Modeling Embedded Processors and Generating Fast Simulators Using the Machine Description Language LISA

Von der Fakultät für Elektrotechnik und Informationstechnik
der Rheinisch–Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften
genehmigte Dissertation

vorgelegt von
Diplom–Ingenieur Stefan Leo Alexander Pees
aus Aachen

Berichter: Universitätsprofessor Dr. sc. techn. Heinrich Meyr
Universitätsprofessor Dr.–Ing. Bernhard Walke

Tag der mündlichen Prüfung: 28. November 2002

Vorwort

Die vorliegende Dissertation entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter des Lehrstuhls für Integrierte Systeme der Signalverarbeitung (ISS) an der Rheinisch-Westfälischen Technischen Hochschule Aachen.

Mein besonderer Dank gilt vor allem Herrn Prof. Dr. sc. techn. Heinrich Meyr, der das Referat für diese Dissertation übernommen und meine Arbeit mit konstruktiver Kritik wissenschaftlich begleitet hat.

Herrn Prof. Dr.-Ing. Bernhard Walke danke ich für die Übernahme des Korreferats und für das meiner Arbeit entgegengebrachte Interesse.

Herrn Dr.-Ing. Vojin Zivojnovic, Herrn Dipl.-Ing. Andreas Ropers, Herrn Dr.-Ing. Andreas Hoffmann und Herrn Dipl.-Ing. Achim Nohl bin ich für die besonders intensive und gute Zusammenarbeit sowie für die konstruktiven Diskussionen und hilfreichen Anregungen zu besonderem Dank verpflichtet.

Andere Projekte der DSP Tools Gruppe haben diese Arbeit beeinflusst. Mein Dank gilt Herrn Dr.-Ing. Thorsten Grötker und Herrn Dr.-Ing. Markus Willems für ihre kritischen und weiterführenden Anregungen.

Frau Wanda Gass und Herrn Subbu Venkat von der Firma Texas Instruments danke ich für die Unterstützung meiner Forschungsarbeit und für die Einblicke in Entwurf und Modellierung moderner Signalprozessoren.

Bei meinen Diplomanden und studentischen Mitarbeitern bedanke ich mich für ihr großes Engagement. Allen Mitarbeitern des Lehrstuhls danke ich für die gute und freundliche Arbeitsatmosphäre.

Mein abschließender und spezieller Dank gilt meiner Familie, vor allem meiner Frau Karin für ihre liebevolle Hilfe und Ermutigung in allen Phasen meiner Arbeit sowie meinen Eltern Eva und Hans Pees für ihr Vertrauen und ihre uneingeschränkte Unterstützung.

Aachen, im August 2003 *Stefan Pees*

Contents

Chapter 1

Introduction

The digitalization of information and entertainment systems today covers the whole area of text, graphics, speech, audio and video processing. A variety of digital signal processing applications like compression, decompression, encryption and different kinds of quality improvements have evolved from these systems. The commercialization of these systems for the consumer market requires cost-effective solutions in silicon. Due to the drop in prices, enormously increased efficiency and advanced miniaturization, such consumer products become more and more available, incorporating application specific architectures that are referred to as *embedded systems.* The probability that one of us has used a product that incorporates embedded systems within the last hour or minute is extremely high. Today they can be found in many products of our every day life. Mobile phones, personal organizers and DVD players are the most obvious ones. But they are also increasingly installed invisible to the user in vehicles, home appliances, cameras, computer peripherals and medical equipment.

At the same time, a universal trend of convergence can be observed. New consumer products increasingly unify functions of different application areas. The third generation mobile telecommunication standard (3G) will provide a single platform for different services that support the transmission of voice, audio, video, text, graphics and other data. This platform demands new versatile architectures that can execute a set of applications instead of just a single application. Programmable components will be crucial element that enables the growth and success of these products.

Furthermore, programmability is an important factor in the system design process. The programmability helps to raise the designer's productivity and the flexibility of software allows late design changes and provides a high degree of reusability, thus shortening the design cycles. For this reason, programmable architectures like off-the-shelf digital signal processors (DSPs), microcontrollers and application-specific instruction-set processors (ASIPs) are increasingly employed into embedded systems and a growing amount of system functions is implemented in software rather than in hardware [68]. Today, one can look at embedded systems as software based solutions that are complemented with dedicated hardware components in order to accelerate computation-intensive operations, to save power or to provide interfaces for input and output.

