN.stall();
}

else {

    // st, str, nop, stop, een, edi, svi, ri and rfi
    // dont use ra in path 0
    if((rb == MEM.IN.ra0<0>) &&
       (MEM.IN.op0<0> != ST && MEM.IN.op0<0> != STR &&
        MEM.IN.op0<0> != NOP && MEM.IN.op0<0> != STOP &&
        MEM.IN.op0<0> != EEN && MEM.IN.op0<0> != EDI &&
        MEM.IN.op0<0> != SVI && MEM.IN.op0<0> != RI &&
        MEM.IN.op0<0> != RFI))
    { rc2 = MEM.IN.Z<0>; }

    // only nops, stops, and branches dont use ra in path 1
    else if((rb == MEM.IN.ra0<1>) &&
            (MEM.IN.op0<1> != NOP && MEM.IN.op0<1> != STOP &&
             MEM.IN.op0<1> != BRANCH))
    { rc2 = MEM.IN.Z<1>; }

    else { rc2 = GPR[rb]; }
}
}

```

```

        MEM.IN.op0<0> != SVI && MEM.IN.op0<0> != RI     &&
        MEM.IN.op0<0> != RFI))
{ rb2 = MEM.IN.Z<0>; }

// only nops, stops, and branches dont use ra in path 1
else if((rb == MEM.IN.ra0<1>) &&
        (MEM.IN.op0<1> != NOP && MEM.IN.op0<1> != STOP &&
         MEM.IN.op0<1> != BRANCH))
{ rb2 = MEM.IN.Z<1>; }

else {
    // st, str, nop, stop, een, edi, svi, ri and rfi
    // dont use ra in path 0
    if((rb == WB.IN.ra0<0>) &&
        (WB.IN.op0<0> != ST && WB.IN.op0<0> != STR &&
         WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
         WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
         WB.IN.op0<0> != SVI && WB.IN.op0<0> != RI &&
         WB.IN.op0<0> != RFI))
    { rb2 = WB.IN.Z<0>; }

    // only nops, stops, and branches dont use ra in path 1
    else if((rb == WB.IN.ra0<1>) &&
            (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
             WB.IN.op0<1> != BRANCH))
    { rb2 = WB.IN.Z<1>; }

    else { rb2 = GPR[rb]; }
}
}

// =====
//          BRANCH Logic
// =====
switch(br_cond) {
    case BRNV: // Branch never
        BSET = false;
        branch_set = false;
        break;
    case BR: // Branch always
        BPC = rb2;
        BSET = true;
        branch_set = true;
        pipe.FE.OUT.flush();
        break;
    case BRZR: // Branch when rb is zero
        if(rc2 == 0) {
            BPC = rb2;
            BSET = true;
            branch_set = true;
            pipe.FE.OUT.flush();
        }
        else { BSET = false; }
        break;
    case BRNZ: // branch when rb != 0
        if(rc2 != 0) {

```

```

        BPC = rb2;
        BSET = true;
        branch_set = true;
        pipe.FE.OUT.flush();
    }
    else { BSET = false; }
    break;
case BRPL: // Branch when rb > 0
    if(rc2 > 0) {
        BPC = rb2;
        BSET = true;
        branch_set = true;
        pipe.FE.OUT.flush();
    }
    else { BSET = false; }
    break;
case BRMI: // Branch when rb < 0
    if(rc2 < 0) {
        BPC = rb2;
        BSET = true;
        branch_set = true;
        pipe.FE.OUT.flush();
    }
    else { BSET = false; }
    break;
default: // DEFAULT
    BSET = false;
    branch_set = false;
    break;
}
}
}

```

Miscellaneous.lisa

```

*****
*
*          Miscellaneous Instructions
*
*****
#include "defines.h"

*****
*
*          NOP/STOP
*      31..27  26           ...           0
*  -----| Op |           Unused           |
*  -----|-----|-----|-----|-----|
*          *
*****
OPERATION nop<id> IN pipe.DC {
    // binary image: 0b0[32] means 32 zeros
    CODING   { 0b0[32] }
    SYNTAX   { "nop" }
    BEHAVIOR

```

```

{
#pragma analyze(off)
    // nothing to do here
    fprintf(stderr,"Nothing to do here\n");
#pragma analyze(on)
}
DOCUMENTATION("NOP") { No operation. Used to insert pipeline bubble }

OPERATION stop<id> IN pipe.DC {
    // opcode = 31
    CODING { 0b11111 0b0[27] }
    SYNTAX { "stop" }
    BEHAVIOR
    {
        // set Run to zero, halting the machine
        RUN = false;
    }
    DOCUMENTATION("STOP") { Set RUN to zero, halting the machine }
}

/*****************
 *              *
 *      Interrupt Instructions      *
 *              *
*****************/
OPERATION een IN pipe.DC {
    // opcode = 10
    CODING { 0b01010 0b0[27] }
    SYNTAX { "een" }
    BEHAVIOR
    {
        // Exception enable. Set overall exception enable
        IE = true;
    }
    DOCUMENTATION("EEN") { Exception enable. Set overall exception enable
        bit }
}

OPERATION edi IN pipe.DC {
    // opcode = 11
    CODING { 0b01011 0b0[27] }
    SYNTAX { "edi" }
    BEHAVIOR
    {
        // Exception disable. Clear overall exception enable
        IE = false;
    }
    DOCUMENTATION("EDI") { Exception disable. Clear overall exception
        enable }
}

OPERATION rfi IN pipe.DC {
    // opcode = 30
    CODING { 0b11110 0b0[27] }
    SYNTAX { "rfi" }
}

```

```

BEHAVIOR
{
    // Return from interrupt. FPC <- IPC; enable exceptions
    RFIPC = true;
    pipe.FE.OUT.flush();
}
DOCUMENTATION("RFI") { Return from interrupt. PC <- IPC; enable
exceptions }
}

OPERATION svi IN pipe.DC {
DECLARE {
    GROUP ra,rb = { reg };
}
// opcode = 16
CODING { 0b10000 ra rb 0b0[17] }
SYNTAX { "svi" ~" " ra "," rb }
BEHAVIOR
{
    // Save II and IPC in ra and rb respectively
    // GPR[ra] <- II && GPR[rb] <- IPC
    GPR[ra] = II;
    GPR[rb] = IPC;
}
DOCUMENTATION("SVI") { Save II and IPC in ra and rb, respectively }
}

OPERATION ri IN pipe.DC {
DECLARE {
    GROUP ra,rb = { reg };
}
// opcode = 17
CODING { 0b10001 ra rb 0b0[17] }
SYNTAX { "ri" ~" " ra "," rb }
BEHAVIOR
{
    // Restore II and IPC from ra and rb, respectively
    // II <- GPR[rb] && IPC <- GPR[rb]
    // II = GPR[ra];
    // IPC = GPR[rb];
    if((WB.IN.ra0<0> == ra) &&
       (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
        WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
        WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
        WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
        WB.IN.op0<0> != STR))
    { II = WB.IN.Z<0>; }
    // nop and stop and branch dont effect ra in path 1
    else if((WB.IN.ra0<1> == ra) &&
            (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
             WB.IN.op0<0> != BRANCH))
    { II = WB.IN.Z<1>; }
    else { II = GPR[ra]; }

    // nop, branch, stop, een, edi, rfi, ri, svi, st, and str
    // dont effect ra in path 0 (or have already been written to in DC
}

```

```

    // stage)
    if((WB.IN.ra0<0> == rb) &&
       (WB.IN.op0<0> != NOP && WB.IN.op0<0> != STOP &&
        WB.IN.op0<0> != EEN && WB.IN.op0<0> != EDI &&
        WB.IN.op0<0> != RFI && WB.IN.op0<0> != RI &&
        WB.IN.op0<0> != SVI && WB.IN.op0<0> != ST &&
        WB.IN.op0<0> != STR))
    { IPC = WB.IN.Z<0>; }
    // nop and stop dont effect ra in path 1
    else if((WB.IN.ra0<1> == rb) &&
            (WB.IN.op0<1> != NOP && WB.IN.op0<1> != STOP &&
             WB.IN.op0<0> != BRANCH))
    { IPC = WB.IN.Z<1>; }
    else { IPC = GPR[rb]; }
}
DOCUMENTATION("RI") { Restore II and IPC from ra and rb,
                     respectively }
}

```

Immediate.lisa

```

/*********************************************
*
*           Immediate Implementation
*
*****************************************/
#include "defines.h"

IMMEDIATE imm5
{
    // 5-bit immediate value
    DECLARE { LABEL value; }
    CODING { value=0bx[5] }
    SYNTAX { SYMBOL( value=#U5 ) }
    EXPRESSION { value }
    DOCUMENTATION { 5-bit unsigned immediate value }
}

IMMEDIATE imm17
{
    // 17-bit immediate value
    DECLARE { LABEL value; }
    CODING { value=0bx[17] }
    SYNTAX { SYMBOL( value=#S17 ) }
    EXPRESSION { value }
    DOCUMENTATION { 17-bit signed value/(label) }
}

// This operation describes a simple register
IMMEDIATE reg
{
    DECLARE { LABEL idx; } // A label (=variable) for the reg addr
    SYNTAX { "r" ~idx=#U } // The assembly is made up by the reg addr
}

```

```
CODING { idx=0bx[5] }      // The reg addr is encoded within 5 bits
EXPRESSION { idx }         // This operation returns the reg addr
DOCUMENTATION { General Purpose Register r0 - r31 }
}
```

APPENDIX E

AYK-14 Code

main.lisa

```
*****  
AYK-14 Implementation  
  
INSTRUCTION FORMATS  
1) RL (Register - Literal) (CP instructions only)  
   -- Single length instructions  
   15      ...      10 9 8 7      ...      4 3      ...      0  
-----  
|       OP CODE       | f |       a       |       m       |  
-----  
OP CODE, f - 8 bit code specifying the operation;  
            RL format only  
a - General register designator  
m - 4-bit literal constant  
  
2) RR (Register - Register)  
   -- Single length instructions  
   The a-designator selects the R[a] register and the  
   m-designator selects the R[m] register  
  
3) RI (Register - Indirect Memory) (CP instructions only)  
   -- Single length instructions  
   a) Type 1 - Local Jumps  
      -- effective jump address found by adding P (program  
register)  
      to the displacement field, d  
   b) Type 2 - Other operations using general registers  
      and a memory reference  
  
4) RK (Register - Constant (Immediate Operand))  
   -- Double length instructions  
   a) when m=0, the operand is equal to y  
   b) when m!=0, the operand is sum of y and R[m]  
  
5) RX (Register - Memory with or without indexing)  
   -- Double length instructions  
   a) whole word(16-bit) operations  
      1) when m=0, the operand is equal to y (direct addressing  
with m=0)  
      2) when m=1-7,9,11,13 or 15, the operand is sum of y and  
R[m]  
         (direct addressing with m=1-7,9,11,13 or 15)  
      3) when m=8,10,12 or 14,  
   b) byte(8-bit) operations
```

RR RI, Type 2											
15	...	10	9	8	7	...	4	3	...	0	
<hr/> OP CODE f a m											

RI, Type 1										
15	...	10	9	8	7	...	4	3	...	0

	OP CODE		f		a		m		b	

RK, RX										
15	...	10	9	8	7	...	4	3	...	0
-----					-----					
	OP	CODE		f		a		m		
-----					-----					
	Y									

OP CODE - code specifying the operation

f - format designator

00 -> Format RR

01 -> Format RI, Type 1 and Type 2

10 -> Format RK

11 -> Format RX

a - General register or subfunction designator

m - General register or subfunction designator

d - Displacement value (two's compliment)

y - Address or arithmetic constant

STATUS REGISTER 1														
2	15	14	13	12	11	10	9	8	7	6	5	4	3	
	0													
<hr/>														
	EMD GRSD PTD BROMMD CD OD CC FPU/O FPR EIOP													
Class		DMA												
<hr/>														

EMD = Executive Mode Designator; 0 = Executive Mode
GRSD = General Register Stack Designator; 0 = Stack 0
PTD = Processor Type Designator; 0 = CPU/SCP/VPM Master; 1 =
EIOP/IOP/SCP/VPM slave
BROMMD = Bootstrap ROM Mode Designator; 0 = Bootstrap ROM Mode
CD = Carry Designator
OD = Overflow Designator
CC = Condition Codes
FPU/O = F.P. Underflow/overflow interrupt; 0 = enable
FPR = F.P. Residue; 1 = enable
EIOP =
Class = Bit 1 = 0, Class III locked out

```

        Bit 2 = 0, Class II locked out
        Bit 3 = 0, Class I locked out
DMA = 0 Disable DMA; 1 Enable DMA

                STATUS REGISTER 2
15 14 13 12 11 10 9   8 7     ...      0
-----
| m=E | m=C | m=A | m=8 | Interrupt ID info |
-----
Bits 15-8 (in two bit fields, 15-14, 13-12, 11-10, and 9-8)
00 := Direct Addressing with indexing
01 := Direct Addressing with indexing
10 := Indirect Addressing without indexing; IW 1 at Y=y
11 := Indirect Addressing with indexing; IW 1 at Y=y+R[m]

*****/*****
#include "defines.h"

// Specify a version string which is contained later
// in the generated tools
VERSION("AYK-14, 2006.1.1")

RESOURCE
{
    MEMORY_MAP
    {
        RANGE(0x0000,0x0fff) -> prog_mem[(31..0)];
        RANGE(0x1000,0x1fff) -> data_mem[(31..0)];
    }

#ifndef LT_PROCESSOR_GENERATOR // Only run if RTL is generated
    // 0x10000 16bit words of program memory
    // FLAGS are set to R|X meaning that prog_mem is readable and
    // executable
    RAM uint16 prog_mem
    {
        SIZE(0x1000);
        BLOCKSIZE(16,8);
        FLAGS(R|X);
    };

    // 0x10000 16bit words of data memory
    // FLAGS are set to R|W meaning that data mem is
    // readable and writeable
    RAM uint16 data_mem
    {
        SIZE(0x1000);
        BLOCKSIZE(16,8);
        FLAGS(R|W);
    };
#else // run for the debugger
    // 0x10000 8bit words of program memory
    // FLAGS are set to R|X meaning that prog_mem is readable and
    // executable
    RAM uint8 prog_mem
    {

```

```

        SIZE(0x1000);
        BLOCKSIZE(8,2);
        FLAGS(R|X);
    };

    // 0x10000 8bit words of data memory
    // FLAGS are set to R|W meaning that data mem is
    // readable and writeable
    RAM uint8 data_mem
    {
        SIZE(0x1000);
        BLOCKSIZE(8,2);
        FLAGS(R|W);
    };
#endif

// Register file (2 stacks of 16 General Purpose Registers)
REGISTER TClocked<int16> R0[0..15]; //general register stack 0
REGISTER TClocked<int16> R1[0..15]; //general register stack 1

// Program Counter (PC) (Program Address Register)
PROGRAM_COUNTER TClocked<uint32> P;

// Status Register 1
REGISTER TClocked<uint16> SR1;
// Status Register 2
REGISTER TClocked<uint16> SR2;

// Byte Signals to Memory
PIN OUT bool UB; //Selects Upper Byte
PIN OUT bool LB; //Selects Lower Byte

uint16 IRO; // holds first 16 bits of instruction
uint16 IR1; // holds address extension (16-bits)

uint16 tmp_P; // temp Program counter wire

bool clr_OD; //Overflow
bool clr_CD; //Carry

REGISTER TClocked<bool> bset; //branch set flag register
REGISTER TClocked<uint16> bpc; //branch program counter

// alu signals
int16 op1;
int16 op2;
int32 dst;
uint8 alu_mode;

PIPELINE pipe = { ONE };

// UNIT declarations for RTL generation
UNIT U_FETCH IN pipe.ONE {
    OPERATIONS (fetch);
}

```

```

        REGISTERS (IR0, IR1);
    };
    UNIT U_DECODE IN pipe.ONE {
        OPERATIONS (decode);
        REGISTERS (P);
    };
    UNIT U_ALU_DC IN pipe.ONE {
        OPERATIONS (A, IROR, DROR, AR, LLRS, LARD, LALS, LALD, LSU, LA);
    };
    UNIT U_ALU_EX IN pipe.ONE {
        OPERATIONS (alu);
        REGISTERS (op1, op2, dst);
    };
    UNIT U_LOGIC_DC IN pipe.ONE {
        OPERATIONS (ANDK, ORR);
    };
    UNIT U_LOAD IN pipe.ONE {
        OPERATIONS (LK, L, LD, LI, LR, LL, SBR, ZBR, LSOR);
    };
    UNIT U_STORE IN pipe.ONE {
        OPERATIONS (S, SD, SZ, BS, SXI, SI, SSOR, SM);
    };
    UNIT U_COMPARE IN pipe.ONE {
        OPERATIONS (CK, C, LC, CBR);
    };
    UNIT U_JUMP IN pipe.ONE {
        OPERATIONS (XJ, LJE, LJNE, LJLS, LJGE, JLR_RK, JLR_RX, LJ, LPR);
        REGISTERS (bset, bpc);
    };
    UNIT U_SR1 IN pipe.ONE {
        OPERATIONS (set_SR1);
    };
    UNIT U_WRITEBACK IN pipe.ONE {
        OPERATIONS (writeback);
    };
}

```

```

OPERATION reset {
    BEHAVIOR {
        // Reset all resource variables to a predefined start point
        int i;

        // Zero out register files
        for(i = 0; i < 16; i++) {
            R0[i] = 0;
            R1[i] = 0;
        }

        // Set program counter to program entry point
        P = LISA_PROGRAM_COUNTER;

        SR1 = 0;
        SR2 = 0;
        clr_OD = 0;
        clr_CD = 0;
    }
}

```

```

        op1 = 0;
        op2 = 0;
        dst = 0;
        bset = false;
        bpc = 0;
    }
}

OPERATION main {
    DECLARE {
        INSTANCE fetch;
    }
    BEHAVIOR {
        // Add pipeline control for future pipelined versions
        PIPELINE(pipe).execute();
        PIPELINE(pipe).shift();
    }
    ACTIVATION { fetch }
}

OPERATION fetch IN pipe.ONE {
    DECLARE {
        INSTANCE decode;
    }
    BEHAVIOR {
        int16 tmp_P0;

        if(bset) {
            tmp_P0 = bpc;
            bset = false;
        }
        else { tmp_P0 = P; }

#ifndef LT_PROCESSOR_GENERATOR // Only run if RTL is generated
    IRO = prog_mem[ tmp_P0 ];
    IR1 = prog_mem[ tmp_P0 + 1 ];
#else // Run for the debugger
    uint8 tmp1 = prog_mem[ tmp_P0 ];
    uint8 tmp2 = prog_mem[ tmp_P0 + 1 ];
    IRO = (tmp2<<8) | tmp1;

    uint8 tmp3 = prog_mem[ tmp_P0 + 2 ];
    uint8 tmp4 = prog_mem[ tmp_P0 + 3 ];
    IR1 = (tmp4<<8) | tmp3;
#endif
        tmp_P = tmp_P0;
    }
    ACTIVATION { decode }
}

// Variable length instruction words
// PC := Fetch Address of the current instruction
// IRO := Instruction word data, high part
// IR1 := Instruction word data, low part (addr extension)
INSTRUCTION decode IN pipe.ONE {
    DECLARE {

```

```

ENUM format = { x16, x32 };
GROUP insn_x16 = { nop || LR || LL || SBR || ZBR || SXI || LI ||
                   SI || IROR || DROR || AR || LLRS || LARD ||
                   LALS || LALD || LSU || LA || LC || CBR ||
                   LJ || LPR || LJE || LJNE || LJLS || LJGE ||
                   ORR || LSOR || SSOR || OCR || TCR || PR };
GROUP insn_x32 = { LK || LK_simple || L || L_simple ||
                   LD || LD_simple || S || S_simple ||
                   SD || SD_simple || SZ || SZ_simple ||
                   BS || BS_simple || SM || SM_simple ||
                   A || A_simple || JLR_RX || JLR_RX_simple ||
                   JLR_RK || JLR_RK_simple || CK || CK_simple ||
                   ANDK || ANDK_simple || XJ || XJ_simple || C };
}

// SWITCH on ENUM to implement multiple roots
// for different instruction word-lengths,
// decoder will select the correct one
SWITCH (format) {
    CASE x16:
    {
        //16-bit instructions
        CODING AT ( P ) { IRO == insn_x16 }
        SYNTAX { insn_x16 }

#ifdef LT_PROCESSOR_GENERATOR // Only run if RTL is generated
        BEHAVIOR { P = tmp_P + 1; }
#else
        BEHAVIOR { P = tmp_P + 2; }
#endif
        ACTIVATION { insn_x16 }
    }
    CASE x32:
    {
        //32-bit instructions := 16-bit Instruction with 16-bit address
        extension
        CODING AT ( P ) { (IRO == insn_x32=[16..31]) && (IR1 ==
insn_x32=[0..15]) }
        SYNTAX { insn_x32 }
        // update PC by four since next fetch is not an instruction
        // it was an address extension (connected to byte size memory)
#ifdef LT_PROCESSOR_GENERATOR // Only run if RTL is generated
        BEHAVIOR { P = tmp_P + 2; }
#else
        BEHAVIOR { P = tmp_P + 4; }
#endif
        ACTIVATION { insn_x32 }
    }
}

OPERATION nop IN pipe.ONE {
    CODING { 0b0[16] }
    SYNTAX { "nop" }
    BEHAVIOR {
        // do nothing
    }
}

```

```

***** Set Status Register 1 (SR1) *****
OPERATION set_SR1 IN pipe.ONE {
    BEHAVIOR {
        // define 3 16-bit ints for CC (Condition Code) OD (Overflow
        // Designator) and CD (Carry Designator). This is the best way I
        // could think of to change bits of an integer in C was to OR them
        // together.
        int16 CC;
        int16 OD;
        int16 CD;

        bool sign_op1 = op1 >> 15; // tie sign_op1 to bit 15 op op1 signal
                                    // (sign bit)
        bool sign_op2 = op2 >> 15; // tie sign_op2 to bit 15 op op2 signal
                                    // (sign bit)
        bool sign_dst = dst >> 15; // tie sign_dst to bit 15 op dst signal
                                    // (sign bit)
        bool carry = dst >> 16; // tie carry to bit 16 of dst (carry bit)

        // Condition Code (CC)
        if((int16)dst == 0) { CC = 0x0000; }
        else if((int16)dst > 0) { CC = 0x0100; }
        else { CC = 0x0300; }

        // Overflow Designator (OD)
        if(clr_OD) { OD = 0x0000; } // Clear bit 10 of SR1 (Overflow
                                    // Designators)
        else {
            if((sign_op1 && sign_op2 && !sign_dst) ||
               (!sign_op1 && !sign_op2 && sign_dst))
                { OD = 0x0400; }
            else { OD = 0x0000; }
        }

        // Carry Designator (CD)
        if(clr_CD) { CD = 0x0000; } // Clear bit 11 of SR1 (Carry
                                    // Designators)
        else {
            if(carry) { CD = 0x0800; }
            else { CD = 0x0000; }
        }

        // AND out bits 8-11 of Status Register 1 (SR1) and then
        // OR the new bits together to update SR1
        SR1 = (SR1 & (0xf0ff)) | (CC | CD | OD);
    }
}

```

arithmetic.lisa

```
*****
```

```

*
*          Arithmetic Instructions
*
***** ****
#include "defines.h"

/***** x32 instructions *****/
// Add (index)
OPERATION A IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        GROUP y = { addr };
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 22, format := RX
    CODING { 0b010110 0b11 a m y }
    SYNTAX { "A" ~ " a , " y , " m }
    BEHAVIOR {
        int16 tmp_Y;
        // Add R[a] to operand Y and store into R[a]
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            if(m==0) { tmp_Y = y; }
            else { tmp_Y = y + R1[m]; }
            op1 = R1[a];
        }
        else { // GPR stack 0
            if(m==0) { tmp_Y = y; }
            else { tmp_Y = y + R0[m]; }
            op1 = R0[a];
        }
        op2 = data_mem[tmp_Y];
        alu_mode = ALU_ADD;
    }
    ACTIVATION { alu, writeback, set_SR1 }
    DOCUMENTATION("A") { This instruction adds the contents of memory
        address Y to the contents of R[a], stores the sum into R[a], and
        sets the condition code in accordance with the resulting quantity
        R[a]. The overflow and carry designators are set in accordance
        with the results of the operation. }
}
// syntax 2 for 'A' instruction
ALIAS OPERATION A_simple IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP y = { addr };
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 22, format := RX
    CODING { 0b010110 0b11 a 0b0[4] y }
    SYNTAX { "A" ~ " a , " y }
    BEHAVIOR {
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { op1 = R1[a]; } // GPR stack 1
        else { op1 = R0[a]; } // GPR stack 0
        op2 = data_mem[y];
        alu_mode = ALU_ADD;
    }
}

```

```

        }
    ACTIVATION { alu, writeback, set_SR1 }
}

/***** x16 instructions *****/
// Increase by 1 (Register)
OPERATION IROR IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 2, format := RR (Register-Register)
    CODING { 0b000010 0b00 a 0b1010 }
    SYNTAX { "IROR" ~" " a }
    BEHAVIOR {
        // Increment contents of R[a] by one
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { op1 = R1[a]; } // GPR stack 1
        else { op1 = R0[a]; } // GPR stack 0
        op2 = 1;
        alu_mode = ALU_ADD;
    }
    ACTIVATION { alu, writeback, set_SR1 }
    DOCUMENTATION("IROR") { This instruction adds one to
        the contents of R[a], stores the sum into R[a], and
        sets the condition code in accordance with the resulting
        quantity R[a]. The overflow and carry designators are set
        in accordance with the results of the operation }
}

// Decrease by 1 (Register)
OPERATION DROR IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 2, format := RR (Register-Register)
    CODING { 0b000010 0b00 a 0b1011 }
    SYNTAX { "DROR" ~" " a }
    BEHAVIOR {
        // Decrement contents of R[a] by one
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { op1 = R1[a]; } // GPR stack 1
        else { op1 = R0[a]; } // GPR stack 0
        op2 = 1;
        alu_mode = ALU_SUB;
    }
    ACTIVATION { alu, writeback, set_SR1 }
    DOCUMENTATION("DROR") { This instruction subtracts one from the
        contents of R[a], stores the difference into R[a], and sets
        the condition code in accordance with the resulting quantity in
        R[a]. The overflow and carry designators are set in accordance
        with the results of the operation }
}

// Add (Register)

```

```

OPERATION AR IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 22, format := RR (Register-Register)
    CODING { 0b010110 0b00 a m }
    SYNTAX { "AR" ~" " a "," m }
    BEHAVIOR {
        // Add R[a] to R[m] and store in R[a]
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            op1 = R1[a];
            op2 = R1[m];
        }
        else { // GPR stack 0
            op1 = R0[a];
            op2 = R0[m];
        }
        alu_mode = ALU_ADD;
    }
    ACTIVATION { alu, writeback, set_SR1 }
    DOCUMENTATION("AR") { This instruction adds the contents of R[m] to
        the contents of R[a], stores the sum into R[a], and sets the
        condition Code in accordance with the resulting quantity in R[a].
        The overflow and carry designators are set in accordance with the
        results of the operation. }
}

//logical right single shift
OPERATION LLRS IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP m = { imm4 }; // unsigned 4-bit literal constant
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 60, format := RL
    CODING { 0b111100 0b00 a m }
    SYNTAX { "LLRS" ~" " a "," m }
    BEHAVIOR {
        // right shift R[a] by count m and zero extend
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { op1 = R1[a]; } // GPR stack 1
        else { op1 = R0[a]; } // GPR stack 0
        op2 = m;
        alu_mode = ALU SHR;

        clr_OD = true;
        clr_CD = true;
    }
    ACTIVATION { alu, writeback, set_SR1 }
    DOCUMENTATION("LLRS") { This instruction shifts the contents of R[a]
        right by the number of places specified by bits 3-0 ( the m-
        designator) of the instruction with zeros extended to fill and sets
        the condition code in accordance with the resulting quantity in
        R[a]. The overflow and carry designators are cleared. }
}

```

```

}

//Algebraic right double shift
OPERATION LARD IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP m = { imm4 }; // unsigned 4-bit literal constant
    }
    //opcode := 60, format := RL
    CODING { 0b111100 0b11 a m }
    SYNTAX { "LARD" ~" " a "," m }
    BEHAVIOR {
        int32 temp;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            temp = ((uint16)R1[a] << 16) | ((uint16)R1[a+1]); // create 32
                // bit variable out of 2 16-bit registers
            temp = temp >> m; // Right shift the variable by count m and sign
                // extend
            R1[a+1] = temp; // R[a+1] holds lower 16-bits of shifted variable
            R1[a] = temp >> 16; // R[a] holds upper 16-bits of shifted
                // variable
        }
        else { // GPR stack 0
            temp = ((uint16)R0[a] << 16) | ((uint16)R0[a+1]); // create 32
                // bit variable out of 2 16-bit registers
            temp = temp >> m; // Right shift the variable by count m and sign
                // extend
            R0[a+1] = temp; // R[a+1] holds lower 16-bits of shifted variable
            R0[a] = temp >> 16; // R[a] holds upper 16-bits of shifted
                // variable
        }
    }

    // Set CC, OD and CD of Status Register 1
    int16 CC; //Condition Code
    if((int32)temp==0) { CC = 0x0000; }
    else if((int32)temp > 0) { CC = 0x0100; }
    else { CC = 0x0300; }

    int16 OD = 0x0000; //Overflow Designator
    int16 CD = 0x0000; //Carry Designator

    // Set SR1
    SR1 = (SR1 & (0xf0ff)) | (CC | OD | CD);
}
DOCUMENTATION("LARD") { This instruction shifts the double lenght
contents of R[a],R[a+1] right by the number of places specified by
bits 3-0 (the m-designator) of the instruction with the R[a] sign
extended to fill and sets the condition code in accordance with the
resulting double length quantity in R[a],R[a+1]. The overflow and
carry designators are cleared. R[a] must be an even-numbered
register. }

//Algebraic left single shift
OPERATION LALS IN pipe.ONE {

```

```

DECLARE {
    GROUP a = { reg };
    GROUP m = { imm4 }; // unsigned 4-bit literal constant
    INSTANCE alu, writeback, set_SR1;
}
//opcode := 61, format := RL
CODING { 0b111101 0b00 a m }
SYNTAX { "LALS" ~" " a "," m }
BEHAVIOR {
    // left shift R[a] by count m and zero extend to fill
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { op1 = R1[a]; } // GPR stack 1
    else { op1 = R0[a]; } // GPR stack 0
    op2 = m;
    alu_mode = ALU_SHL;

    clr_CD = true;
}
ACTIVATION { alu, writeback, set_SR1 }
DOCUMENTATION("LALS") { This instruction shifts the contents of R[a]
left by the number of places specified by bits 3-0 (the m
designator) of the instruction with zeros extended to fill and sets
the condition code in accordance with the resulting quantity in
R[a]. The overflow designator is set in accordance with the results
of the operation. The carry designator is cleared. }
}

//Algebraic left double shift
OPERATION LALD IN pipe.ONE {
DECLARE {
    GROUP a = { reg };
    GROUP m = { imm4 }; // unsigned 4-bit literal constant
}
//opcode := 61, format := RL
CODING { 0b111101 0b10 a m }
SYNTAX { "LALD" ~" " a "," m }
BEHAVIOR {
    uint32 temp;
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { // GPR stack 1
        temp = ((uint16)R1[a] << 16) | ((uint16)R1[a+1]); // create 32
                                                // bit variable out of 2 16-bit registers
        temp = temp << m; // left shift the variable by count m and zero
                            // fill
        R1[a+1] = temp; // R[a+1] holds lower 16-bits of shifted
                         // variable
        R1[a] = temp >> 16; // R[a] holds upper 16-bits of shifted
                           // variable
    }
    else { // GPR stack 0
        temp = ((uint16)R0[a] << 16) | ((uint16)R0[a+1]); // create 32
                                                // bit variable out of 2 16-bit registers
        temp = temp << m; // left shift the variable by count m and zero
                            // fill
        R0[a+1] = temp; // R[a+1] holds lower 16-bits of shifted
                         // variable
    }
}
}

```

```

        R0[a] = temp >> 16; // R[a] holds upper 16-bits of shifted
                           // variable
    }

    // Set CC, OD and CD of Status Register 1
    int16 CC; //Condition Code
    if((int32)temp==0) { CC = 0x0000; }
    else if((int32)temp > 0) { CC = 0x0100; }
    else { CC = 0x0300; }

    int16 OD; //Overflow Designator
    bool sign_op1 = op1 >> 15;
    bool sign_op2 = op2 >> 15;
    bool sign_dst = temp >> 15;
    if((sign_op1 && sign_op2 && !sign_dst) ||
       (!sign_op1 && !sign_op2 && sign_dst))
    { OD = 0x0400; }
    else { OD = 0x0000; }

    int16 CD = 0x0000; //Carry Designator

    // Set SR1
    SR1 = (SR1 & (0xf0ff)) | (CC | OD | CD);
}

DOCUMENTATION("LALD") { This instruction shifts the double lenght
contents of R[a],R[a+1] left by the number of places specified by
bits 3-0 (the m-designator) of the instruction with zeros extended
to fill and sets the condition code in accordance with the
resulting double length quantity in R[a],R[a+1]. The overflow
designator is set in accordance with the results of the operation.
The carry designator is cleared. R[a] must be an even-numbered
register. }

// Subtract Literal
OPERATION LSU IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP m = { imm4 }; // unsigned 4-bit literal constant
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 62, format := RL
    CODING { 0b111110 0b00 a m }
    SYNTAX { "LSU" ~ " a , m }
    BEHAVIOR {
        // Subtract literal m from R[a] and store result into R[a]
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { op1 = R1[a]; } // GPR stack 1
        else { op1 = R0[a]; } // GPR stack 0
        op2 = m;
        alu_mode = ALU_SUB;
    }
    ACTIVATION { alu, writeback, set_SR1 }
}
DOCUMENTATION("LSU") { This instruction subtracts the 4-bit literal
contained in bits 3-0 (the m-designator) of the instruction from
the contents in R[a], stores the difference in R[a], and sets the

```

```

        condition code in accordance with the resulting quantity in R[a].
        The overflow and carry designators are set in accordance with the
        results of the operation. }
    }

// Add literal
OPERATION LA IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP m = { imm4 }; // unsigned 4-bit literal constant
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 62, format := RL
    CODING { 0b111110 0b10 a m }
    SYNTAX { "LA" ~" " a "," m }
    BEHAVIOR {
        // Add literal m to R[a] and store result into R[a]
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { op1 = R1[a]; } // GPR stack 1
        else { op1 = R0[a]; } // GPR stack 0
        op2 = m;
        alu_mode = ALU_ADD;
    }
    ACTIVATION { alu, writeback, set_SR1 }
    DOCUMENTATION("LA") { This instruction adds the 4-bit literal
        contained in bits 3-0 (the m-designator) of the instruction to the
        contents of R[a], stores the sum into R[a], and sets the condition
        code in accordance with the resulting quantity in R[a]. The
        overflow and carry designators are set in accordance with the
        results of the operation. }
}

//One's Compliment (Register)
OPERATION OCR IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 2, format := RR
    CODING { 0b000010 0b00 a 0b0110 }
    SYNTAX { "OCR" ~" " a }
    BEHAVIOR {
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { op1 = R1[a]; } // GPR stack 1
        else { op1 = R0[a]; } // GPR stack 0
        op2 = 0xffff;
        alu_mode = ALU_XOR;
        clr_OD = true;
        clr_CD = true;
    }
    ACTIVATION { alu, writeback, set_SR1 }
    DOCUMENTATION("OCR") { This instruction performs ones compliment of
        the contents of R[a] (equivalent to logical XOR with all ones),
        stores the result into R[a], and sets the condition code in
        accordance with the resulting quantity in R[a]. The overflow and
        carry designators are cleared. }
}

```

```

}

//Two's compliment (Register)
OPERATION TCR IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        INSTANCE alu, writeback;
    }
    //opcode := 2, format := RR
    CODING { 0b000010 0b00 a 0b0100 }
    SYNTAX { "TCR" ~" " a }
    BEHAVIOR {
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { op2 = R1[a]; } // GPR stack 1
        else { op2 = R0[a]; } // GPR stack 0
        op1 = 0;
        alu_mode = ALU_SUB;
    }
    ACTIVATION { alu, writeback }
    DOCUMENTATION("TCR") { This instruction performs the twos compliment
        of the contents of R[a] ( equivalent to subtraction from zero),
        stores the result into R[a], and sets the condition code in
        accordance with the resulting quantity in R[a]. The overflow
        designator is set when the maximum negative number is complemented.
        The carry designator is set in accordance with the results of the
        operation. }
}

// Make Positive (Register)
OPERATION PR IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        INSTANCE alu, writeback;
    }
    //opcode := 2, format := RR
    CODING { 0b000010 0b00 a 0b0000 }
    SYNTAX { "PR" ~" " a }
    BEHAVIOR {
        int16 tmp;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { tmp = R1[a]; } // GPR stack 1
        else { tmp = R0[a]; } // GPR stack 0
        op1 = 0;
        op2 = tmp;
        if(tmp < 0) { alu_mode = ALU_SUB; }
        else {
            alu_mode = ALU_ADD;
            clr_OD = true
            clr_CD = true;
        }
    }
    ACTIVATION { alu, writeback }
    DOCUMENTATION("PR") { If the contents of R[a] are less than zero,
        this instruction performs the twos compliment of the contents of
        R[a] (equivalent to subtraction from zero), stores the result into
        R[a], and sets the condition code in accordance with the resulting

```

```

quantity in R[a]. The overflow designator is set when the maximum
negative number is complemented. The carry designator is set in
accordance with the results of the operation. If the contents of
R[a] are equal to or greater than zero, the contents of R[a] are
unchanged and the condition code is set in accordance with the
resulting quantity in R[a]. The overflow and carry designators are
cleared. }
}

/***************** Arithmetic and Logic Unit (ALU) *****/
OPERATION alu IN pipe.ONE {
    BEHAVIOR {
        switch(alu_mode) {
            case ALU_ADD:
                dst = (uint16)op1 + (uint16)op2;
                break;
            case ALU_SUB:
                dst = (uint16)op1 - (uint16)op2;
                break;
            case ALU_AND:
                dst = (uint16)op1 & (uint16)op2;
                break;
            case ALU_OR:
                dst = (uint16)op1 | (uint16)op2;
                break;
            case ALU_XOR:
                dst = (uint16)op1 ^ (uint16)op2;
                break;
            case ALU_SHR:
                dst = (uint16)op1 >> op2;
                break;
            case ALU_SHL:
                dst = op1 << op2;
                break;
            default:
                dst = 0;
                break;
        }
    }
}

/***************** Register Writeback***** */
OPERATION writeback IN pipe.ONE {
    DECLARE { REFERENCE a; }
    BEHAVIOR {
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { R1[a] = dst; } // GPR stack 1
        else { R0[a] = dst; } // GPR stack 0
    }
}

```

logical.lisa

```

*****
*
*          Logical Instructions
*
*****
#include "defines.h"

*****
x32 instructions ****
// And (constant)
OPERATION ANDK IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        GROUP y = { addr };
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 30, format := RK
    CODING { 0b011110 0b10 a m y }
    SYNTAX { "ANDK" ~" " a "," y "," m }
    BEHAVIOR {
        int16 tmp_Y;
        // Bitwise AND R[a] to operand Y and store in R[a]
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R1[m]; } //Direct Addressing with Index
            op1 = R1[a];
        }
        else { // GPR stack 0
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R0[m]; } //Direct Addressing with Index
            op1 = R0[a];
        }
        op2 = tmp_Y;
        alu_mode = ALU_AND;

        clr_OD = true;
        clr_CD = true;
    }
    ACTIVATION { alu, writeback, set_SR1 }
    DOCUMENTATION("ANDK") { This instruction performs the bit-by-bit
        Logical AND of the contents of R[a] and the operand Y, stores the
        result into R[a] and sets the condition code in accordance with the
        resulting quantity in R[a]. The overflow and carry designators are
        cleared. }
}
// syntax 1 for ANDK instruction (Direct Addressing)
ALIAS OPERATION ANDK_simple IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP y = { addr };
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 30, format := RK
    CODING { 0b011110 0b10 a 0b0[4] y }
    SYNTAX { "ANDK" ~" " a "," y }
    BEHAVIOR {

```