Designers of today's embedded systems are facing a rapidly growing system complexity. Driven by the advances in semiconductor technology, the amount of system functionality that can be realized

on a single chip is growing enormously. The increasing integration density of integrated circuits has led to a shift from dedicated hardware devices that are spread over several chips towards single-chip solutions. In these *systems-on-chips* (SoC), designers are confronted with the challenge of integrating heterogeneous hardware and software components.

Product innovation constantly shortens product life cycles to the point that companies which fail to enter the market timely with a new product must reduce the prices (and margins) in order to be competitive. Thus, time-to-market is crucial for the profitability of products and market share.

Due to the complexity and time-to-market constraints, the designer's productivity has become a vital factor for successful products. But designers are facing a dilemma – on the one hand the system complexity and heterogeneity is exploding and on the other hand the design time must shrink continuously. The only way out of this dilemma is the use of an appropriate design methodology and efficient design automation tools.

A particular important part in the design methodology of electronic systems is design verification. According to statements from industry, two thirds of the man-power in design teams is spent on verification. Further design challenges are found in the co-design and co-verification of heterogeneous hardware and software components.

Standard approaches are not suited to verify and validate embedded systems consisting of one or more processor cores, their software, dedicated hardware components, buses and interfaces [26]. For this reason, many research groups in the field of electronic design automation have focused on hardware/software verification environments. However, it is surprising that the systematic, efficient modeling and simulation of embedded processors and software has been explored only superficially.

Processor models and simulators are needed in various areas of hardware/software co-design. During processor design, different models are required for instruction set specification and performance analysis. Instruction set simulators are employed for debugging during application software development and co-simulation of systems. Very detailed models are required for the system-level integration of processor cores. However, available processor models seem to be separated into software-oriented and hardware-oriented models. A suitable methodology and appropriate tools are missing that enable the creation of processor models that cover both, the hardware and software side.

Furthermore, the variety of processor models used in the design methodology of embedded systems involves a further problem. Handwritten processor models from independent design groups – such as software and hardware teams – cause severe consistency and verification issues. A retargetable modeling and simulation approach helps the designer to concentrate on the modeling task itself and lets him reuse efficient simulation technology that is provided by a tool-set.

At the same time, there is a considerable variety of embedded processors that each serve specific application areas. Building custom development tools for these processors is an error-prone and tedious design process. Simulator designers are confronted with the problem of matching the tool to an abstract model of the processor architecture. Efficient simulation technology cannot be reused with new devices because of the customized solutions for each processor. The simulators available from semiconductor vendors prove that simulation speed is indeed a major issue that suffers from bad implementation of processor models. It is reported that instruction set simulators

achieve only speeds between 2k and 20k instructions per second on a 100 MIPS computer [85]. Measurements show for example that the instruction set simulator for the Texas Instruments C54x DSP runs at a simulation speed of 2200 instructions per second on a 100 MIPS computer. The GSM speech coder/decoder program must be run for three days to process only one minute of speech signal. This is equivalent to spending 68000 instructions of the workstation to simulate only one instruction of the target DSP! Although the C54x may be considered more complex than the CPU of the workstation, the orders of magnitude clearly show that this kind of implementation is certainly not efficient.

Designers of new processors and software for systems-on-chip need a reliable and transparent methodology for the creation of processor models covering software and hardware properties on different abstraction levels and fast simulation technology. This can be achieved with two components:

- Processor models that are formally described using an appropriate machine description language and
- Retargetable and fast simulation tools that are generated based on processor descriptions.

This work reports the machine description LISA (language for instruction-set architectures), its generic processor model and the fast retargetable processor simulator that can be generated from LISA descriptions.

The following chapter gives the motivation for selecting zero-delay processor models for processor architecture exploration, embedded software verification and integration into system-on-chip environments. After the classification of the main abstraction levels in the temporal and spatial domains, three types of models are highlighted that this work will focus on. Instruction-accurate models are used for the functional verification of embedded software and for the development of high-level language compilers. Cycle-accurate or cycle-count accurate models provide measures for the design of embedded processors which involves simulation of design candidates during architecture exploration. Finally, cycle- and phase accurate models are used to satisfy the verification requirements of processor-based systems that include the simulation of accelerators, peripherals, buses, the memory sub-system and custom logic. The discussion of previous and related work to the approach of retargeting the simulator and other software development tools based on a central description will show that the issues of fast and retargetable cycle-accurate simulation have not been solved so far and motivate the development of the language LISA.

Chapter three presents the main contribution of this work and describes the LISA language. After a short overview on the concept of using operations as the basic modeling components, the necessary attributes of these operations and their representation in the LISA language is discussed based on examples that are mostly derived from real processor descriptions. Description examples are also used to illustrate the capability of LISA to cover typical modeling issues such as VLIW processors and instruction word coding idiosyncrasies. A particular focus in the discussion of modeling issues is dedicated to the temporal model abstraction and the provisions in the LISA language to create timed processor models based on a generic pipeline model. The generic model enhances the sequential execution of behavior code specified in C by the notion of time and parallel execution. Operations are partially ordered by assignment to time instances based on a zero-delay model. The complete ordering is achieved through a scheduler that provides the ordering of parallel operations.

The forth chapter introduces the main goal of modeling using the LISA language which is fast processor simulation. A technique used to achieve high model speed for specific DSP processors has been discussed in preceding work [97]. In this thesis the technique is generalized to the class of processor models that can be described using the LISA language and its generic pipeline model. A particular role of generalizing compiled simulation play the compile-time statements in the LISA language which enable a high degree of predictability in the model description and the activation tables that implement the generic model in order to create timed processor models.

Several models of real-world processors have been described in the LISA language and simulators have been successfully generated. The models have been validated using register traces based on various application programs. Chapter five gives an overview of three processor models and discusses the measurement of the execution speed that could be achieved with the generated simulators. The first model is a cycle-accurate description of the DLX processor which is an academic architecture known from the book of Hennessy and Patterson [33]. The bit accurate model of the ADSP21xx DSP of Analog Devices is particularly interesting because it could be compared to a customized compiled simulator from a previous project [69]. The third model has complex architecture implementing two pipelines and VLIW type instruction encoding. The C62x DSP of Texas Instruments is modern processor that covers a wide range of modeling issues and its successful model implementation in LISA can be taken as the most relevant proof of concept in this project.

Chapter 2

Processor Modeling

The complexity of embedded processors compels designers to develop and employ abstract models for the specification, design and verification of the processor hardware and software. The modeling on different abstraction levels is a standard approach to reduce the complexity at the cost of accuracy. Abstract models have several advantages:

- The required modeling effort is low, because all unnecessary details of the architecture are avoided.
- Small changes to the abstract model enable the exploration of a wide range of architecture opportunities.
- Renunciation of details ensures high evaluation efficiency of architecture candidates. Executable models provide a high simulation speed.
- Abstract models tend to be clear and understandable.

During the design process, less abstract models are produced from the higher levels of abstraction. Each level of abstraction represents a view of the target architecture that removes unnecessary details. The conversion from an abstract model to a more accurate model is very critical and the functional equivalence of these models must be proved by either formal verification or simulation. In most cases, the formal verification of processors and processor based embedded systems cannot be carried out due to complexity [32]. Consequently, extensive simulation sessions must be used for verification. For this reason, the duration of simulation runs is critical and high simulation speed is extremely important [85, 35, 62].

Distinct from non-programmable hardware components, there are inherently two views of processors, the view of hardware designers and the view of software designers. Both types of designers are interested in different aspects and different accuracy levels of the same processor architecture, these will now be detailed.

2.1 Accuracy Levels

Processor models can be build at different abstraction levels. Figure 2.1.1 shows a very abstract processor model that illustrates the general structure. It consists of three main components, the instruction processor, the data processor and the external interface unit [53]. The instruction processor interprets instructions from the instruction memory and controls the data processing unit. The data processor modifies and transforms data accordingly. The data is transferred from and to the data memory. The third unit, the external interface unit controls access to external data and instructions. The state of the processor is described by these three units. The instruction processor selects and controls the transition functions that are applied to all units and which drive the processor into a new state.