```

        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { op1 = R1[a]; } // GPR stack 1
        else { op1 = R0[a]; } // GPR stack 0
        op2 = y;
        alu_mode = ALU_AND;

        clr_OD = true;
        clr_CD = true;
    }
    ACTIVATION { alu, writeback, set_SR1 }
}

/***************** x16 instructions *****/
// Or (Register)
OPERATION ORR IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 31, format := RR
    CODING { 0b011111 0b00 a m }
    SYNTAX { "ORR" ~" " a "," m }
    BEHAVIOR {
        // bitwise OR R[a] with R[m] and store result into R[a]
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            op1 = R1[a];
            op2 = R1[m];
        }
        else { // GPR stack 0
            op1 = R0[a];
            op2 = R0[m];
        }
        alu_mode = ALU_OR;

        clr_OD = true;
        clr_CD = true;
    }
    ACTIVATION { alu, writeback, set_SR1 }
    DOCUMENTATION("ORR") { This instruction performs the bit-by-bit
        logical OR of the contents of R[a] and the contents of R[m], stores
        the result into R[a], and sets the condition code in accordance
        with the resulting quantity in R[a]. The overflow and carry
        designators are cleared. }
}

```

compare.lisa

```

/*****************
*               *
*       Compare Instructions      *
*               *
*****************/

```

```

#include "defines.h"

***** x32 instructions *****
// Compare Constant (CK)
OPERATION CK IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        GROUP y = { addr };
        // INSTANCE set_SR1;
    }
    //opcode := 24, format := RK
    CODING { 0b011000 0b10 a m y }
    SYNTAX { "CK" ~" " a "," y "," m }
    BEHAVIOR {
        int16 tmp_Y;
        int16 tmp_a;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R1[m]; } //Direct Addressing with Index
            tmp_a = R1[a];
        }
        else { // GPR stack 0
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R0[m]; } //Direct Addressing with Index
            tmp_a = R0[a];
        }

        // Set CC, OD and CD of Status Register 1
        int16 CC; //Condition Code
        if(tmp_a == tmp_Y) { CC = 0x0000; }
        else if(tmp_a > tmp_Y) { CC = 0x0100; }
        else { CC = 0x0300; }

        int16 OD; //Overflow Designator
        int16 CD; //Carry Designator

        // Set SR1
        SR1 = (SR1 & 0xf0ff) | (CC | OD | CD);
    }
    // ACTIVATION { set_SR1 }
    DOCUMENTATION("CK") { This instruction arithmetically compares the
        contents of R[a] to the operand Y and sets the condition code,
        overflow, and carry designators in accordance with the results
        of the operation. }
}
ALIAS OPERATION CK_simple IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP y = { addr };
        // INSTANCE set_SR1;
    }
    //opcode := 24, format := RK
    CODING { 0b011000 0b10 a 0b0[4] y }
}

```

```

SYNTAX { "CK" ~" " a "," y }

BEHAVIOR {
    int16 tmp_a;
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { tmp_a = R1[a]; } // GPR stack 1
    else { tmp_a = R0[a]; } // GPR stack 0
    int16 tmp_Y = y;

    // Set CC, OD and CD of Status Register 1
    int16 CC; //Condition Code
    if(tmp_a==tmp_Y) { CC = 0x0000; }
    else if(tmp_a > tmp_Y) { CC = 0x0100; }
    else { CC = 0x0300; }

    int16 OD; //Overflow Designator
    int16 CD; //Carry Designator

    // Set SR1
    SR1 = (SR1 & 0xf0ff) | (CC | OD | CD);
}
// ACTIVATION { set_SR1 }
}

OPERATION C IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        GROUP y = { addr };
        // INSTANCE set_SR1;
    }
    //opcode := 24, format := RX
    CODING { 0b011000 0b11 a m y }
    SYNTAX { "C" ~" " a "," y "," m }
    BEHAVIOR {
        int16 tmp_Y;
        int16 tmp_a;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R1[m]; } //Direct Addressing with Index
            tmp_a = R1[a];
        }
        else { // GPR stack 0
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R0[m]; } //Direct Addressing with Index
            tmp_a = R0[a];
        }
        // Set CC, OD and CD of Status Register 1
        int16 CC; //Condition Code
        if(tmp_a==tmp_Y) { CC = 0x0000; }
        else if(tmp_a > tmp_Y) { CC = 0x0100; }
        else { CC = 0x0300; }
    }
}

```

```

        int16 OD; //Overflow Designator
        int16 CD; //Carry Designator

        // Set SR1
        SR1 = (SR1 & 0xff) | (CC | OD | CD);
    }
    // ACTIVATION { set_SR1 }
    DOCUMENTATION("C") { This instruction arithmetically compares the
        contents of R[a] to the contents of memory address Y and sets the
        condition code, overflow, and carry designators in accordance with
        the results of the operation. }
}

/********************* x16 instructions *****/
// compare literal
INSTRUCTION LC IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP m = { imm4 }; // unsigned 4-bit literal constant
        // INSTANCE set_SR1;
    }
    //opcode := 63, format := RL
    CODING { 0b111111 0b01 a m }
    SYNTAX { "LC" ~" " a "," m }
    BEHAVIOR {
        int16 tmp_a;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { tmp_a = R1[a]; } // GPR stack 1
        else { tmp_a = R0[a]; } // GPR stack 0

        // Set CC, OD and CD of Status Register 1
        int16 CC; //Condition Code
        if(tmp_a==m) { CC = 0x0000; }
        else if(tmp_a > m) { CC = 0x0100; }
        else { CC = 0x0300; }

        int16 OD; //Overflow Designator
        int16 CD; //Carry Designator

        // Set SR1
        SR1 = (SR1 & 0xff) | (CC | OD | CD);
    }
    // ACTIVATION { set_SR1 }
    DOCUMENTATION("LC") { This instruction arithmetically compares the 4-
        bit literal contained in bits 3-0 (the m-designator) of the
        instruction with the contents of R[a] and sets the condition code,
        overflow and carry designators in accordance with the results of
        the operation. }
}

// Compare bit to zero
OPERATION CBR IN pipe.ONE {
    DECLARE {

```

```

        GROUP a = { reg };
        GROUP m = { imm4 };
        // INSTANCE set_SR1;
    }
    //opcode := 7, format := RR
    CODING { 0b000111 0b00 a m }
    SYNTAX { "CBR" ~" " a "," m }
    BEHAVIOR {
        uint16 tmp;
        // (Ra)m : 0; set CC, 0->OV, 0->C
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { tmp = R1[a] & (0x0001 << m); } // GPR stack 1
        else { tmp = R0[a] & (0x0001 << m); } // GPR stack 0

        // Set CC, OD and CD of Status Register 1
        int16 CC; //Condition Code
        int16 OD; //Overflow Designator
        int16 CD; //Carry Designator

        // Set SR1
        SR1 = (SR1 & 0xf0ff) | (CC | OD | CD);
    }
    // ACTIVATION { set_SR1 }
    DOCUMENTATION("CBR") { This instruction arithmetically compares the
        bit in R[a] specified by the 4-bit literal contained in bits 3-0
        (the m-designator) of the instruction with zero and sets the
        condition code in accordance with the results of the operation. Bit
        8 of Status Register 1 will be set for any bit tested; bit 9 of
        Status Register 1 will be cleared unless bit 15 of R[a] is being
        tested in which case it will be set equal to bit 15 of R[a]. The
        overflow and carry designators are cleared. }
}

```

conditionaljump.lisa

```

*****
*
*          Conditional Jump Instructions
*
*****
#include "defines.h"

*****
// index jump
OPERATION XJ IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        GROUP y = { addr };
        INSTANCE alu, writeback;
    }
    //opcode := 41, format := RK
    CODING { 0b101001 0b10 a m y }
    SYNTAX { "XJ" ~" " a "," y "," m }
    BEHAVIOR {

```

```

int16 tmp_Y;
// If (Ra) != 0, (Ra)-1 -> Ra, Y -> P
bool SD = SR1 >> 14; // Stack Designator (SD)
if(SD) { // GPR stack 1
    if(R1[a]) {
        // R1[a] = R1[a] - 1;
        // use the alu subtractor instead of creating a new one
        op1 = R1[a];
        op2 = 1;
        alu_mode = ALU_SUB;
        if(m==0) { tmp_Y = y; }
        else { tmp_Y = y + R1[m]; }
        bpc = tmp_Y;
        bset = true;
    }
}
else { // GPR stack 0
    if(R0[a]) {
        // R0[a] = R0[a] - 1;
        // use the alu subtractor instead of creating a new one
        op1 = R0[a];
        op2 = 1;
        alu_mode = ALU_SUB;
        if(m==0) { tmp_Y = y; }
        else { tmp_Y = y + R0[m]; }
        bpc = tmp_Y;
        bset = true;
    }
}
}
ACTIVATION { alu, writeback }
DOCUMENTATION("XJ") { If the contents of R[a] are not equal to zero,
    this instruction subtracts one from the contents of R[a] and jumps
    to the instruction located at the address specified by the Y
    operand. If the contents of R[a] are equal to zero, the next
    instruction is executed. The condition code, overflow, and carry
    designators are unchanged. }
}
ALIAS OPERATION XJ_simple IN pipe.ONE {
DECLARE {
    GROUP a = { reg };
    GROUP y = { addr };
}
//opcode := 41, format := RK
CODING { 0b101001 0b10 a 0b0[4] y }
SYNTAX { "XJ" ~" " a "," y }
BEHAVIOR {
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { // GPR stack 1
        if(R1[a]) {
            // R1[a] = R1[a] - 1;
            // use the alu subtractor instead of creating a new one
            op1 = R1[a];
            op2 = 1;
            alu_mode = ALU_SUB;
            bpc = y;
            bset = true;
        }
    }
}
}

```

```

        }
    }
    else { // GPR stack 0
        if(R0[a]) {
            // R0[a] = R0[a] - 1;
            // use the alu subtractor instead of creating a new one
            op1 = R0[a];
            op2 = 1;
            alu_mode = ALU_SUB;
            bpc = y;
            bset = true;
        }
    }
}

/***************************************** x16 instructions *****/
//Local Jump Equal
OPERATION LJE IN pipe.ONE {
    DECLARE {
        GROUP d = { imm8 };
    }
    //opcode := 44, format := RI-1
    CODING { 0b101100 0b01 d }
    SYNTAX { "LJE" ~" " d }
    BEHAVIOR {
        bool bit8 = SR1 >> 8;
        if(!bit8) { // If bit 8 of Status Register 1 contains a zero then

            // jump
            bpc = SIGN_EXTEND_8(d);
            bset = true;
        }
    }
    DOCUMENTATION("LJE") { If bit 8 of Status Register 1 contains zero
        (condition code indicates equal comparison or zero arithmetic
        result) This instruction jumps to the instruction located
        at the address specified by the 8 bit sign extended displacement
        (the d-designator) from this jump instruction. If bit 8 of Status
        Register 1 contains one next instruction is executed. The condition
        code, overflow and carry designators are unchanged. }
}

//Local Jump Not Equal
OPERATION LJNE IN pipe.ONE {
    DECLARE {
        GROUP d = { imm8 };
    }
    //opcode := 45, format := RI-1
    CODING { 0b101101 0b01 d }
    SYNTAX { "LJNE" ~" " d }
    BEHAVIOR {
        bool bit8 = SR1 >> 8;
        if(bit8) { // If bit 8 of Status Register 1 contains a one then
jump */
            bpc = SIGN_EXTEND_8(d);
        }
    }
}

```

```

        bset = true;
    }
}

DOCUMENTATION("LJNE") { If bit 8 of Status Register 1 contains one
    (condition code indicates greater than comparison, less than
    comparison, or non-zero arithmetic result) this instruction
    jumps to the instruction located at the address specified by the 8
    bit sign extended displacement (the d-designator) from this jump
    instruction. If bit 8 of Status Register 1 contains zero,
    the next instruction is executed. The condition code, overflow and
    carry designators are unchanged. }
}

// Local Jump Less
OPERATION LJLS IN pipe.ONE {
DECLARE {
    GROUP d = { imm8 };
}
//opcode := 47, format := RI-1
CODING { 0b101111 0b01 d }
SYNTAX { "LJLS" ~" " d }
BEHAVIOR {
    bool bit9 = SR1 >> 9;
    if(bit9) { // If bit 9 of Status Register 1 contains a one then
jump */
        bpc = SIGN_EXTEND_8(d);
        bset = true;
    }
}
DOCUMENTATION("LJLS") { If bit 9 of Status Register 1 contains
    one, (condition code indicates less than comparison of non-zero
    negative arithmetic result) this instruction jumps to the
    instruction located at the address specified by the 8-bit sign
    extended displacement ( the d-designator) from this jump
    instruction. If bit 9 of Status Register 1 contains zero, the next
    instruction is executed. The condition code, overflow, and carry
    designators are unchanged. }
}

// Local jump Greater or Equal
OPERATION LJGE IN pipe.ONE {
DECLARE {
    GROUP d = { imm8 };
}
//opcode := 46, format := RI-1
CODING { 0b101110 0b01 d }
SYNTAX { "LJGE" ~" " d }
BEHAVIOR {
    // If SR1(9) is 0, (P) + d -> P
    bool bit9 = SR1 >> 9;
    if(!bit9) { // If bit 9 of Status Register 1 contains a zero then
jump
        bpc = SIGN_EXTEND_8(d);
        bset = true;
    }
}

```

```

DOCUMENTATION("LJGE") { If bit 9 of Status Register 1 contains zero
    (condition code indicates equal comparison, greater than
    comparison, zero arithmetic, or non-zero and positive arithmetic
    result), this instruction jumps to the instruction located at the
    address specified by the 8-bit sign-extended displacement (the d-
    designator) from this jump instruction. If bit 9 of Status Register
    1 contains one, the next instruction is executed. The condition
    code, overflow, and carry designators are unchanged. }
}

```

unconditionaljump.lisa

```

/*********************************************
*
*          Unconditional Jump Instructions
*
*****************************************/
#include "defines.h"

/****************************************** x32 instructions *****/
// Jump, Link Register
OPERATION JLR_RK IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        GROUP y = { addr };
        INSTANCE alu, writeback;
    }
    //opcode := 42, format := RK
    CODING { 0b101010 0b10 a m y }
    SYNTAX { "JLR" ~" " a "," y "," m }
    BEHAVIOR {
        int16 tmp_Y;
        // P + 2 -> R[a]; bpc <- Y
        op1 = P;
        op2 = 2;
        alu_mode = ALU_ADD;

        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R1[m]; } //Direct Addressing with Index
        }
        else { // GPR stack 0
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R0[m]; } //Direct Addressing with Index
        }
        bpc = tmp_Y;
        bset = true;
    }
    ACTIVATION { alu, writeback }
    DOCUMENTATION("JLR") { This instruction adds two to the contents of
        the P-Register, stores the results into R[a], and jumps to the
        instruction located at the address specified by the operand Y. The
    }
}

```

```

        condition code, overflow, and carry designators are unchanged. }

}

//syntax 2 for JLR_RX instruction (Direct Addressing)
ALIAS OPERATION JLR_RK_simple IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP y = { addr };
        INSTANCE alu, writeback;
    }
    //opcode := 42, format := RK
    CODING { 0b101010 0b10 a 0b0[4] y }
    SYNTAX { "JLR" ~" " a "," y }
    BEHAVIOR {
        op1 = P;
        op2 = 2;
        alu_mode = ALU_ADD;

        bpc = y;
        bset = true;
    }
    ACTIVATION { alu, writeback }
}

OPERATION JLR_RX IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP m = { imm4 };
        GROUP y = { addr };
        INSTANCE alu, writeback;
    }
    //opcode := 42, format := RX
    CODING { 0b101010 0b11 a m y }
    SYNTAX { "JLR" ~" " a ",",*" y "," m }
    BEHAVIOR {
        int16 tmp_Y;
        op1 = P;
        op2 = 2;
        alu_mode = ALU_ADD;

        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R1[m]; } //Direct Addressing with Index
        }
        else { // GPR stack 0
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R0[m]; } //Direct Addressing with Index
        }
        bpc = data_mem[tmp_Y];
        bset = true;
    }
    ACTIVATION { alu, writeback }
    DOCUMENTATION("JLR*") { This instruction adds two to the contents of
        the P-register, stores the result into R[a], and jumps to the
        instruction located at the address specified by the contents of
    }
}

```

```

        memory address Y. The condition code, overflow, and carry
        designators are unchanged. }
    }
//syntax 2 for JLR_RX instruction (Direct Addressing)
ALIAS OPERATION JLR_RX_simple IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP y = { addr };
        INSTANCE alu, writeback;
    }
//opcode := 42, format := RX
CODING { 0b101010 0b11 a 0b0[4] y }
SYNTAX { "JLR" ~" " a ",*" y }
BEHAVIOR {
    op1 = P;
    op2 = 2;
    alu_mode = ALU_ADD;

    bpc = data_mem[y];
    bset = true;
}
ACTIVATION { alu, writeback }
}

/******************* x16 instructions *****/
// Local Jump
OPERATION LJ IN pipe.ONE {
    DECLARE {
        GROUP d = { imm8 };
    }
//opcode := 40, format := RI-1
CODING { 0b101000 0b01 d }
SYNTAX { "LJ" ~" " d }
BEHAVIOR {
    // (P) + d -> P
    bpc = SIGN_EXTEND_8(d) + P;
    bset = true;
}
DOCUMENTATION("LJ") { This instruction jumps to the instruction
located at the address specified by the sign-extended 8-bit
displacement(the d-designator) from this jump instruction. The
condition code, overflow, and carry designators are unchanged. }
}

//Load P register
OPERATION LPR IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
    }
//opcode := 3, format := RR
CODING { 0b000011 0b00 a 0b0100 }
SYNTAX { "LPR" ~" " a }
BEHAVIOR {
    // (Ra) -> P
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { bpc = R1[a]; } // GPR stack 1
}

```

```

        else { bpc = R0[a]; } // GPR stack 0
        bset = true;
    }
DOCUMENTATION("LPR") { This instruction jumps to the instruction at
    the address specified by the contents of R[a]. The condition code,
    overflow, and carry designators are unchanged. }
}

```

load.lisa

```

/*********************************************
*
*          Load Instructions
*
*****************************************/
#include "defines.h"

/***************************************** x32 instructions *****/
// Load Constant (LK)
OPERATION LK IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        GROUP y = { addr };
        INSTANCE set_SR1;
    }
    //opcode := 1 format := RK (Register-Constant(Immediate Operand))
    CODING { 0b000001 0b10 a m y }
    SYNTAX { "LK" ~" " a "," y "," m }
    BEHAVIOR {
        int16 tmp_Y;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R1[m]; } //Direct Addressing with Index
            R1[a] = tmp_Y;
            dst = tmp_Y;
        }
        else { // GPR stack 0
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R0[m]; } //Direct Addressing with Index
            R0[a] = tmp_Y;
            dst = tmp_Y;
        }

        clr_OD = true;
        clr_CD = true;
    }
    ACTIVATION { set_SR1 }
    DOCUMENTATION("LK") { This instruction loads the operand Y
        into R[a] and sets the condition code in accordance with the
        resulting quantity in R[a]. The overflow and carry designators
        are cleared }
}

```

```

// syntax 2 for LK instruction (Direct Addressing)
ALIAS OPERATION LK_simple IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP y = { addr };
        INSTANCE set_SR1;
    }
    CODING { 0b000001 0b10 a 0b0[4] y }
    SYNTAX { "LK" ~" " a "," y }
    BEHAVIOR {
        int16 tmp = y;
        dst = tmp;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { R1[a] = tmp; } // GPR stack 1
        else { R0[a] = tmp; } // GPR stack 0

        clr_OD = true;
        clr_CD = true;
    }
    ACTIVATION { set_SR1 }
}

// Load Index (L)
OPERATION L IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        GROUP y = { addr };
        INSTANCE set_SR1;
    }
    //opcode := 1, format := RX (Register-Memory with or without
    indexing)
    CODING { 0b000001 0b11 a m y }
    SYNTAX { "L" ~" " a "," y "," m }
    BEHAVIOR {
        int16 tmp_Y;
        int16 tmp;
        int8 tmp1;
        int8 tmp2;

        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R1[m]; } //Direct Addressing with Index
        #ifdef LT_PROCESSOR_GENERATOR
            tmp = data_mem[tmp_Y];
            R1[a] = tmp;
            dst = tmp;
        #else
            tmp1 = data_mem[tmp_Y];
            tmp2 = data_mem[tmp_Y+1];
            R1[a] = dst = ((tmp1 << 8) | tmp2);
        #endif
        }
        else { // GPR stack 0
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R0[m]; } //Direct Addressing with Index
        }
    }
}

```

```

#ifndef LT_PROCESSOR_GENERATOR
    tmp = data_mem[tmp_Y];
    R0[a] = tmp;
    dst = tmp;
#else
    tmp1 = data_mem[tmp_Y];
    tmp2 = data_mem[tmp_Y+1];
    R0[a] = dst = ((tmp1 << 8) | tmp2);
#endif
}

clr_OD = true;
clr_CD = true;
}
ACTIVATION { set_SR1 }
DOCUMENTATION("L") { This instruction loads the contents of memory
address Y into R[a] and sets the condition code in accordance with
the resulting quantity in R[a]. The overflow and carry designators
are cleared }
}
// syntax 2 for L instruction (Direct Addressing)
ALIAS OPERATION L_simple IN pipe.ONE {
DECLARE {
    GROUP a = { reg };
    GROUP y = { addr };
    INSTANCE set_SR1;
}
//opcode := 1, format := RX
CODING { 0b000001 0b11 a 0b0[4] y }
SYNTAX { "L" ~ " a , y }
BEHAVIOR {
    int16 tmp;
    uint8 tmp1;
    uint8 tmp2;
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { // GPR stack 1
#endif
        tmp = data_mem[y];
        R1[a] = tmp;
        dst = tmp;
#else
        tmp1 = data_mem[y];
        tmp2 = data_mem[y+1];
        R1[a] = dst = ((tmp1 << 8) | tmp2);
#endif
    }
    else { // GPR stack 0
#endif
        tmp = data_mem[y];
        R0[a] = tmp;
        dst = tmp;
#else
        tmp1 = data_mem[y];
        tmp2 = data_mem[y+1];
        R0[a] = dst = ((tmp1 << 8) | tmp2);
#endif
    }
}

```

```

        clr_OD = true;
        clr_CD = true;
    }
    ACTIVATION { set_SR1 }
}

//Load Double
OPERATION LD IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        GROUP y = { addr };
    }
    //opcode := 2, format := RX
    CODING { 0b000010 0b11 a m y }
    SYNTAX { "LD" ~" " a "," y "," m }
    BEHAVIOR {
        int32 tmp;
        int16 tmp_Y;
        uint8 tmp1;
        uint8 tmp2;
        uint8 tmp3;
        uint8 tmp4;

        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R1[m]; } //Direct Addressing with Index
#ifndef LT_PROCESSOR_GENERATOR
            R1[a] = data_mem[tmp_Y];
            R1[a+1] = data_mem[tmp_Y+1];
#else
            tmp1 = data_mem[tmp_Y];
            tmp2 = data_mem[tmp_Y+1];
            R1[a] = ((tmp1 << 8) | tmp2);

            tmp3 = data_mem[tmp_Y+2];
            tmp4 = data_mem[tmp_Y+3];
            R1[a+1] = ((tmp3 << 8) | tmp4);
#endif
        }
        else { // GPR stack 0
            if(m==0) { tmp_Y = y; } //Direct Addressing
            else { tmp_Y = y + R0[m]; } //Direct Addressing with Index
#ifndef LT_PROCESSOR_GENERATOR
            R0[a] = data_mem[tmp_Y];
            R0[a+1] = data_mem[tmp_Y+1];
#else
            tmp1 = data_mem[tmp_Y];
            tmp2 = data_mem[tmp_Y+1];
            R0[a] = ((tmp1 << 8) | tmp2);

            tmp3 = data_mem[tmp_Y+2];
            tmp4 = data_mem[tmp_Y+3];
            R0[a+1] = ((tmp3 << 8) | tmp4);
#endif
        }
    }
}

```

```

}

// Set CC, OD and CD of Status Register 1
// set dst to be sent to OPERATION set_SR1
int32 dst0 = (tmp1 << 24) | (tmp2 << 16) | (tmp3 << 8) | tmp4;

int16 CC; //Condition Code
if((int32)dst0==0) { CC = 0x0000; }
else if((int32)dst0 > 0) { CC = 0x0100; }
else { CC = 0x0300; }

int16 OD = 0x0000; //Overflow Designator
int16 CD = 0x0000; //Carry Designator

// Set SR1
SR1 = (SR1 & 0xf0ff) | (CC | OD | CD);
}

DOCUMENTATION("LD") { This instruction loads the double length
contents of memory addresses Y,Y+1 into R[a],R[a+1] and sets the
condition code in accordance with the resulting double length
quantity in R[a],R[a+1]. The overflow and carry designators are
cleared. R[a] must be an even numbered register and Y must be
an even numbered address. }

//syntax 2 for LD instruction (Direct Addressing)
ALIAS OPERATION LD_simple IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP y = { addr };
    }
    //opcode := 2, format := RX
    CODING { 0b000010 0b11 a 0b0[4] y }
    SYNTAX { "LD" ~" " a "," y }
    BEHAVIOR {
        int32 tmp;
        uint8 tmp1;
        uint8 tmp2;
        uint8 tmp3;
        uint8 tmp4;

        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
#ifndef LT_PROCESSOR_GENERATOR
            R1[a] = data_mem[y];
            R1[a+1] = data_mem[y+1];
#else
            tmp1 = data_mem[y];
            tmp2 = data_mem[y+1];
            tmp3 = data_mem[y+2];
            tmp4 = data_mem[y+3];

            R1[a] = ((tmp1 << 8) | tmp2);
            R1[a+1] = ((tmp3 << 8) | tmp4);
#endif
        }
    }
}

```

```

        }
    else { // GPR stack 0
#endif LT_PROCESSOR_GENERATOR
    R0[a] = data_mem[y];
    R0[a+1] = data_mem[y+1];
#else
    tmp1 = data_mem[y];
    tmp2 = data_mem[y+1];
    tmp3 = data_mem[y+2];
    tmp4 = data_mem[y+3];

    R0[a] = ((tmp1 << 8) | tmp2);
    R0[a+1] = ((tmp3 << 8) | tmp4);
#endif
}

// Set CC, OD and CD of Status Register 1
// set dst to be sent to OPERATION set_SR1
int32 dst0 = (tmp1 << 24) | (tmp2 << 16) | (tmp3 << 8) | tmp4;

int16 CC; //Condition Code
if((int32)dst0==0) { CC = 0x0000; }
else if((int32)dst0 > 0) { CC = 0x0100; }
else { CC = 0x0300; }

int16 OD; //Overflow Designator
int16 CD; //Carry Designator

// Set SR1
SR1 = (SR1 & 0xf0ff) | (CC | OD | CD);
}

}

/***** x16 instructions *****/
//Load indirect
OPERATION LI IN pipe.ONE {
DECLARE {
    GROUP a,m = { reg };
    INSTANCE set_SR1;
}
//opcode := 1, format := RI-2
CODING { 0b000001 0b01 a m }
SYNTAX { "LI" ~ " a , m" }
BEHAVIOR {
    int16 tmp;
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { // GPR stack 1
#endif LT_PROCESSOR_GENERATOR
        tmp = data_mem[R1[m]];
        R1[a] = tmp;
        dst = tmp;
#else
        uint8 tmp1 = data_mem[R1[m]];
        uint8 tmp2 = data_mem[R1[m]+1];
        R1[a] = dst = ((tmp1 << 8) | tmp2);
#endif
}
}

```

```

        }
    else { // GPR stack 0
#ifndef LT_PROCESSOR_GENERATOR
    tmp = data_mem[R0[m]];
    R0[a] = tmp;
    dst = tmp;
#else
    uint8 tmp1 = data_mem[R0[m]];
    uint8 tmp2 = data_mem[R0[m]+1];
    R0[a] = dst = ((tmp1 << 8) | tmp2);
#endif
    }
}

clr_OD = true;
clr_CD = true;
}
ACTIVATION { set_SR1 }
DOCUMENTATION("LI") { This instruction loads the contents of memory
address Y* into R[a] and sets the condition code in accordance with
the resulting quantity in R[a]. The overflow and carry designators
are cleared. }
}

// Load Register (LR)
OPERATION LR IN pipe.ONE {
DECLARE {
    GROUP a,m = { reg };
    INSTANCE set_SR1;
}
//opcode := 1 format := RR (Register-Register)
CODING { 0b000001 0b00 a m }
SYNTAX { "LR" ~ " a , m" }
BEHAVIOR {
    int16 tmp;
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { // GPR stack 1
        tmp = R1[m];
        R1[a] = tmp;
        dst = tmp;
    }
    else { // GPR stack 0
        tmp = R0[m];
        R0[a] = tmp;
        dst = tmp;
    }
}

clr_OD = true;
clr_CD = true;
}
ACTIVATION { set_SR1 }
DOCUMENTATION("LR") { This instruction loads the contents of
R[m] into R[a] and sets the condition code in accordance
with the resulting quantity in R[a]. The overflow and
carry designators are cleared. }
}

```

```

// load literal (LL)
OPERATION LL IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP m = { imm4 }; // unsigned 4-bit literal constant
        INSTANCE set_SR1;
    }
    //opcode := 63, format := RL
    CODING { 0b111111 0b00 a m }
    SYNTAX { "LL" ~ " a , m" }
    BEHAVIOR {
        uint16 tmp;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            tmp = 0x000f & m;
            R1[a] = tmp;
            dst = tmp;
        }
        else { // GPR stack 0
            tmp = 0x000f & m;
            R0[a] = tmp;
            dst = tmp;
        }
        clr_OD = true;
        clr_CD = true;
    }
    ACTIVATION { set_SR1 }
    DOCUMENTATION("LL") { This instruction loads the 4-bit literal
        Contained in bits 3-0 (the m-designator) of the instruction into
        bits 3-0 of R[a], clears bits 15-4 of R[a], and sets the condition
        code in accordance with the resulting quantity in R[a]. The
        overflow and carry designators are cleared. }
}

// Set Bit
OPERATION SBR IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP m = { imm4 };
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 5, format := RR
    CODING { 0b000101 0b00 a m }
    SYNTAX { "SBR" ~ " a , m" }
    BEHAVIOR {
        uint16 tmp;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        op2 = (0x0001 << (uint16)m);
        if(SD) { op1 = R1[a]; } // GPR stack 1
        else { op1 = R0[a]; } // GPR stack 0
        alu_mode = ALU_OR;
        clr_OD = true;
        clr_CD = true;
    }
    ACTIVATION { alu, writeback, set_SR1 }
}

```

```

DOCUMENTATION("SBR") { This instruction sets the bit in R[a]
specified by the 4-bit literal contained in bits 3-0 (the m-
designator) of the instruction leaving the other bits in R[a]
unchanged and sets the condition code in accordance with the
resulting quantity in R[a]. The overflow and carry designators are
cleared. }
}

// Zero Bit
OPERATION ZBR IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP m = { imm4 };
        INSTANCE alu, writeback, set_SR1;
    }
    //opcode := 6, format := RR
    CODING { 0b000110 0b00 a m }
    SYNTAX { "ZBR" ~" " a "," m }
    BEHAVIOR {
        uint16 temp;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        op2 = ~(0x0001 << (uint16)m);
        if(SD) { op1 = R1[a]; } // GPR stack 1
        else { op1 = R0[a]; } // GPR stack 0
        alu_mode = ALU_AND;
        clr_OD = true;
        clr_CD = true;
    }
    ACTIVATION { alu, writeback, set_SR1 }
    DOCUMENTATION("ZBR") { This instruction clears the bit in R[a]
specified by the 4-bit literal contained in bits 3-0 (the m-
designator) of the instruction leaving the other bits in R[a]
unchanged and sets the condition code in accordance with the
resulting quantity in R[a]. The overflow and carry designators are
cleared. }
}

// Load Status Register 1
OPERATION LSOR IN pipe.ONE {
    DECLARE { GROUP a = { reg }; }
    //opcode := 3, format := RR
    CODING { 0b000011 0b00 a 0b0101 }
    SYNTAX { "LSOR" ~" " a }
    BEHAVIOR {
        bool SD = SR1 >> 14;
        if(SD) { SR1 = R1[a]; } // GPR stack 1
        else { SR1 = R0[a]; } // GPR stack 0
    }
    DOCUMENTATION("LSOR") { This instruction loads the contents of R[a]
into Status Register 1 }
}

```

store.lisa

```
*****  
*  
*          Store Instructions  
*  
*****  
#include "defines.h"  
  
***** x32 instructions *****  
// Store Index (S)  
OPERATION S IN pipe.ONE {  
    DECLARE {  
        GROUP a,m = { reg };  
        GROUP y = { addr };  
    }  
    //opcode := 11, format := RX (Register-Memory with or without  
    // indexing)  
    CODING { 0b001011 0b11 a m y }  
    SYNTAX { "S" ~" " a "," y "," m }  
    BEHAVIOR {  
        int16 tmp_Y;  
        bool SD = SR1 >> 14; // Stack Designator (SD)  
        if(SD) { // GPR stack 1  
            if(m==0) { tmp_Y = y; } //Direct Addressing  
            else { tmp_Y = y + R1[m]; } //Direct Addressing with Index  
#ifdef LT_PROCESSOR_GENERATOR  
            data_mem[tmp_Y] = R1[a];  
#else  
            data_mem[tmp_Y] = R1[a] >> 8;  
            data_mem[tmp_Y+1] = R1[a];  
#endif  
        }  
        else { // GPR stack 0  
            if(m==0) { tmp_Y = y; } //Direct Addressing  
            else { tmp_Y = y + R0[m]; } //Direct Addressing with Index  
#ifdef LT_PROCESSOR_GENERATOR  
            data_mem[tmp_Y] = R0[a];  
#else  
            data_mem[tmp_Y] = R0[a] >> 8;  
            data_mem[tmp_Y+1] = R0[a];  
#endif  
        }  
        UB = 1;  
        LB = 1;  
    }  
    DOCUMENTATION("S") { This instruction stores the contents of  
        R[a] into memory address Y. The condition code, overflow, and  
        carry designators are unchanged. }  
}  
ALIAS OPERATION S_simple IN pipe.ONE {  
    DECLARE {  
        GROUP a = { reg };  
        GROUP y = { addr };  
    }  
    //opcode := 11, format := RX
```

```

CODING { 0b001011 0b11 a 0b0[4] y }
SYNTAX { "S" ~" " a "," y }
BEHAVIOR {
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { // GPR stack 1
#ifndef LT_PROCESSOR_GENERATOR
        data_mem[y] = R1[a];
#else
        data_mem[y] = R1[a] >> 8;
        data_mem[y+1] = R1[a];
#endif
    }
    else { // GPR stack 0
#ifndef LT_PROCESSOR_GENERATOR
        data_mem[y] = R0[a];
#else
        data_mem[y] = R0[a] >> 8;
        data_mem[y+1] = R0[a];
#endif
    }
    UB = 1;
    LB = 1;
}
}

// Store Double (SD)
OPERATION SD IN pipe.ONE {
DECLARE {
    GROUP a,m = { reg };
    GROUP y = { addr };
}
//opcode := 12, format := RX (Register-Memory with or without
indexing)
CODING { 0b001100 0b11 a m y }
SYNTAX { "SD" ~" " a "," y "," m }
BEHAVIOR {
    int16 tmp_Y;
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { // GPR stack 1
        if(m==0) { tmp_Y = y; } //Direct Addressing
        else { tmp_Y = y + R1[m]; } // Direct Addressing with Index
#ifndef LT_PROCESSOR_GENERATOR
        data_mem[tmp_Y] = R1[a];
        data_mem[tmp_Y+1] = R1[a+1];
#else
        data_mem[tmp_Y] = R1[a] >> 8;
        data_mem[tmp_Y+1] = R1[a];
        data_mem[tmp_Y+2] = R1[a+1] >> 8;
        data_mem[tmp_Y+3] = R1[a+1];
#endif
    }
    else { // GPR stack 0
        if(m==0) { tmp_Y = y; } //Direct Addressing
        else { tmp_Y = y + R0[m]; } // Direct Addressing with Index
#ifndef LT_PROCESSOR_GENERATOR

```

```

        data_mem[tmp_Y] = R0[a];
        data_mem[tmp_Y+1] = R0[a+1];
#endif
        data_mem[tmp_Y] = R0[a] >> 8;
        data_mem[tmp_Y+1] = R0[a];
        data_mem[tmp_Y+2] = R0[a+1] >> 8;
        data_mem[tmp_Y+3] = R0[a+1];
#endif
}
UB = 1;
LB = 1;
}
DOCUMENTATION("SD") { This instruction stores the double
length contents of R[a],R[a+1] into memory addresses
Y,Y+1. The condition code, overflow, and carry designators
are unchanged. R[a] must be an even numbered register and
Y must be an even numbered address. }
}
ALIAS OPERATION SD_simple IN pipe.ONE {
DECLARE {
    GROUP a = { reg };
    GROUP y = { addr };
}
//opcode := 12, format := RX
CODING { 0b001100 0b11 a 0b0[4] y }
SYNTAX { "SD" ~" " a "," y }
BEHAVIOR {
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { // GPR stack 1
#ifndef LT_PROCESSOR_GENERATOR
        data_mem[y] = R1[a];
        data_mem[y+1] = R1[a+1];
#else
        data_mem[y] = R1[a] >> 8;
        data_mem[y+1] = R1[a];
        data_mem[y+2] = R1[a+1] >> 8;
        data_mem[y+3] = R1[a+1];
#endif
    }
    else { // GPR stack 0
#ifndef LT_PROCESSOR_GENERATOR
        data_mem[y] = R0[a];
        data_mem[y+1] = R0[a+1];
#else
        data_mem[y] = R0[a] >> 8;
        data_mem[y+1] = R0[a];
        data_mem[y+2] = R0[a+1] >> 8;
        data_mem[y+3] = R0[a+1];
#endif
    }
    UB = 1;
    LB = 1;
}
}

```

```

// store zeros
OPERATION SZ IN pipe.ONE {
    DECLARE {
        GROUP m = { reg };
        GROUP y = { addr };
    }
    //opcode := 17, format := RX
    CODING { 0b010001 0b11 0b0000 m y }
    SYNTAX { "SZ" ~" " y , " m" }
    BEHAVIOR {
        int16 tmp_Y;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            if(m==0) { tmp_Y = y; } // Direct Addressing
            else { tmp_Y = y + R1[m]; } // Direct Addressing with Index
        }
        else { // GPR stack 0
            if(m==0) { tmp_Y = y; } // Direct Addressing
            else { tmp_Y = y + R0[m]; } // Direct Addressing with Index
        }
        #ifdef LT_PROCESSOR_GENERATOR
            data_mem[tmp_Y] = 0;
        #else
            data_mem[tmp_Y] = 0;
            data_mem[tmp_Y+1] = 0;
        #endif
        UB = 1;
        LB = 1;
    }
    DOCUMENTATION("SZ") { This instruction stores all zeros into memory
        address Y. The condition code, overflow, and carry designators
        are unchanged. The a- designator is not used. }
}
ALIAS OPERATION SZ_simple IN pipe.ONE {
    DECLARE {
        GROUP y = { addr };
    }
    //opcode := 17, format := RX
    CODING { 0b010001 0b11 0b0000 0b0[4] y }
    SYNTAX { "SZ" ~" " y }
    BEHAVIOR {
        #ifdef LT_PROCESSOR_GENERATOR
            data_mem[y] = 0;
        #else
            data_mem[y] = 0;
            data_mem[y+1] = 0;
        #endif
        UB = 1;
        LB = 1;
    }
}

//Byte Store
OPERATION BS IN pipe.ONE {
    DECLARE {

```

```

    GROUP a,m = { reg };
    GROUP y = { addr };
}
//opcode := 10, format := RX
CODING { 0b001010 0b11 a m y }
SYNTAX { "BS" ~" " a "," y "," m }
BEHAVIOR {
    int16 tmp_Y;
    // select which word (HW or LW) to write to
    uint8 byte_select = m & 0x01;
    if(byte_select) {
        UB = 0;
        LB = 1;
    }
    else {
        UB = 1;
        LB = 0;
    }