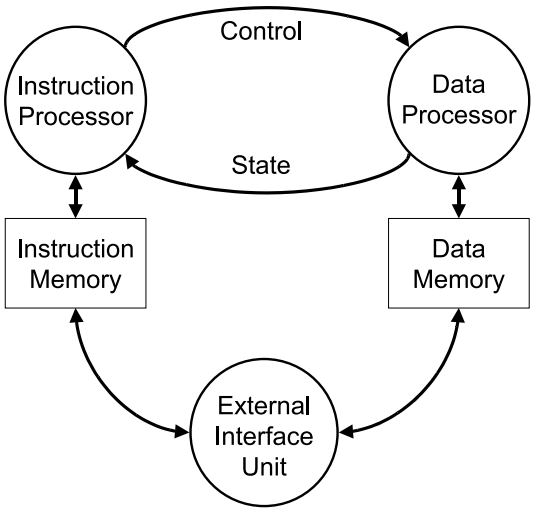

Figure 2.1.1: Abstract processor structure

Processor abstraction levels are mainly characterized by the level of detail of the real hardware that are represented in the model. On the one hand, abstraction can be identified by the smallest hardware structures that are atomic elements in the model, like registers, logic gates and transistors for example, this aspect will be referred to as *spatial* accuracy. Depending on the accuracy level, more or less processor states will be covered in the model. On the other hand, time can be modeled quasi-continuously or based on explicit advancement of time (zero-delay). This characterization will be called *temporal* accuracy. The temporal accuracy determines how frequent state transition functions will be applied to the processor model.

The spatial and temporal accuracy levels are usually correlated, although the time resolution can be varied in a certain range given an abstraction level of architecture. With increasing resolution of time, also a growing number of hardware components and structures become exposed in the model. They are necessary parts of the architecture causing the specific operation timing. The mechanisms of the instruction pipeline for example define the latencies between instruction fetch, decode, execution and write-back.

Whereas hardware designers usually use quasi-continuous models of time, their interest in abstraction levels is mostly focused on spatial accuracy. However, designers of software typically use zero-delay models and the distinction of different levels of temporal accuracy is very important for software design [98]. In the following, the main spatial and temporal abstraction levels

relevant for the design of embedded systems will be presented.

2.1.1 Abstraction of time

The temporal abstraction describes the resolution of time in the model. More accurate models distinguish the order of two events that cannot be distinguished in more abstract models. Figure 2.1.2 illustrates the main temporal accuracy levels that are discussed in the following:

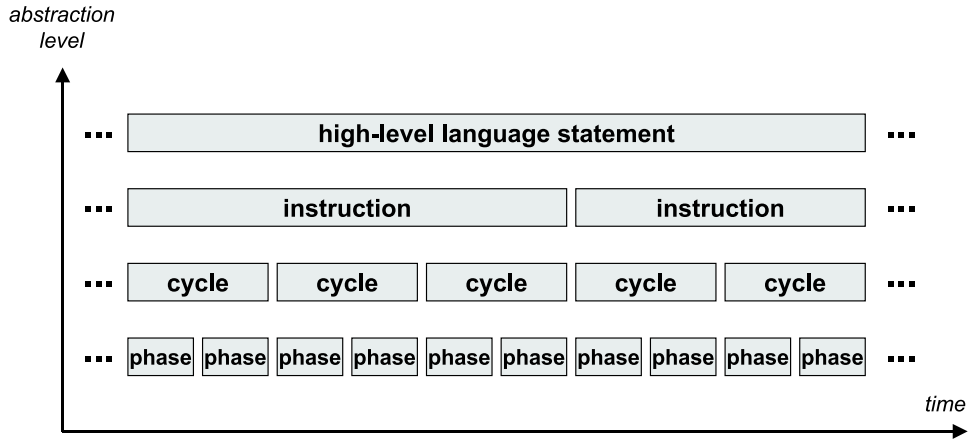

Figure 2.1.2: Temporal accuracy

- **Statement accurate**

  The state of the processor model is reproduced between high-level language (HLL) statements. At early stages of design when the full cycle-accuracy is not required, such models are employed to provide estimates of the execution time based on the executed HLL-statements.