    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { // GPR stack 1
        if(m==0) { tmp_Y = y; } //Direct Addressing
        else { tmp_Y = y + R1[m]; } //Direct Addressing with index
#define LT_PROCESSOR_GENERATOR
        data_mem[tmp_Y] = R1[a] & 0x00ff; // send lower 8 bits unchanged
and zero out upper 8
#else
        if(UB) { data_mem[tmp_Y] = R1[a]; } // it is an 8-bit bus so the
lower byte will be sent
        else { data_mem[tmp_Y+1] = R1[a]; }
#endif
    }
    else { // GPR stack 0
        if(m==0) { tmp_Y = y; } //Direct Addressing
        else { tmp_Y = y + R0[m]; } //Direct Addressing with index
#define LT_PROCESSOR_GENERATOR
        data_mem[tmp_Y] = R0[a] & 0x00ff; // send lower 8 bits unchanged
and zero out upper 8
#else
        if(UB) { data_mem[tmp_Y] = R0[a]; } // it is an 8-bit bus so the
lower byte will be sent
        else { data_mem[tmp_Y+1] = R0[a]; }
#endif
    }
}
DOCUMENTATION("BS") { This instruction stores bits 7-0 of the
Contents of R[a] into the selected byte at memory address Y; the
other byte at memory address Y is unchanged. The condition code,
overflow, and carry designators are unchanged. Unless otherwise
specified for indirect addressing, this instruction uses bit 0 of
R[m] as the byte pointer. if m=0, Y=y and the byte pointer=0. A
byte pointer of 0 selects the upper byte (bits 15-8); a byte
pointer of 1 selects the lower byte(bits 7-0). }
}
ALIAS OPERATION BS_simple IN pipe.ONE {
DECLARE {
    GROUP a = { reg };

```

```

        GROUP y = { addr };
    }
//opcode := 10, format := RX
CODING { 0b001010 0b11 a 0b0[4] y }
SYNTAX { "BS" ~" " a "," y }
BEHAVIOR {
    // select which word (HW or LW) to write to
    UB = 1;
    LB = 0;
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { // GPR stack 1
#define LT_PROCESSOR_GENERATOR
        data_mem[y] = R1[a] & 0x00ff; // send lower 8 bits unchanged and
zero out upper 8
#else
        if(UB) { data_mem[y] = R1[a]; }
        else { data_mem[y+1] = R1[a]; }
#endif
    }
    else { // GPR stack 0
#define LT_PROCESSOR_GENERATOR
        data_mem[y] = R0[a] & 0x00ff; // send lower 8 bits unchanged and
zero out upper 8
#else
        if(UB) { data_mem[y] = R0[a]; }
        else { data_mem[y+1] = R0[a]; }
#endif
    }
}
}

// Store multiple
OPERATION SM IN pipe.ONE {
DECLARE {
    GROUP a,m = { reg };
    GROUP y = { addr };
}
//opcode := 13, format := RX
CODING { 0b001101 0b11 a m y }
SYNTAX { "SM" ~" " a "," y "," m }
BEHAVIOR {
    int16 tmp_Y;
    int i;
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { // GPR stack 1
        if(m==0) { tmp_Y = y; } //Direct Addressing
        else { tmp_Y = y + R1[m]; } //Direct Addressing with Index

        if(m >= a) {
            for(i=a; i <= m; i++) {
#define LT_PROCESSOR_GENERATOR
                data_mem[tmp_Y] = R1[i];
                tmp_Y += 1;
#else
                data_mem[tmp_Y] = R1[i] >> 8;
#endif
            }
        }
    }
}

```

```

        data_mem[tmp_Y+1] = R1[i];
        tmp_Y += 2;
#endif
    }
}
else {
    for(i=a; i >= m; i--) {
#endif LT_PROCESSOR_GENERATOR
        data_mem[tmp_Y] = R1[i];
        tmp_Y += 1;
#else
        data_mem[tmp_Y] = R1[i] >> 8;
        data_mem[tmp_Y+1] = R1[i];
        tmp_Y += 2;
#endif
    }
}
else { // GPR stack 0
    if(m==0) { tmp_Y = y; } //Direct Addressing
    else { tmp_Y = y + R0[m]; } //Direct Addressing with Index

    if(m >= a) {
        for(i=a; i <= m; i++) {
#endif LT_PROCESSOR_GENERATOR
            data_mem[tmp_Y] = R0[i];
            tmp_Y += 1;
#else
            data_mem[tmp_Y] = R0[i] >> 8;
            data_mem[tmp_Y+1] = R0[i];
            tmp_Y += 2;
#endif
    }
}
else {
    for(i=a; i >= m; i--) {
#endif LT_PROCESSOR_GENERATOR
        data_mem[tmp_Y] = R0[i];
        tmp_Y += 1;
#else
        data_mem[tmp_Y] = R0[i] >> 8;
        data_mem[tmp_Y+1] = R0[i];
        tmp_Y += 2;
#endif
    }
}
UB = 1;
LB = 1;
}
DOCUMENTATION("SM") { If m is greater than or equal to a, this
Instruction stores the contents of sequential registers R[a]
through R[m] into sequential memory addresses beginning at Y. If m
is less than a, this instruction stores the contents of sequential
registers R[a], R[a-1] ... R[0] ... R[m] into sequential memory
addresses beginning at Y. The condition code, overflow, and
carry designators are unchanged. In this instruction Y equals y
}

```

```

        (indexing and indirect addressing cannot be used). The beginning
        address may be either odd or even. }
    }

ALIAS OPERATION SM_simple IN pipe.ONE {
    DECLARE {
        GROUP a = { reg };
        GROUP y = { addr };
    }
    //opcode := 13, format := RX
    CODING { 0b001101 0b11 a 0b0[4] y }
    SYNTAX { "SM" ~" " a "," y }
    BEHAVIOR {
        int i;
        bool SD = SR1 >> 14; // Stack Designator (SD)
        if(SD) { // GPR stack 1
            for(i=a; i >= 0; i--) {
#ifndef LT_PROCESSOR_GENERATOR
                data_mem[y] = R1[i];
                y += 1;
#else
                data_mem[y] = R1[i] >> 8;
                data_mem[y+1] = R1[i];
                y += 2;
#endif
        }
        }
        else { // GPR stack 0
            for(i=a; i >= 0; i--) {
#ifndef LT_PROCESSOR_GENERATOR
                data_mem[y] = R0[i];
                y += 1;
#else
                data_mem[y] = R0[i] >> 8;
                data_mem[y+1] = R0[i];
                y += 2;
#endif
        }
        }
        UB = 1;
        LB = 1;
    }
}

//***** x16 instructions *****/
// Store and Index by 1 (SXI)
OPERATION SXI IN pipe.ONE {
    DECLARE {
        GROUP a,m = { reg };
        INSTANCE alu, writeback;
    }
    //opcode := 15, format := RI (Register-Indirect Memory)
    CODING { 0b001111 0b01 a m }
    SYNTAX { "SXI" ~" " a "," m }
    BEHAVIOR {
        uint16 addr;

```

```

int16 tmp_a;

bool SD = SR1 >> 14; // Stack Designator (SD)
if(SD) { // GPR stack 1
    addr = R1[m];
    tmp_a = R1[a];
#ifndef LT_PROCESSOR_GENERATOR
    data_mem[addr] = tmp_a;
#else
    data_mem[addr] = tmp_a >> 8;
    data_mem[addr+1] = tmp_a;
#endif
R1[m] = addr + 1;
if(a==m) {
    op1 = tmp_a;
    op2 = 1;
    alu_mode = ALU_ADD;
}
}
else { // GPR stack 0
    addr = R0[m];
    tmp_a = R0[a];
#ifndef LT_PROCESSOR_GENERATOR
    data_mem[addr] = tmp_a;
#else
    data_mem[addr] = tmp_a >> 8;
    data_mem[addr+1] = tmp_a;
#endif
R0[m] = addr + 1;
if(a==m) {
    op1 = tmp_a;
    op2 = 1;
    alu_mode = ALU_ADD;
}
}
UB = 1;
LB = 1;
}

ACTIVATION { if(a==m) { alu, writeback } }
DOCUMENTATION("SXI") { This instruction stores the contents
of R[a] into memory address Y* (R[m]==address) and increments the
contents of R[m] by one. if a=m, the contents of R[a] will be
incremented by 1 after being stored into memory address Y*. The
condition code, overflow, and carry designators are unchanged }
}

// Store Indirect (SI)
OPERATION SI IN pipe.ONE {
DECLARE {
    GROUP a,m = { reg };
}
//opcode := 11, format := RI (Register-Indirect Memory)
CODING { 0b001011 0b01 a m }
SYNTAX { "SI" ~" " a "," m }
BEHAVIOR {
    bool SD = SR1 >> 14; // Stack Designator (SD)
}
}

```

```

        if(SD) { // GPR stack 1
#ifdef LT_PROCESSOR_GENERATOR
        data_mem[R1[m]] = R1[a];
#else
        data_mem[R1[m]] = R1[a] >> 8;
        data_mem[R1[m]+1] = R1[a];
#endif
}
else { // GPR stack 0
#ifdef LT_PROCESSOR_GENERATOR
        data_mem[R0[m]] = R0[a];
#else
        data_mem[R0[m]] = R0[a] >> 8;
        data_mem[R0[m]+1] = R0[a];
#endif
}
UB = 1;
LB = 1;
}
DOCUMENTATION("SI") { This instruction stores the contents of R[a]
into memory address Y* (R[m]==address). The condition code,
overflow, and carry designators are unchanged. }
}

// Store Status Register
OPERATION SSOR IN pipe.ONE {
DECLARE {
    GROUP a = { reg };
    INSTANCE set_SR1;
}
//opcode := 3, format := RR
CODING { 0b000011 0b00 a 0b0001 }
SYNTAX { "SSOR" ~" " a }
BEHAVIOR {
    uint16 tmp = SR1;
    bool SD = SR1 >> 14; // Stack Designator (SD)
    if(SD) { R1[a] = tmp; } // GPR stack 1
    else { R0[a] = tmp; } // GPR stack 0
    dst = tmp;

    clr_OD = true;
    clr_CD = true;
}
ACTIVATION { set_SR1 }
DOCUMENTATION("SSOR") { This instruction stores the contents of
Status Register 1 into R[a] and sets the condition code in
accordance with the resulting quantity in R[a]. The overflow and
carry designators are cleared. }
}

```

immediate.lisa

```

*
*           Immediate Implementation
*
***** ****
// 4-bit register index
IMMEDIATE reg
{
    DECLARE { LABEL idx; } //Label for register index
    SYNTAX { "R" ~idx } //
    CODING { idx=0bx[4] } // register index is 4 bits
    EXPRESSION { idx } // returns register index
}

// 4-bit immediate value
// Literal format -- 4-bit unsigned integers
IMMEDIATE imm4
{
    DECLARE { LABEL value; }
    CODING { value=0bx[4] }
    SYNTAX { SYMBOL( value=#U4 ) }
    EXPRESSION { value }
}

// 8-bit immmmediate value
// Byte format -- 8-bit unsigned integers
IMMEDIATE imm8
{
    DECLARE { LABEL value; }
    CODING { value=0bx[8] }
    SYNTAX { SYMBOL( value=#S8 ) }
    EXPRESSION { value }
}

// 16-bit immediate value
IMMEDIATE addr
{
    DECLARE { LABEL value; }
    CODING { value=0bx[16] }
    SYNTAX { SYMBOL( value=#U16 ) }
    EXPRESSION { value }
}

```

APPENDIX F

MEMORY_CFG.H

```
*****  
**  
** This confidential and proprietary software may be used only **  
** as authorized by a licensing agreement from CoWare, Inc. **  
** In the event of publication, the following notice is **  
** applicable: **  
**  
** (c) COPYRIGHT 2001-2007 COWARE, INC. **  
** ALL RIGHTS RESERVED **  
**  
** The entire notice above must be reproduced on all authorized **  
** copies. **  
**  
*****/  
  
/* $Id: memory_cfg.h,v 1.3 2006/02/16 11:22:59 zerres Exp $  
 */  
  
/** Memory configuration:  
 This file defines the ranges of the memory  
 subsystem defined in memory.lisa.  
 */  
  
#ifndef MEMORY_CFG_H  
#define MEMORY_CFG_H  
  
#define PMEM_SIZE    0x000010000  
#define PMEM_START   0x00000000  
#define PMEM_END     0x0000ffff  
#define PMEM_PAGE    0  
/* Bytes in word */  
#define PMEM_BIW     1  
  
#define DMEM_SIZE    0x000010000  
#define DMEM_START   0xf0010000  
#define DMEM_END     0xf001ffff  
#define DMEM_PAGE    1  
/* Bytes in word */  
#define DMEM_BIW     1  
  
#endif // MEMORY_CFG_H
```

MEMORY_IF.H

```
/*********************************************
**
** This confidential and proprietary software may be used only      **
** as authorized by a licensing agreement from CoWare, Inc.          **
** In the event of publication, the following notice is applicable:   **
**
**             (c) COPYRIGHT 2001-2007 COWARE, INC.                      **
**                         ALL RIGHTS RESERVED                           **
**
** The entire notice above must be reproduced on all authorized     **
** copies.                                                               **
**
********************************************/
```

```
/* $Id: memory_if.h,v 1.2 2006/02/16 11:22:59 zerres Exp $ */
 */

/***
Memory interface:
This file defines the macros for the memory access
initiated by the core. All memory access is done
via these macros. If the memory sub-system changes
only these access macros have to be adopted.
*/

#ifndef MEMORY_IF_H
#define MEMORY_IF_H

#include "memory_cfg.h"

#define PMEM_LD(dst,addr) dst=prog_mem[addr];

#define DMEM_LD(dst,addr) dst=data_mem[addr];

#define DMEM_ST(src,addr) data_mem[addr]=src;

#endif // MEMORY_IF_H
```

APPENDIX G

defines.h

```
*****  
*  
* CoWare LISATek Tool Suite  
* Copyright (c) 2001-2005 by CoWare, Inc.  
* ALL RIGHTS RESERVED  
*  
* This file contains proprietary and confidential  
* information of CoWare, Inc. and may be used  
* and disclosed only as authorized in a license  
* agreement controlling such use and disclosure  
*  
*****  
#ifndef _DEFINES_H  
#define _DEFINES_H  
  
#define DBGOUT 2  
#define bits ExtractToLong  
#define set_range SetRegion  
  
#define EXTEND_27(s) ((int32)((s & 0x000000ff)))  
  
#define SIGN_EXTEND_16(s) ((int32)((s & 0x00008000) ? (s | 0xffff0000)  
: (s & 0x0000ffff)))  
  
#define SIGN_EXTEND_15(s) ((int32)((s & 0x00010000) ? (s | 0xffffe0000)  
: (s & 0x000fffff)))  
  
#define SIGN_EXTEND_10(s) ((int32)((s & 0x00200000) ? (s | 0xffc00000)  
: (s & 0x00fffff)))  
  
// ALU constants  
#define ALU_ADD 1  
#define ALU_SUB 2  
#define ALU_AND 3  
#define ALU_OR 4  
#define ALU_NOT 5  
#define ALU_NEG 6  
#define ALU_SHR 7  
#define ALU_SHRA 8  
#define ALU_SHL 9  
#define ALU_SHC 10  
  
// BRANCH constants  
#define BRNV 1  
#define BR 2  
#define BRZR 3  
#define BRNZ 4  
#define BRPL 5
```

```
#define BRMI 6

// OPCODE constants
#define LD 1
#define LDR 2
#define ST 3
#define STR 4
#define LA 5
#define LAR 6

#define BRANCH 8
#define BRANCHL 9

#define ADD 12
#define ADDI 13
#define SUB 14
#define NEG 15
#define AND 20
#define ANDI 21
#define OR 22
#define ORI 23
#define NOT 24

#define SHR 26
#define SHRA 27
#define SHL 28
#define SHC 29

#define NOP 0
#define STOP 31

#endif
```

APPENDIX H

Assembly Source Files

Non-pipelined SRC

```
*=====
* FILE:          app.asm
*
* DESCRIPTION: Some ALU immediate and register instructions
*
*=====
/* begin data */
.data
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
UART_RX_Stat: .equ 0xF300
UART_RX_Data: .equ 0xF304
UART_TX_Stat: .equ 0xF110
UART_TX_Data: .equ 0xF114

/* begin program code */
.text
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
;
.global _start
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
;
/* set label for program entry */

_start:
    la r3,0
    la r1,UART_ISR_RX
    addi r1,r1,1
    la r2,UART_ISR_TX
    addi r2,r2,1
    st r1,UART_RX_Stat
    st r2,UART_TX_Stat
    lar r30,_Loop
    lar r29,_Uart
    lar r28,_Wait
    een
    nop
_Loop:
    brnz r29,r3
    br r30
    nop
```

```

_Uart:
    st r0,UART_TX_Data
    nop
_Wait:
    brzr r30,r3
    br r28
    nop
UART_ISR_RX:
    ld r0,UART_RX_Data
    la r3,1
    rfi
UART_ISR_TX:
    la r3,0
    rfi

=====
* FILE:          app.asm
*
* DESCRIPTION: Some ALU immediate and register instructions
*
=====

/* begin data */
.data
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
//cnt: .equ 5
//stp: .equ 1
/* begin program code */
.text
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
;
.global _start
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
;
/* set label for program entry */
_start:
    la r0,1
    la r1,_loop
    la r4,_end
    lar r6,3
    la r3,3
    // la r4,4
    la r2,0(r6)
    la r7,-32
    str r7,0
    ldr r16,-4
    shr r8,r7,3
    shr r9,r7,r6
    shl r10,r9,3
    shl r11,r8,r6
    shra r12,r7,3

```

```

        shra r13,r7,r6
        shc r14,r7,8
        shc r15,r7,r6
        nop
        een
_start:
        nop
        addi r2,r2,-1
        str r2,0
        brlzs r5,r4,r2
        br      r1
_end:
        edi
        nop

/* mark end of assembly program */
.end

=====
* FILE:          app.asm
*
* DESCRIPTION: Some ALU immediate and register instructions
*
=====

// begin data
.data
// must be added to have the instructions
// stored big endian in prog mem
.bend
//cnt: .equ 5
//stp: .equ 1
// begin program code
.text
// must be added to have the instructions
// stored big endian in prog mem
.bend
;
.global _start
// must be added to have the instructions
// stored big endian in prog mem
.bend
;
// set label for program entry
_start:
        la r1,3(r0)
        st r1,0
        la r1,4
        st r1,0(r1)
        la r1,5(r0)
        st r1,4(r0)
        la r1,7(r0)
        st r1,8(r0)
        la r1,11(r0)
        st r1,12(r0)
        la r1,8(r0)
        ld r2,1(r0) // Direct addressing with r0 specified r2 <= 3

```

```

ld r3,0(r1) // Displacement addressing r3 <= 5
ld r4,0(r1) // Register Indirect addressing with c2=0 implied
// r4 <= 7
ld r5,12 // Direct addressing with r0 implied r5 <= 11 (0xB)
ld r6,0(r0) // Both r0 and c2=0 specified just to check r6 <= 1
ld r7,0(r1) // Register Indirect addressing with c2=0 stated
// r7 <= 7
nop
la r7,-32(r0)
// la r7,4294967264
shr r9,r7,r6,0
shr r9,r7,r6
shr r8,r7,r0,3
shr r8,r7,3
shl r10,r8,r0,3
shl r10,r8,3
shl r11,r9,r6,0
shl r11,r9,r6
shra r12,r7,r0,3
shra r12,r7,3
shra r13,r7,r6,0
shra r13,r7,r6
shc r14,r7,r0,8
shc r14,r7,8
shc r15,r7,r6,0
shc r15,r7,r6
// mark end of assembly program
.end

```

Pipelined SRC

```

=====
* FILE: app.asm
*
* DESCRIPTION: Some ALU immediate and register instructions
*
=====
/* begin data */
.data
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
//cnt: .equ 5
//stp: .equ 1
/* begin program code */
.text
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
;
.global _start
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
;

```

```

        /* set label for program entry */
_start:
    lar r0,_loop
    lar r1,_end
    // br r0
    addi r1,r1,1    // r1 <- 1
    nop
    nop
    ld r2,0(r1)    // alu-alu dependency r2 <- 2
    ld r3,0(r2)    // alu-load dependency r3 <- mem[r2] = 3
    sub r4,r3,r1   // load-alu dependency r4 <- 2

    la r5,0(r4)    // alu-ladr dependency r5 <- 2
    brlpl r6,r0,r5 // ladr-brl dependency
    not r5,r5
    nop
    nop
    nop
_loop:
    brnz r1,r2
    br r0
    st r6,0          // brl-store(ra) dependency mem[0] <- 6
    een
    ldr r7,0          // r7 <- 6
    st r1,0(r7)      // load-store(rb) dependency mem[6] <- 1
_end:
    /* mark end of assembly program */
.end

```

```

=====
* FILE:          app.asm
*
* DESCRIPTION: Some ALU immediate and register instructions
*
=====
/* begin data */
.data
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
//cnt: .equ 5
//stp: .equ 1
/* begin program code */
.text
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
;
.global _start
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
;
/* set label for program entry */
_start:
    la r0,_loop

```

```

la r7,-32
ld r2,0
la r3,-5
la r4,4
la r6,3
add r1,r2,r3
st r1,1
nop
add r1,r4,r1
add r5,r1,r1
brmi r0,r5
addi r5,r5,2
st r5,2
br r0
neg r7,r7
nop
_loop:
    shr r8,r7,3
    shr r9,r7,r6
    shl r10,r9,3
    shl r11,r8,r6
    shra r12,r7,3
    shra r13,r7,r6
    shc r14,r7,8
    shc r15,r7,r6

/* mark end of assembly program */
.end

```

VLIW SRC

```

=====
* FILE:          app.asm
*
* DESCRIPTION: Some ALU immediate and register instructions
*
=====
// begin data
.data
// must be added to have the instructions
// stored big endian in prog mem
.bend
//cnt: .equ 5
//stp: .equ 1
// begin program code
.text
// must be added to have the instructions
// stored big endian in prog mem
.bend
;
.global _start
// must be added to have the instructions
// stored big endian in prog mem
.bend
;
```

```

        // set label for program entry
_start:
    addi r1,r1,3 | addi r0,r0,1
    nop | add r2,r1,r0
    la r5,0xff | nop
    st r2,0 | nop
    ld r3,0 | la r4,0xff
    een | shr r6,r4,5

```

Pipelined VLIW SRC

```

=====
* FILE:          app.asm
*
* DESCRIPTION: Some ALU immediate and register instructions
*
=====
/* begin data */
.data
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
UART_RX_Stat: .equ 0xF300
UART_RX_Data: .equ 0xF304
UART_TX_Stat: .equ 0xF110
UART_TX_Data: .equ 0xF114
/* begin program code */
.text
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
;
.global _start
/* must be added to have the instructions
   stored big endian in prog mem */
.bend
;
/* set label for program entry */

_start:
    la r3,0 | la r1,_UART_ISR_RX
    st r1, UART_RX_Stat | la r2,_UART_ISR_TX
    st r2, UART_TX_Stat | lar r30,_Loop
    lar r29,_Uart | lar r28,_Wait
_Loop:
    een | brnz r29,r3
    nop | br r30
_Uart:
    st r0,UART_TX_Data | nop
_Wait:
    nop | nop
    nop | brzr r30,r3
    nop | br r28
_UART_ISR_RX:
    ld r0,UART_RX_Data | la r3,1

```

```

        rfi | nop
_UART_ISR_TX:
        rfi | la r3,0

/* mark end of assembly program */
.end

```

AYK-14

```

=====
* FILE:          app.asm
*
* DESCRIPTION: Some ALU immediate and register instructions
*
=====

// begin data
.data
// must be added to have the instructions
// stored big endian in prog mem

//cnt: .equ 5
//stp: .equ 1
// begin program code
.text
// must be added to have the instructions
// stored big endian in prog mem
;
.global _start
// must be added to have the instructions
// stored big endian in prog mem
;
// set label for program entry
_start:
    IROR R2           // R2 <- 1
//    SXI R2,R2
//    DROR R2           // R2 <- 0
//    SM R1,0,R5
//    SM R8,8,R6
//    CK R2,1           // SR1 <- 0x0300
//    DROR R2           // R2 <- -1
//    LJLS _loop
    LL R2,0x1         // R2 <- 1
    SXI R2,R1         // data_mem[R1] <- R2 = 1
    LL R1,0xe         // R1 <- 0x000e
    LA R2,12          // R2 <- 0x000d

//_loop:
    BS R2,0,R6         // data_mem[0] <- 0x0d
    BS R2,0,R3         // data_mem[1] <- 0x0d
    LD R3,0             // R3 <- 0xd0d
    LSU R1,3            // R1 <- 0x000b
    LALD R1,12          // R1 <- 0xb000, R2 <- 0xd000
    LARD R1,6            // R1 <- 0xfec0, R2 <- 0x0340
    LLRS R1,4            // R1 <- 0x0fec
    LALS R2,4            // R2 <- 0x3400
    LR R3,R2            // R3 <- 0x3400
    DROR R2             // R2 <- 0x33ff

```

```

    IROR R1      // R1 <- 0x0fed
    AR R3,R1    // R3 <- 0x43ed
    ORR R3,R2    // R3 <- 0x73ff
    TCR R3      // R3 <- 0x8c01
    ZBR R3,15    // R3 <- 0x0c01
    OCR R3      // R3 <- 0xf3fe
    SBR R3,10    // R3 <- 0xf7fe
    LI R4,R0    // R4 <- data_mem[R0] = 0xdead
    LR R5,R4    // R5 <- 0xdead
    ANDK R4,0xe0d // R4 <- 0x0e0d
    LK R6,0,R4    // R6 <- 0x0e0d
    ANDK R5,0,R6    // R5 <- 0x0e0d
    LK R7,0xb    // R7 <- 0x000b

    // mark end of assembly program
    .end

```

SRC UART Echo Program

UART with interrupts SRC Assembly Code

```

-----INTERRUPT EXAMPLES
----UART with INTERRUPTS
with address select
    dataout <=
        x"28c00000" when x"0000", -- _start:    la r3,0
        x"28400041" when x"0004", --           la r1,UART_ISR_RX + 1
        x"28800051" when x"0008", --           la r2,UART_ISR_TX + 1
        x"1841f300" when x"000c", --           st r1, UART_RX_Stat
        x"1881f110" when x"0010", --           st r2, UART_TX_Stat
        x"37800010" when x"0014", --           lar r30,_Loop
        x"37400014" when x"0018", --           lar r29,_Uart
        x"37000014" when x"001c", --           lar r28,_Wait
        x"50000000" when x"0020", --           een
        x"403a3003" when x"0024", -- _Loop:     brnz r29,r3
        x"403c0001" when x"0028", --           br r30
        x"1801f114" when x"002c", -- _Uart:     st r0,UART_TX_Data
        x"403c3002" when x"0030", -- _Wait:     brzr r30,r3
        x"40380001" when x"0034", --           br r28
        x"0801f304" when x"0040", -- _UART_ISR_RX: ld r0,UART_RX_Data
        x"28c00001" when x"0044", --           la r3,1
        x"f0000000" when x"0048", --           rfi
        x"28c00000" when x"0050", -- _UART_ISR_TX: la r3,0
        x"f0000000" when x"0054", --           rfi
        x"f8000000" when others;         

        --                                     UART_RX_Stat: .equ 0xFFFFF300
        --                                     UART_RX_Data: .equ 0xFFFFF304
        --                                     UART_TX_Stat: .equ 0xFFFFF110
        --                                     UART_TX_Data: .equ 0xFFFFF114
        --
        --00000000 0000000000 28c00000 la r3, 0 ; Load a zero into r3 (flag)
        --00000004 0000000004 28400041 la r1, UART_ISR_RX + 1 ; Prepare
        --                      ; to set the ISR vector plus set the IE bit for UART
        --                      ; RX

```

```

--00000008 0000000008 28800051 la r2, UART_ISR_TX + 1 ; Prepare --
; to set the
-- ; ISR vector plus set the IE bit for UART TX
--0000000c 0000000012 1841f300 st r1, UART_RX_Stat ; Start the
-- ; UART Receiver
--00000010 0000000016 1881f110 st r2, UART_TX_Stat ; Start the
-- ; UART Transmitter
--00000014 0000000020 3780000c lar r30, Loop
--00000018 0000000024 37400010 lar r29, Uart
--0000001c 0000000028 37000010 lar r28, Wait
--00000020 0000000032 50000000 een ; enable interrupts
--
-- ;
--00000024 0000000036 403a3003 Loop: brnz r29, r3 ; if we have --
; data branch to UART
--00000028 0000000040 403c0001 br r30 ; Do nothing
--
--0000002c 0000000044 1801f114 Uart: st r0, UART_TX_Data ; Store
-- ; r0 into TX register
-- ;
--00000030 0000000048 403c3002 Wait: brzr r30, r3 ; Data send ?
-- ; Branch to Loop
--00000034 0000000052 40380001 br r28 ; else stay in wait
-- ;
-- ;
-- ; UART RX interrupt service routine
-- ;
-- ;
-- .org 40H ; ISRs must start at 00000XX0H, where XX
-- ; is any 8-bit value
--00000040 0000000064 0801f304 UART_ISR_RX: ld r0, UART_RX_Data
-- ; Load the RX value into r0
--00000044 0000000068 28c00001 la r3, 1 ; (set flag) r3 gets 1
--00000048 0000000072 f0000000 rfi
-- ;
-- ;
-- ; UART TX interrupt service routine
-- ;
-- ;
-- .org 50H
--00000050 0000000080 28c00000 UART_ISR_TX: la r3, 0
--00000054 0000000084 f0000000 rfi

```

AYK-14 LED Switch Program

```

-- LED TEST
with address1 select
    dataout1 <=
        x"141e" when x"0000",
        x"6210" when x"0001",
        x"000c" when x"0002",
        x"6311" when x"0003",
        x"0000" when x"0004",
        x"0c15" when x"0005",
        x"06f0" when x"0006",

```

```

        x"f010" when x"0007",
        x"06e0" when x"0008",
        x"f000" when x"0009",
        x"0600" when x"000a",
        x"0018" when x"000b",
        x"055e" when x"000c",
        x"2d5f" when x"000d",
        x"b90c" when x"000e",
        x"0000" when x"000f",
        x"0000" when others;

-- LED TEST
with address2 select
    dataout2 <=
        x"141e" when x"0000", -- _start:      SBR R1,14
        x"6210" when x"0001", --                  LSOR R1
        x"000c" when x"0002", --                  LK R15,LED_Addr
        x"6311" when x"0003", --                  LK R14,SW_Addr
        x"0000" when x"0004", --                  LL R0,_loop
        x"0c15" when x"0005", --                  LI R5,R14
        x"06f0" when x"0006", --                  SI R5,R15
        x"f010" when x"0007", -- _loop:
                                         LPR R0
                                         --
                                         x"06e0" when x"0008",
                                         x"f000" when x"0009",
                                         x"0600" when x"000a",
                                         x"0018" when x"000b",
                                         x"055e" when x"000c",
                                         x"2d5f" when x"000d",
                                         x"b90c" when x"000e",
                                         x"0000" when x"000f",
                                         x"0000" when others;

```

APPENDIX I

UART Source Code

UART.v

```
`timescale 1ns / 1ps
///////////////////////////////
// Company:
// Engineer:
//
// Create Date:      21:25:50 10/10/2006
// Design Name:
// Module Name:     UART
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
///////////////////////////////
module UART(Data, Serial_out, UART_Done, Read, Write, Sel, CE, clk,
            reset, Serial_in);

    parameter CICTL = 2'b00, CIN = 2'b01, COSTAT = 2'b10, COUT = 2'b11;
    parameter idle = 1'b0, B_Ready = 1'b1;
    parameter Buff_size = 8;

    //To-From SRC
    inout [31:0] Data;
    output UART_Done;
    output Serial_out;

    //From SRC
    input Read;
    input Write;
    input [1:0] Sel;
    input CE;
    input clk;
    input reset;
    input Serial_in;

    //Internal Signals
    reg [31:0] Data_Int;
    reg [7:0] TX_datareg;
    reg Byte_ready;
    reg TX_flag;
```

```

    reg [1:0] state, next_state;
    reg TX_Status;
    reg RX_Status;
    reg [7:0] RX_datareg;
    wire Serial_in;
    wire [7:0] RCV_datareg;

    assign UART_Done = CE;
    assign Data = (Read && CE) ? Data_Int : 32'b0;

/*
*****UART_Receiver RX(RCV_datareg, load, Serial_in, reset, clk,
    RX_enable); //Receiver
*****UART_Transmitter TX(Serial_out, TX_clear, TX_datareg, Byte_ready,
    TX_enable, clk, reset); //Transmitter
*****Clock_Prog CLK(RX_enable, TX_enable, clk, reset); //Clock(RX/TX
    // Enables)
***** */

/*TX Data Register*/
always @ (posedge clk) begin
    if(reset) begin
        TX_flag <= 0;
        TX_datareg <= 0;
    end
    else begin
        TX_flag <= 0;
        if(Write && CE && Sel == COUT) begin
            TX_datareg <= Data[7:0];
            TX_flag <= 1;
        end
    end
end

/*Transmit Status Register*/
always @ (posedge clk) begin
    if(reset) TX_Status <= 1;
    else if(Write && CE && Sel == COUT) TX_Status <= 0;
    else if(TX_clear) TX_Status <= 1;
end

/*Receive Status Register*/
always @ (posedge clk) begin
    if(reset) RX_Status <= 0;
    else if(load) RX_Status <= 1;
    else if(Read && CE && Sel == CIN) RX_Status <= 0;
end

/*Data Out Mux*/
always @ (Sel or TX_Status or RX_Status or RX_datareg) begin
    if(Sel == COSTAT) Data_Int <= {TX_Status, 31'b0};
    else if(Sel == CICL) Data_Int <= {RX_Status, 31'b0};
    else Data_Int <= {24'b0, RX_datareg};
end

*/

```

```

/*RX UART STATE MACHINE*/
always @ (posedge clk) begin
    if(reset) begin rx_uart_state <= rx_uart_idle;
    else rx_uart_state <= rx_uart_next_state;
end

always @ (rx_uart_state or load or Read_Rx or Write_Rx or RxBuffStatus)
begin
    load_RxBuff <= 0;
    inc_RxBuff_status <= 0;
    case(rx_uart_state)
        rx_uart_idle: begin
            if(load) rx_uart_next_state <= rx_uart_loading;
            else rx_uart_next_state <= rx_uart_idle;
        end

        rx_uart_loading: begin
            if(Read_Rx == Write_Rx && RxBuffStatus == 4'b1000)
                rx_uart_next_state <= rx_uart_idle;
            else begin
                rx_uart_next_state <= Rx_uart_idle;
                load_RxBuff <= 1;
                inc_RxBuff_status <= 1;
            end
        end

        default: rx_uart_next_state <= rx_uart_idle;
    endcase
end

/*RX SRC STATE MACHINE*/
always @ (posedge clk) begin
    if(reset) begin rx_src_state <= rx_src_idle;
    else rx_src_state <= rx_src_next_state;
end

always @ (rx_src_state or CE or Sel or Read) begin
    case(rx_src_state)
        rx_src_idle: begin
            if(CE && Sel == COUT && Read) begin
                rx_src_next_state <= rx_src_reading;
            end
            else rx_src_next_state <= rx_src_idle;
        end

        rx_src_reading: begin
            rx_src_next_state <= rx_src_idle;
            read_RxBuff <= 1;
            dec_RxBuff_status <= 1;
        end

        default: rx_src_next_state <= rx_src_idle;
    endcase
end

/*RX BUFFER*/
always @ (posedge clk) begin
    if(reset) RxBuff <= 0;

```

```

    else begin
        if(load_RxBuff) begin
            RxBuff[Write_Rx] <= RCV_datareg;
            if(Write_Rx == 7) Write_Rx <= 0;
            else Write_Rx <= Write_Rx + 1;
        end
        if(read_RxBuff) begin
            RX_Datareg <= RxBuff[Read_Rx];
            if(Read_Rx == 7) Read_Rx <= 0;
            else Read_Rx <= Read_Rx + 1;
        end
    end
end

/*RX BUFFER STATUS*/
always @ (posedge clk) begin
    if(reset) RxBuffStatus <= 0;
    else begin
        if(inc_RxBuff && ~dec_RxBuff) begin
            RxBuffStatus <= RxBuffStatus + 1;
        end
        else if(~inc_RxBuff && dec_RxBuff) begin
            RxBuffStatus <= RxBuffStatus - 1;
        end
    end
end
end

/*****************************************/
/*RX UART STATE MACHINE*/
always @ (posedge clk) begin
    if(reset) begin rx_uart_state <= rx_uart_idle;
    else rx_uart_state <= rx_uart_next_state;
end

always @ (rx_uart_state or load or Read_Rx or Write_Rx or RxBuffStatus)
begin
    load_RxBuff <= 0;
    inc_RxBuff_status <= 0;
    case(rx_uart_state)
        rx_uart_idle: begin
            if(load) rx_uart_next_state <= rx_uart_loading;
            else rx_uart_next_state <= rx_uart_idle;
        end
        rx_uart_loading: begin
            if(Read_Rx == Write_Rx && RxBuffStatus == 4'b1000)
                rx_uart_next_state <= rx_uart_idle;
            else begin
                rx_uart_next_state <= Rx_uart_idle;
                load_RxBuff <= 1;
                inc_RxBuff_status <= 1;
            end
        end
        default: rx_uart_next_state <= rx_uart_idle;
    endcase
end

/*RX SRC STATE MACHINE*/

```

```

always @ (posedge clk) begin
    if(reset) begin rx_src_state <= rx_src_idle;
    else rx_src_state <= rx_src_next_state;
end

always @ (rx_src_state or CE or Sel or Read) begin
    case(rx_src_state)
        rx_src_idle: begin
            if(CE && Sel == COUT && Read) begin
                rx_src_next_state <= rx_src_reading;
            end
            else rx_src_next_state <= rx_src_idle;
        end
        rx_src_reading: begin
            rx_src_next_state <= rx_src_idle;
            read_RxBuff <= 1;
            dec_RxBuff_Status <= 1;
        end
        default: rx_src_next_state <= rx_src_idle;
    endcase
end

/*RX BUFFER*/
always @ (posedge clk) begin
    if(reset) RxBuff <= 0;
    else begin
        if(load_RxBuff) begin
            RxBuff[Write_Rx] <= RCV_datareg;
            if(Write_Rx == 7) Write_Rx <= 0;
            else Write_Rx <= Write_Rx + 1;
        end
        if(read_RxBuff) begin
            RX_Datareg <= RxBuff[Read_Rx];
            if(Read_Rx == 7) Read_Rx <= 0;
            else Read_Rx <= Read_Rx + 1;
        end
    end
end

/*RX BUFFER STATUS*/
always @ (posedge clk) begin
    if(reset) RxBuffStatus <= 0;
    else begin
        if(inc_RxBuff && ~dec_RxBuff) begin
            RxBuffStatus <= RxBuffStatus + 1;
        end
        else if(~inc_RxBuff && dec_RxBuff) begin
            RxBuffStatus <= RxBuffStatus - 1;
        end
    end
end

/*****************************************/
/*Control signals for TX*/
always @ (posedge clk) begin
    if(reset) state <= idle;

```

```

    else state <= next_state;
end

always @ (state or TX_flag) begin
  case(state)
    idle: begin Byte_ready <= 0;
      if(TX_flag) begin
        next_state <= B_Ready;
      end
      else begin
        next_state <= idle;
      end
    end
    B_Ready: begin
      Byte_ready <= 1;
      next_state <= idle;
    end
  endcase
end

endmodule

```

UART_Receiver.v

```

`timescale 1ns / 1ps
///////////////////////////////
// Company:
// Engineer:
//
// Create Date: 09:28:07 10/07/2006
// Design Name:
// Module Name: UART_Receiver
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
///////////////////////////////
module UART_Receiver(RCV_datareg, load, Serial_in, reset_, clk,
                     RX_enable);

  parameter word_size = 8;
  parameter half_word = word_size/2;
  parameter Num_counter_bits = 4;
  parameter Num_state_bits = 2;
  //state encoding
  parameter idle = 2'b00;
  parameter starting = 2'b01;
  parameter receiving = 2'b10;

```

```

parameter loading = 2'b11;
parameter all_ones = 9'b1_1111_1111;

//outputs
output [word_size-1:0] RCV_datareg;      //RCV data Register
output load;                           //signal to load RX Status register

//inputs
input Serial_in;                      //serial in line from PC
input RX_enable;                      //set to baude rate
input reset_;                          //SRC reset
input clk;                            //SRC Clk (50 MHz)

//registers
reg [word_size-1:0] RCV_datareg;
reg [word_size-1:0] RCV_shftreg;
reg [Num_counter_bits-1:0] Sample_counter;
reg [Num_counter_bits:0] Bit_counter;
reg [Num_state_bits-1:0] state, next_state;
reg inc_Bit_counter, clr_Bit_counter;
reg inc_Sample_counter, clr_Sample_counter;
reg shift;
reg load;
reg read_not_ready_out, read_not_ready_in, Error1, Error2;

//****************************************************************************
//state memory
always @ (posedge clk) begin
  if(reset_) state <= idle;
  else state <= next_state;
end

//next state and conditional outputs
always @ (state or Serial_in or read_not_ready_in or Sample_counter or
Bit_counter or RX_enable) begin
  read_not_ready_out = 0;
  clr_Sample_counter = 0; inc_Sample_counter = 0;
  clr_Bit_counter = 0; inc_Bit_counter = 0;
  shift = 0;
  load = 0;
  Error1 = 0; Error2 = 0;
  next_state = state;

  case(state)
    idle: begin
      if(Serial_in == 0) begin      //if serial in is a zero then ...
        next_state = starting;    //go to starting state
      end
    end
    starting: begin
      if(Serial_in == 1) begin      //if serial in goes high to soon...
        next_state = idle; //we dont have a start bit so back to idle
        clr_Sample_counter = 1; //clear sample counter
      end
      else begin
        if(Sample_counter == half_word-1) begin //if its a start bit
          // then ...
        end
      end
    end
  endcase
end

```

```

        if(RX_enable) begin
            next_state = receiving;    //next state is receiving and ...
            clr_Sample_counter = 1; //clear the sample counter
        end
    end
    else if(RX_enable)    //else on an RX enable then ...
        inc_Sample_counter = 1;      //increment the sample counter
    end
end
receiving: begin
    if(Sample_counter < word_size-1) begin //if we havent sampled
                                            // the bit enough ...
        if(RX_enable) //on RX enable ...
            inc_Sample_counter = 1;      //increment the sample counter
        end
        else begin //else ...
            if(RX_enable) begin
                clr_Sample_counter = 1; //clear the sample counter
                if(Bit_counter != word_size) begin //if we dont have a
                                            // word then ...
                    shift = 1; //shift the bits and ...
                    inc_Bit_counter = 1; //increment the bit counter
                end
                else begin //else we have a word and ...
                    read_not_ready_out = 1; //set read not ready out high and
                    clr_Bit_counter = 1; //clear the bit counter
                    if(read_not_ready_in) Error1 = 1; //Error 1
                    else if(Serial_in == 0) Error2 = 1;
                    else load = 1; //no errors then load RCV datareg
                    next_state = idle; //next state
                end
            end
        end
    end
    default: next_state = idle; //default state is idle
endcase
end

//****************************************************************************
//RCV Data Register
always @ (posedge clk) begin
    if(reset_) RCV_datareg <= 0;
    else if(load) RCV_datareg <= RCV_shftreg;
end

//RCV Shift Register
always @ (posedge clk) begin
    if(reset_) RCV_shftreg <= 0;
    else
        if(shift) RCV_shftreg <= {Serial_in, RCV_shftreg[word_size-1:1]};
end

//Sample Counter Register with RX_enable enable
always @ (posedge clk) begin
    if(reset_) Sample_counter <= 0;
    else begin
        if(clr_Sample_counter) Sample_counter <= 0;

```

```

        else if(inc_Sample_counter)
            Sample_counter <= Sample_counter + 1;
    end
end

//Bit Counter Register
always @ (posedge clk) begin
    if(reset_) Bit_counter <= 0;
    else begin
        if(clr_Bit_counter) Bit_counter <= 0;
        else if(inc_Bit_counter) Bit_counter <= Bit_counter + 1;
    end
end

endmodule

```

UART_Transmitter.v

```

`timescale 1ns / 1ps
///////////////////////////////
// Company:
// Engineer:
//
// Create Date:      22:04:07 10/10/2006
// Design Name:
// Module Name:     UART_Transmitter
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////
module UART_Transmitter(Serial_out, clear, XMT_datareg, Byte_ready,
TX_enable, clk, reset_);

parameter word_size = 8;          // Size of data word, e.g. 8 bits
parameter one_hot_count = 4;     // Number of one-hot states
parameter state_count = one_hot_count; // Number of bits in state
                                    // register
parameter size_bit_count = 3;    // Size of the bit counter, e.g. 4
                                // Must count to word_size + 1
parameter idle = 4'b0001;        // one-hot state encoding
parameter waiting = 4'b0010;
parameter holding = 4'b0100;
parameter sending = 4'b1000;
parameter all_ones = 9'b1_1111_1111; // Word + 1 extra bit

//outputs
output  clear;

```

```

output Serial_out;           //Serial output to data channel

//inputs
Input [word_size-1:0] XMT_datareg; //host data bus containing data
                                  // word
input Byte_ready;      //used by host to signal ready
input TX_enable;        //set to baude rate
input clk;              //SRC clock (50 MHz)
input reset_;           //SRC reset

//registers
reg [word_size:0] XMT_shftreg; // Transmit Shift Register:
                                // {data, start bit}
reg Load_XMT_shftreg; // Flag to load the XMT_shftreg
reg [state_count-1:0] state, next_state; // State machine controller
reg [size_bit_count:0] bit_count; // Counts the bits that are
                                // transmitted
reg [size_bit_count:0] start_bit_count; // Counts the bits that are
                                // transmitted
reg clear; // Clears bit_count after last bit is sent
reg shift; // Causes shift of data in XMT_shftreg
reg start; // Signals start of transmission

assign Serial_out = XMT_shftreg[0]; // LSB of shift register

//****************************************************************************
//State Memory(Transitions)
always @ (posedge clk) begin
  if (reset_ == 1) state <= idle;
  else state <= next_state;
end

//Output and Next State Logic
always @ (state or Byte_ready or bit_count or start_bit_count or
TX_enable) begin
  Load_XMT_shftreg = 0;
  clear = 0;
  shift = 0;
  start = 0;
  next_state = state;

  case (state)
    idle: begin
      if(Byte_ready == 1) begin //when we have a byte ready to send
        Load_XMT_shftreg = 1; //Load the XMT shift register and...
        next_state = waiting; //enter the waiting state
      end
    end
    waiting: begin
      if(TX_enable) begin //when we get a TX enable then ...
        start = 1; //output a start bit and ...
        next_state = holding; //enter the holding state
      end
    end
    holding: begin
      if(TX_enable) begin //on the next TX enable then ...
        next_state = sending; //enter the sending state
      end
    end
  endcase
end

```

```

        end
    end
    sending: begin
        if (bit_count != word_size + 1) begin //if we havent outputed
            // all 8 bits + stop then ...
            shift = 1; //shift the bits in the shift register, output on
            // serial line and ...
            next_state = holding; //enter the holding state
        end
        else begin
            clear = 1; //if we have outputed all 8 bits + stop then ...
            next_state = idle; //return to the idle state
        end
    end
    default: next_state = idle; //default next state is idle
endcase
end

//****************************************************************************
//XMT Shift Register
always @ (posedge clk) begin
    if(reset_) XMT_shftreg <= all_ones;
    else begin
        if (Load_XMT_shftreg) XMT_shftreg <= {XMT_datareg,1'b1}; // Load
        // shift reg, insert stop bit
        else if (start) XMT_shftreg[0] <= 0; // Signal start of
        // transmission
        else if (shift) XMT_shftreg <= {1'b1, XMT_shftreg[word_size:1]};
        // Shift right, fill with 1's
    end
end

//Bit Count Register
always @ (posedge clk) begin
    if(reset_) bit_count <= 0;
    else begin
        if(clear) bit_count <= 0;
        else if(shift) bit_count <= bit_count + 1;
    end
end
endmodule

```

Clock_Prog.v

```

`timescale 1ns / 1ps

module Clock_Prog (RX_shift, TX_shift, clk, reset_);
    /* Baud_Rate = 9600;
    Clk_Cycle = 20 ns; */
    // parameter RX_baud = 650;
    // parameter TX_baud = 5200;
    /* Baud Rate = 19200*/
    // parameter RX_baud = 1300;

```

```

//      parameter TX_baud = 10400;
/* Baud Rate for Model Sim*/
parameter RX_baud = 8;
parameter TX_baud = 64;

input clk;
input reset_;

output RX_shift;
output TX_shift;

reg RX_shift;
reg TX_shift;
reg [13:0] RX_count;
reg [13:0] TX_count;

always @ (posedge clk) begin
  if(reset_) begin
    RX_count = 0;
    TX_count = 0;
    RX_shift = 0;
    TX_shift = 0;
  end

  RX_shift = 0; TX_shift = 0;
  RX_count = RX_count + 1;
  TX_count = TX_count + 1;
  if(RX_count == RX_baud) begin
    RX_count = 0;
    RX_shift = 1;
  end
  if(TX_count == TX_baud) begin
    TX_count = 0;
    TX_shift = 1;
  end
end
endmodule

```

APPENDIX J

SRC Instruction Set Users Manual

Contents

1 Instruction-Set

ALU IMMEDIATE
ALU REGISTER
ALU UNARY
BRANCH
BRANCH LINK
EDI
EEN
LA
LA
LAR
LD
LD
LDR
NOP
RFI
RI
SHC
SHC
SHL
SHL
SHR
SHR
SHRA
SHRA
ST
ST
STOP
STR
SVI

[Index](#) | [Next](#)

ALU IMMEDIATE

<opcode> <ra> , <rb> , <c2>

opcode	ra	rb	c2
31	27	26	0

Description:

ALU Immediate Instructions

Operands:

opcode

Coding	Syntax	Description
01101	addi	Add rb to immediate constant, and put result into ra
10111	ori	"OR" rb and immediate constant, and put result into ra
10101	andi	"AND" rb and immediate constnat, and put result into ra

ra

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

c2

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxx	<value: hex 17bit-value>	17-bit value/(label) sign extended to 31-ExtractToLong

[Index](#) | [Next](#)

Generated with the CoWare Documentation Generator

Fri Aug 31
2007

ALU REGISTER

<opcode> <ra> , <rb> , <rc>

opcode	ra	rb	rc	000000000000
31		27	26	0

Description:

ALU Register Instructions

Operands:

opcode

Coding	Syntax	Description
01100	add	Add rb to rc, and put result into ra
01110	sub	Subtract rc from rb, and put result into ra
10100	and	"AND" rb and rc, and put result into ra
10110	or	"OR" rb and rc, and put result into ra

ra

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31
<i>rb</i>		
Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31
<i>rc</i>		
Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

ALU UNARY

<opcode> <ra> , <rc>

opcode	ra	00000	rc	000000000000
31		27	26	22 21 17 16 12 11 0

Description:

ALU Unary Instructions

Operands:

opcode		
Coding	Syntax	Description
01111	neg	Place 2's compliment negative of rc into ra
11000	not	Place logical "NOT" of rc into ra

ra

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rc

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

BRANCH

<c3>

01000	00000	rb	rc	000000000	c3
31	27	26	22	21	17 16 12 11 3 2 0

Description:

Branch to target in rb if rc satisfies condition c3

Operands:

rb

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rc

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

c3

Coding	Syntax	Description
000	brnv	branch never
001	br <rb>	branch unconditionally to rb
010	brzr <rb> , <rc>	branch to rb if rc is zero
011	brnz <rb> , <rc>	branch to rb if rc is non-zero
100	brpl <rb> , <rc>	branch to rb if rc is positive or zero (sign is plus)
101	brmi <rb> , <rc>	branch to rb if rc is negative (sign is minus)

 BRANCH LINK

<c3>

01001	ra	rb	rc	000000000	c3
31 27 26 22 21 17 16 12 11 3 2 0					

Description:

Branch to rb if rc satisfies c3, and save PC in ra

Operands:
ra

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rc

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

c3

Coding	Syntax	Description
000	brlnv <ra>	Do not branch, but save PC in ra
001	brel <ra> , <rb>	Branch unconditionally to rb, and save PC in ra
010	brlzr <ra> , <rb> , <rc>	Branch to rb if rc is zero, and save PC in ra
011	brlnz <ra> , <rb> , <rc>	Branch to rb if rc is non-zero, and save PC in ra
100	bndlpl <ra> , <rb> , <rc>	Branch to rb if rc is positive, and save PC in ra
101	bndlmi <ra> , <rb> , <rc>	Branch to rb if rc is negative, and save PC in ra

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

EDI

edi

01011	00000000000000000000000000000000
31	27

Description:

Exception disable. Clear overall exception enable

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

EEN

een

01010	00000000000000000000000000000000
31	27 26

Description:

Exception enable. Set overall exception enable bit

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

LA

la <ra> , <c2>

00101	ra	00000	c2				
31	27	26	22	21	17	16	0

Description:

Load value of absolute address into ra; rb is register 0

Operands:

ra

Coding	Syntax	Description
--------	--------	-------------

idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31
<i>c2</i>		
Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxx	<value: hex 17bit-value>	17-bit value/(label) sign extended to 31-ExtractToLong

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

LA

la <ra> , <c2> (<rb>)

00101	ra	rb	c2
31	27	26	22 21 17 16 0

Description:

Load value of displacement address into ra

Operands:

ra	Coding	Syntax	Description
	idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31
<i>rb</i>			
rb	Coding	Syntax	Description
	idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31
<i>c2</i>			
c2	Coding	Syntax	Description
	value=xxxxxxxxxxxxxxxxxx	<value: hex 17bit-value>	17-bit value/(label) sign extended to 31-ExtractToLong

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

LAR

lar <ra> , <(c1+CURRENT_ADDRESS): hex 32bit-value>

00110	ra	c1=xxxxxxxxxxxxxxxxxxxxxx
31	27	26 22 21 0

Description:

Load value of relative address into ra

Operands:

ra	Coding	Syntax	Description
	idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

LD

ld <ra> , <c2>

00001	ra	00000	c2
31	27	26 22 21 17 16 0	

Description:

Load from absolute address; rb is register 0

Operands:

ra	Coding	Syntax	Description
	idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

c2

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxxxxxx	<value: hex 17bit-value>	17-bit value/(label) sign extended to 31-ExtractToLong

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

LD

ld <ra> , <c2> (<rb>)

00001	ra	rb	c2	
31	27	26	22	21 17 16 0

Description:

Load from displacement address

*Operands:**ra*

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

c2

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxx	<value: hex 17bit-value>	17-bit value/(label) sign extended to 31-ExtractToLong

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007[Index](#) | [Previous](#) | [Next](#)

LDR

ldr <ra> , <(c1+CURRENT_ADDRESS): hex 32bit-value>

00010	ra	c1=xxxxxxxxxxxxxxxxxxxxxx	
31	27	26	22 21 0

Description:

Load from a relative address

*Operands:**ra*

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

NOP

nop

Description:

No operation. Used to insert pipeline bubble

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

RFI

rfi

Description:

Return from interrupt PC \leftarrow TPC; enable exceptions

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

RI

ri <*ra*> , <*rb*>

10001	ra	rb	00000000000000000000
31	27	26	22 21 17 16 0

Description:

Restore II and IPC from ra and rb, respectively

Operands:

ra

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

SHC

<opcode> <ra> , <rb> , <rc>

opcode	ra	rb	rc	000000000000
31	27	26	21	17 16 12 11 0

Description:

shift rb left circularly into ra by count in rc; c3 is 0

Operands:

opcode

Coding	Syntax	Description
11101	shc	

ra

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rc

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

SHC

<opcode> <ra> , <rb> , <c3>

opcode	ra				rb				000000000000												c3		
31			27	26			22	21			17	16									5	4	0

Description:

shift rb left circularly into ra by constant c3

Operands:

opcode

Coding	Syntax	Description
11101	shc	

c3

Coding	Syntax	Description
value=xxxxx	<value: unsigned decimal 5bit-value>	5-bit unsigned immediate value

ra

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

SHL

<opcode> <ra> , <rb> , <rc>

opcode	ra				rb				rc				000000000000											
31			27	26			22	21			17	16			12	11					0			

Description:
shift rb left into ra by count in rc; c3 is 0

Operands:

opcode

Coding	Syntax	Description
11100	shl	

ra

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rc

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

SHL

<opcode> <ra> , <rb> , <c3>

opcode	ra	rb	000000000000												c3			
31		27	26		22	21		17	16						5	4		0

Description:

shift rb left into ra by constant c3

Operands:

opcode

Coding	Syntax	Description
11100	shl	

c3

Coding	Syntax	Description

value=xxxxxx	<value: unsigned decimal 5bit-value>	5-bit unsigned immediate value
<i>ra</i>		
Coding	Syntax	Description
idx=xxxxxx r<idx: unsigned decimal 5bit-value>		
General Purpose Register r0 - r31		
<i>rb</i>		
Coding	Syntax	Description
idx=xxxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

SHR

<opcode> <ra> , <rb> , <rc>

opcode	ra	rb	rc	00000000000000
31	27	26	21	17 16 12 11 0

Description:

shift rb right into ra by count in rc; c3 is 0

Operands:

opcode

Coding	Syntax	Description
11010	shr	

ra

Coding	Syntax	Description
idx=xxxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rc

Coding	Syntax	Description
idx=xxxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

SHR

<opcode> <ra> , <rb> , <c3>

opcode	ra	rb	000000000000												c3
31	27	26	22	21	17	16	5	4	0						

Description:

shift rb right into ra by constant c3

Operands:

opcode

Coding	Syntax	Description
11010	shr	

c3

Coding	Syntax	Description
value=xxxxx	<value: unsigned decimal 5bit-value>	5-bit unsigned immediate value

ra

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

<opcode> <ra> , <rb> , <rc>

opcode	ra	rb	rc	000000000000

31			27	26			22	21			17	16			12	11								0
----	--	--	----	----	--	--	----	----	--	--	----	----	--	--	----	----	--	--	--	--	--	--	--	---

Description:

shift rb right with sign-extend into ra by count in rc; c3 is 0

Operands:

opcode

Coding	Syntax	Description
11011	shra	

ra

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rc

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

SHRA

<opcode> <ra> , <rb> , <c3>

opcode	ra	rb	000000000000	c3
31		27 26	22 21	17 16 5 4 0

Description:

shift rb right with sign-extend into ra by constant c3

Operands:

opcode

Coding	Syntax	Description
11011	shra	

c3

Coding	Syntax	Description
value=xxxxxx	<value: unsigned decimal 5bit-value>	5-bit unsigned immediate value

ra

Coding	Syntax	Description
idx=xxxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

ST

st <ra> , <c2>

00011	ra	00000	c2
31	27	26	22 21 17 16 0

Description:

Store into absolute address; rb is register 0

Operands:

ra

Coding	Syntax	Description
idx=xxxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

c2

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxx	<value: hex 17bit-value>	17-bit value/(label) sign extended to 31-ExtractToLong

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

ST

st <*ra*> , <*c2*> (<*rb*>)

Description:

Store into displacement address

Operands:

ra

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

c2

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxx	<value: hex 17bit-value>	17-bit value/ (label) sign extended to 31-ExtractToLong

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

STOP

stop

11111	00000000000000000000000000000000		
31	27	26	0

Description:

Set RUN to zero, halting the machine

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#) | [Next](#)

STR

`str <ra> , <(c1+CURRENT_ADDRESS): hex 32bit-value>`

00100	ra	c1=xxxxxxxxxxxxxxxxxxxxxx
31	27	26 22 21 0

Description:

Store into relative address

Operands:

ra

Coding	Syntax	Description
idx=xxxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Fri Aug 31 2007

[Index](#) | [Previous](#)

SVI

svi <*ra*> , <*rb*>

10000	ra	rb	00000000000000000000				
31	27	26	22	21	17	16	0

Description:

Save II and IPC in ra and rb, respectively

Operands:

ra

Coding	Syntax	Description
Idx=xxxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

rb

Coding	Syntax	Description
idx=xxxxx	r<idx: unsigned decimal 5bit-value>	General Purpose Register r0 - r31

[Index](#) |
[Previous](#)

Generated with the CoWare Documentation Generator

Fri Aug 31
2007

APPENDIX K

AYK-14 Instruction Set Users Manual

Contents

1 Instruction-Set

A
ANDK
AR
BS
C
CBR
CK
DROR
IROR
JLR
JLR*
L
LA
LALD
LALS
LARD
LC
LD
LI
LJ
LJE
LJGE
LJLS
LJNE
LK
LL
LLRS
LPR
LR
LSU
ORR
S
SBR
SD
SI
SM
SXI
SZ
XJ
ZBR

[Index](#) | [Next](#)

A

A $\langle a \rangle$, $\langle y \rangle$, $\langle m \rangle$

Description:

This instruction adds the contents of memory address Y to the contents of R[a], stores the sum into R[a], and sets the condition code in accordance with the resulting quantity R[a]. The overflow and carry designators are set in accordance with the results of the operation.

Operands:

y

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

Index

Generated with the CoWare Documentation Generator

Tue Aug 28
2007

[Index](#) | [Previous](#) | [Next](#)

ANDK

ANDK <*a*> , <*y*> , <*m*>

Description:

This instruction performs the bit-by-bit logical AND of the contents of R[a] and the operand Y, stores the result into R[a] and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are cleared.

Operands:

y

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
--------	--------	-------------

idx=xxxx	R<idx>	
a		
Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Tue Aug 28
[Next](#) Generator 2007

[Index](#) | [Previous](#) | [Next](#)

AR

AR <a> , <m>

010110	00	a	m							
15		10	9	8	7		4	3		0

Description:

This instruction adds the contents of R[m] to the contents of R[a], stores the sum into R[a], and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are set in accordance with the results of the operation.

Operands:

a		
Coding	Syntax	Description
idx=xxxx	R<idx>	
m		
Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Tue Aug 28
[Next](#) Generator 2007

[Index](#) | [Previous](#) | [Next](#)

BS

BS <a> , <y> , <m>

001010	11	a	m	y									
31		26	25	24	23		20	19		16	15		0

Description:

This instruction stores bits 7-0 of the contents of R[a] into the selected byte at memory address Y; the other byte at memory address Y is unchanged. The condition code, overflow, and carry designators are unchanged. Unless otherwise specified for indirect addressing, this instruction uses bit 0 of R[m] as the byte pointer. If m=0, Y=y and the byte pointer=0. A byte pointer of 0 selects the upper byte (bits 15-8); a byte pointer of 1 selects the lower byte (bits 7-0).

Operands:

Y

Coding	Syntax	Description
value=xxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

C

C <a> , <y> , <m>

011000	11	a	m	y
31			26	25 24 23 20 19 16 15 0

Description:

This instruction arithmetically compares the contents of R[a] to the contents of memory address Y and sets the condition code, overflow, and carry designators in accordance with the results of the operation.

Operands:

Y

Coding	Syntax	Description
value=xxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description

idx=xxxx	R<idx>	
a		
Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Tue Aug 28
[Next](#) Generator 2007

[Index](#) | [Previous](#) | [Next](#)

CBR

CBR <a> , <m>

000111	00	a	m								
15		10	9	8	7		4	3	2	1	0

Description:

This instruction arithmetically compares the bit in R[a] specified by the 4-bit literal contained in bits 3-0 (the m- designator) of the instruction with zero and sets the condition code in accordance with the results of the operation. Bit 8 of Status Register 1 will be set for any bit tested; bit 9 of Status Register 1 will be cleared unless bit 15 of R[a] is being tested in which case it will be set equal to bit 15 of R[a]. The overflow and carry designators are cleared.

Operands:

a		
Coding	Syntax	Description
idx=xxxx	R<idx>	

m		
Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Tue Aug 28
[Next](#) Generator 2007

[Index](#) | [Previous](#) | [Next](#)

CK

CK <a> , <y> , <m>

011000	10	a	m	y
--------	----	---	---	---

31 26 25 24 23 20 19 16 15 0

Description:

This instruction arithmetically compares the contents of R[a] to the operand Y and sets the condition code, overflow, and carry designators in accordance with the results of the operation.

Operands:

Y

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

DROR

DROR <*a*>

000010				00		a			1011			
15				10	9	8	7		4	3		0

Description:

This instruction subtracts one from the contents of R[a], stores the difference into R[a], and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are set in accordance with the results of the operation.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

Index | Previous | Generated with the CoWare Documentation Tue Aug 28

IROR

IROR <a>

000010	00	a	1010
15		10	9 8 7 4 3 0

Description:

This instruction adds one to the contents of R[a], stores the sum into R[a], and sets the condition code in accordance with the resulting quantity R[a]. The overflow and carry designators are set in accordance with the results of the operation

Operands:

a

Coding	Syntax	Description
idx=xxxxx	R<idx>	

JLR

JLR <a> , <y> , <m>

101010	10	a	M	y
31		26 25 24 23	20 19	16 15 0

Description:

This instruction adds two to the contents of the P-Register, stores the results into R[a], and jumps to the instruction located at the address specified by the operand Y. The condition code, overflow, and carry designators are unchanged.

Operands:

y

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description

idx=xxxx	R<idx>	
a		
Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

JLR*

JLR <a> , * <y> , <m>

101010	11	a	m	y
31		26	25	24 23 20 19 16 15 0

Description:

This instruction adds two to the contents of the P-register, stores the result into R[a], and jumps to the instruction located at the address specified by the contents of memory address Y. The condition code, overflow, and carry designators are unchanged.

Operands:

y

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

L

L <a> , <y> , <m>

Description:

This instruction loads the contents of memory address Y into R[a] and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are cleared

Operands:

Y

Coding	Syntax	Description
value=xxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

LA

*L*A <*a*> , <*m*>

111110	10	a	m
15			0

Description:

This instruction adds the 4-bit literal contained in bits 3-0 (the m-designator) of the instruction to the contents of R[a], stores the sum into R[a], and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are set in accordance with the results of the operation.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

LALD

LALD <a> , <m>

111101	10	a	m
15		10	9 8 7 4 3 0

Description:

This instruction shifts the double lenght contents of R[a],R[a+1] left by the number of places specified by bits 3-0 (the m-designator) of the instruction with zeros extended to fill and sets the condition code in accordance with the resulting double length quantity in R[a],R[a+1]. The overflow designator is set in accordance with the results of the operation. The carry designator is cleared. R[a] must be an even-numbered register.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

LALS

LALS <a> , <m>

111101	00	a	m
15		10	9 8 7 4 3 0

Description:

This instruction shifts the contents of R[a] left by the number of places specified by bits 3-0 (the m-designator) of the instruction with zeros extended to fill and sets the condition code in accordance with the resulting quantity in R[a]. The overflow designator is set in accordance with the results of the operation. The carry designator is cleared.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

LARD

LARD <a> , <m>

111100	11	a	M
15		10	9 8 7 6 5 4 3 2 1 0

Description:

This instruction shifts the double lenght contents of R[a],R[a+1] right by the number of places specified by bits 3-0 (the m-designator) of the instruction with the R[a] sign extended to fill and sets the condition code in accordance with the resulting double length quantity in R[a],R[a+1]. The overflow and carry designators are cleared. R[a] must be an even-numbered register.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

LC

LC <a> , <m>

111111	01	a	m
15		10	9 8 7 4 3 0

Description:

This instruction arithmetically compares the 4-bit literal contained in bits 3-0 (the m-designator) of the instruction with the contents of R[a] and sets the condition code, overflow and carry designators in accordance with the results of the operation.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

LD

LD <a> , <y> , <m>

000010	11	a	m	y
31		26	25 24 23	20 19 16 15 0

Description:

This instruction loads the double length contents of memory addresses Y,Y+1 into R[a],R[a+1] and sets the condition code in accordance with the resulting double length quantity in R[a],R[a+1]. The overflow and carry designators are cleared. R[a] must be an even numbered register and Y must be an even numbered address.

Operands:

y

Coding	Syntax	Description
value=xxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

LI

LI <a> , <m>

000001	01	a	M	y
31		26	25	24 23 20 19 16 15 0

Description:

This instruction loads the contents of memory address Y* into R[a] and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are cleared.

Operands:

y

Coding	Syntax	Description
value=xxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28

LJ

LJ <d>

101000	01	d	
15		10	9 8 7 0

Description:

This instruction jumps to the instruction located at the address specified by the sign-extended 8-bit displacement (the d-designator) from this jump instruction. The condition code, overflow, and carry designators are unchanged.

Operands:

d

Coding	Syntax	Description
value=xxxxxxxx	<value: signed decimal 8bit-value>	

LJE

LJE <d>

101100	01	d	
15		10	9 8 7 0

Description:

If bit 8 of Status Register 1 contains zero (condition code indicates equal comparison or zero arithmetic result) This instruction jumps to the instruction located at the address specified by the 8 bit sign extended displacement (the d-designator) from this jump instruction. If bit 8 of Status Register 1 contains one next instruction is executed. The condition code, overflow and carry designators are unchanged.

Operands:

d

Coding	Syntax	Description
value=xxxxxxxx	<value: signed decimal 8bit-value>	

LJGE

LJGE <d>

101110	01	d	
15		10	9 8 7 0

Description:

If bit 9 of Status Register 1 contains zero (condition code indicates equal comparison, greater than comparison, zero arithmetic, or non-zero and positive arithmetic result), this instruction jumps to the instruction located at the address specified by the 8-bit sign-extended displacement (the d-designator) from this jump instruction. If bit 9 of Status Register 1 contains one, the next instruction is executed. The condition code, overflow, and carry designators are unchanged.

Operands:

d

Coding	Syntax	Description
value=xxxxxxxx	<value: signed decimal 8bit-value>	

LJLS

LJLS <d>

101111	01	d	
15		10	9 8 7 0

Description:

If bit 9 of Status Register 1 contains one, (condition code indicates less than comparison of non-zero negative arithmetic result) this instruction jumps to the instruction located at the address specified by the 8-bit sign extended displacement (the d-designator) from this jump instruction. If bit 9 of Status Register 1 contains zero, the next instruction is executed. The condition code, overflow, and carry designators are unchanged.

Operands:

d

Coding	Syntax	Description
value=xxxxxxxx	<value: signed decimal 8bit-value>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

LJNE

LJNE <d>

101101	01	d													
15				10	9	8	7								0

Description:

If bit 8 of Status Register 1 contains one (condition code indicates greater than comparison, less than comparison, or non-zero arithmetic result) this instruction jumps to the instruction located at the address specified by the 8 bit sign extended displacement (the d-designator) from this jump instruction. If bit 8 of Status Register 1 contains zero, the next instruction is executed. The condition code, overflow and carry designators are unchanged.

Operands:

d

Coding	Syntax	Description
value=xxxxxxxx	<value: signed decimal 8bit-value>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

LK

LK <a> , <y> , <m>

000001	10	a	m	y											
31				26	25	24	23		20	19		16	15		0

Description:

This instruction loads the operand Y into R[a] and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are cleared

Operands:

y

Coding	Syntax	Description
value=xxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

LL

LL <a> , <m>

111111	00	A	m
15		10	9 8 7 4 3 0

Description:

This instruction loads the 4-bit literal contained in bits 3-0 (the m-designator) of the instruction into bits 3-0 of R[a], clears bits 15-4 of R[a], and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are cleared.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

LLRS

LLRS <a> , <m>

111100	00	A	m
15			0

Description:

This instruction shifts the contents of R[a] right by the number of places specified by bits 3-0 (the m-designator) of the instruction with zeros extended to fill and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are cleared.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

LPR

LPR <a>

000011	00	A	0100
15			0

Description:

This instruction jumps to the instruction at the address specified by the contents of R[a]. The condition code, overflow, and carry designators are unchanged.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | [Next](#)

LR

LR <a> , <m>

000001	00	A	m
15		10	9 8 7 4 3 0

Description:

This instruction loads the contents of R[m] into R[a] and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are cleared.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | [Next](#)

LSU

LSU <a> , <m>

111110	00	A	m
15		10	9 8 7 4 3 0

Description:

This instruction subtracts the 4-bit literal contained in bits 3-0 (the m-designator) of the instruction from the contents in R[a], stores the difference in R[a], and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are set in accordance with the results of the operation.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

ORR

ORR <a> , <m>

011111	00	A	m				
15		10	9	8	7	4	3

Description:

This instruction performs the bit-by-bit logical OR of the contents of R[a] and the contents of R[m], stores the result into R[a], and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are cleared.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

S

S <a> , <y> , <m>

001011	11	a	M	y	
--------	----	---	---	---	--

31				26	25	24	23		20	19		16	15												0
----	--	--	--	----	----	----	----	--	----	----	--	----	----	--	--	--	--	--	--	--	--	--	--	--	---

Description:

This instruction stores the contents of R[a] into memory address Y. The condition code, overflow, and carry designators are unchanged.

Operands:

Y

Coding	Syntax	Description
value=xxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

SBR

SBR <a> , <m>

000101	00	a	m																					
15		10	9	8	7			4	3			0												

Description:

This instruction sets the bit in R[a] specified by the 4-bit literal contained in bits 3-0 (the m-designator) of the instruction leaving the other bits in R[a] unchanged and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are cleared.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description

value=xxxx <value: unsigned decimal 4bit-value>

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

SD

SD $\langle a \rangle$, $\langle y \rangle$, $\langle m \rangle$

Description:

This instruction stores the double length contents of R[a],R[a+1] into memory addresses Y,Y+1. The condition code, overflow, and carry designators are unchanged. R[a] must be an even numbered register and Y must be an even numbered address.

Operands:

Y

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

SI

SI <*a*> , <*m*>

0010111	01	a	m
15			0

Description:

This instruction stores the contents of R[a] into memory address Y*. The condition code, overflow, and carry designators are unchanged.

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

SM

SM <a> , <y> , <m>

001101	11	a	M	y
31		26	25	24 23 20 19 16 15 0

Description:

If m is greater than or equal to a, this instruction stores the contents of sequential registers R[a] through R[m] into sequential memory addresses beginning at Y. If m is less than a, this instruction stores the contents of sequential registers R[a], R[a-1] ... R[0] ... R[m] into sequential memory addresses beginning at Y. The condition code, overflow, and carry designators are unchanged. In this instruction Y equals y (indexing and indirect addressing cannot be used). The beginning address may be either odd or even.

Operands:

y

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

idx=xxxx	R<idx>	
----------	--------	--

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Tue Aug 28
[Next](#) Generator 2007

[Index](#) | [Previous](#) | [Next](#)

SXI

SXI <a> , <m>

001111	01	a	m							
15		10	9	8	7		4	3		0

Description:

This instruction stores the contents of R[a] into memory address Y* (R[m]==address) and increments the contents of R[m] by one. if a=m, the contents of R[a] will be incremented by 1 after being stored into memory address Y*. The condition code, overflow, and carry designators are unchanged

Operands:

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Tue Aug 28
[Next](#) Generator 2007

[Index](#) | [Previous](#) | [Next](#)

SZ

SZ <y> , <m>

010001	11	0000	M	y									
31		26	25	24	23		20	19		16	15		0

Description:

This instruction stores all zeros into memory address Y. The condition code, overflow, and carry designators are unchanged. The a- designator is not used.

Operands:

y

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

[Index](#) | [Previous](#) | [Next](#)

XJ

XJ <a> , <y> , <m>

101001	10	a	M	y
31		26	25	24 23 20 19 16 15 0

Description:

If the contents of R[a] are not equal to zero, this instruction subtracts one from the contents of R[a] and jumps to the instruction located at the address specified by the Y operand. If the contents of R[a] are equal to zero, the next instruction is executed. The condition code, overflow, and carry designators are unchanged.

Operands:

y

Coding	Syntax	Description
value=xxxxxxxxxxxxxxxxxx	<value: unsigned decimal 16bit-value>	

m

Coding	Syntax	Description
idx=xxxx	R<idx>	

a

Coding	Syntax	Description
idx=xxxx	R<idx>	

[Index](#) | [Previous](#) | Generated with the CoWare Documentation Generator Tue Aug 28 2007

ZBR

ZBR <a> , <m>

000110	00	a	m
15		10 9 8 7	4 3 0

Description:

This instruction clears the bit in R[a] specified by the 4-bit literal contained in bits 3-0 (the m-designator) of the instruction leaving the other bits in R[a] unchanged and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are cleared.

*Operands:**a*

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

ZBR

ZBR <a> , <m>

000110	00	a	m
15		10 9 8 7	4 3 0

Description:

This instruction clears the bit in R[a] specified by the 4-bit literal contained in bits 3-0 (the m-designator) of the instruction leaving the other bits in R[a] unchanged and sets the condition code in accordance with the resulting quantity in R[a]. The overflow and carry designators are cleared.

*Operands:**a*

Coding	Syntax	Description
idx=xxxx	R<idx>	

m

Coding	Syntax	Description
value=xxxx	<value: unsigned decimal 4bit-value>	

[Index](#) |
[Previous](#)

Generated with the CoWare Documentation
Generator

Mon Aug 27
2007

BIBLIOGRAPHY

- [1] A. Hoffmann, A. Nohl, G. Braun, and H. Meyr, *A Survey on Modeling Issues Using the Machine Description Language LISA*, , 2001
- [2] A. Hoffmann, F. Fiedler, A. Nohl, and Surender Parupalli, *A Methodology and Tooling Enabling Application Specific Processor Design*, IEEE Conference on VLSI Design, 2005
- [3] G. Hadjiyiannis, S. Hanono, and S. Devadas, *ISDL: An Instruction Set Description Language for Retargetability*, in Proc. of the Design Automation Conference (DAC), Jun 1997
- [4] A. Wieferink, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, G. Braun, and A. Nohl, *System Level Processor/Communication Co-exploration Methodology for Multiprocessor System-On-Chip Platforms*, , 2005
- [5] S. Yang, Y. Qian, H. Tie-Jun, S. Rui, and H. Chao-Huan, *A New HW/SW Co-design Methodology to Generate a System Level Platform Based on LISA*, 2005
- [6] Vincent P. Heuring and Harry F. Jordan, *Computer System Design and Architecture Second Edition*, Pearson Education Inc., 2004
- [7] Digilent Inc. <http://www.digilentinc.com>
- [8] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, *EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability*, In Proc. of the Conference on Design, Automation & Test in Europe, Mar. 1999
- [9] Peter Marwedel, *The MIMOLA Design System: Tools for the Design of Digital Processors*, In Proceedings of the 21st Design Automation Conference, pages 587-593, 1983
- [10] M. Freericks, *The nML Machine Description Formalism*, Tech. Rep. 1991/51, Technische Universitat Berlin, Fachbereich Informatik, Berlin, 1991
- [11] R. Gonzales, *XTensa: A Configurable and Extensible Processor*, IEEE Micro, Mar. 2000

- [12] Rex Coombs, Maggie Graham, and Marilyn Bigalke, “AYK-14 Celebrates 25 Years of Service”, Tester, January 22, 2002, Available Online at: http://www.dcmilitary.com/dcmilitary_archives/stories/012202/13361-1.shtml.
- [13] General Dynamics Information Systems, Software Design Specification for the Extended Memory Reach (EMR) Firmware Update on the VHSIC Processor Module, Contract No: N00163-D-93-0006, Bloomington, MN, Date April 30, 1998.
- [14] Computing Devices International, AN/AYK-14(V) Programmer’s Reference Manual, Volume I, doc. 13211927D, April 1995.
- [15] CoWare, *CoWare Processor Designer Family: Processor Designer Reference Manual*, Product Version V2006.1.0, 2006
- [16] CoWare, *CoWare Processor Designer Family: Processor Design Guide*, Product Version V2006.1.0, 2006
- [17] CoWare, *CoWare Processor Designer Family: Debugger Reference Manual*, Product Version V2006.1.0, 2006
- [18] CoWare, *CoWare Processor Designer Family: HDL Code Generation Guide*, Product Version V2006.1.0, 2006
- [19] Computing Devices International, *AN/AYK-14(V) Programmer’s Reference Manual, Volume I*, doc. 13211927D, April 1995.
- [20] CoWare, *CoWare Processor Designer Family: LISA Modeling Fundamentals*, Product Version V2006.1.0, 2006
- [21] CoWare, *LISATek Product Family: LISA Language Reference Manual*, Product Version V2005.2.2, 2006
- [22] CoWare, *CoWare Processor Designer Family: Documentation Design Guide*, Product Version V2006.1.0, 2006
- [23] Michael D. Cilletti, *Verilog HDL*, Pearson Education Inc, 2003