- **Instruction-accurate**

  A processor model at this level of abstraction can be observed after each instruction being executed on the target. These types of models are used for software development and debugging. Executable models at this level are very common and called instruction-set simulators. However, such models are not useful for processors with complex pipelines that are exposed to the programmer due to different types of hazards.

  In both types, statement and instruction accurate models, time is driven forward by the software running on the processor. A very common type of executable models are simulators which are instruction accurate (software-driven) but that reproduce the cycle counts being consumed by instructions. We will call them cycle-count accurate models in distinction to cycle accurate models.

- **Cycle accurate**

  Cycle based models reproduce the state of the processor at the clock cycle boundaries. At this level, the mechanisms of pipelines can be completely described. Since the model is

based on the main processor clock, even interfacing with attached hardware components can be covered as long as transfers are synchronous to the processor clock.

- **Phase accurate**

  In phase accurate models the clock cycles are subdivided into phases. The state of the processor is available at the boundaries between each phase. These phases are typically introduced to resolve conflicts of resources that shall be accessed multiple times per clock cycle (e.g. dual-access memory). Much more important are these models however for the reproduction of non-synchronous interaction with attached hardware components.

- **Quasi continuous**

  The processor state can be reproduced at any time. The resolution is only limited by the implementation of the executable model or the shortest time that can be resolved in the simulator.

2.1.2 Abstraction of architecture

The spatial accuracy describes the detail of the modeled hardware structures and the extend of components integrated in the model. At higher abstraction levels, the models are mainly software-oriented and at lower levels more hardware-oriented.

- **Application program level**

  At this level, the HLL program is modeled bit-accurate based on the processor arithmetic – the functional units in the data paths. HLL programs for digital signal processors are frequently not independent from the target processor. C compilers for many DSP processors provide language extensions to support specific data types, memory space configurations, dedicated register sets and circular addressing.

- **Instruction set architecture level**

  At the architecture level, the programming model of the processor is reproduced. In addition to the data path of the application program level, the control path of the processor is covered on this level as well. This abstraction level can be combined with instruction-, cycle- and phase-accurate temporal accuracy. Depending on the temporal abstraction, details of the instruction and data pipelines are exposed. Furthermore, external interface units of the processor such as DMA controllers, bus arbiters, serial and parallel ports may be covered. Especially if the processor model shall be embedded into a system-level model, the latter components will be modeled.

- **Register transfer level (RTL)**

  The processor state and state update are described based on registers and register transfers. This level is mostly combined with quasi-continuous, phase- or cycle-accurate temporal abstraction. It is a common starting point for the architecture implementation in hardware. For this reason, these models are typically extended to all components of the processor architecture.

- **Gate level**

  The atomic elements at this level are logic gates. Many of these models are synthesized from RTL models (semi-custom design). For this reason they cover the same components as RTL models. However, due to requirements and constraints of the data-paths, design and optimization at this level may be necessary (full-custom design).

- **Transistor level**

  The atomic elements are transistors. All architectural elements of the processor are covered. There are even lower levels of abstraction, however they are not relevant to the scope of this work.

2.2 Application Domains of Hardware/software Models

Especially in the design of general purpose processors (GPPs), a strict distinction between hardware oriented and software oriented processor models has evolved. The hardware-oriented models are used for the design of the processor hardware and other hardware components that are attached to the processor. Software-oriented models are used for architecture exploration, compiler design and application software development hiding the details of timing and the actual microarchitecture implementation. The distinction of pure hardware and software models is not useful for the design of embedded processors. The reason is that software development for embedded systems requires detailed timing information due to real-time constraints. This makes the embedded software design very sensitive to changes of the microarchitecture and thus the timing in the processor. The interaction between the software running on the processor and the connected hardware must be captured in system-level models. The close interdependencies of hardware and software models exist in processor design as well as in the design of processor based systems. Consequently, a separation of hardware-oriented and software-oriented models must be avoided. The main types of joint hardware/software models required in processor- and system-level design are discussed in the following sections.

2.2.1 Processor Design

The advancement in silicon technology and SoC realization has fed the trend to application-specific instruction-set processors (ASIPs). In opposition to off-the-shelf processor cores, ASIPs are designed typically by the system house. There are even partially configurable processor cores available on the market such as the cores from ARC or Tensilica. Tailoring of processors towards a target application means that the processor architecture is gradually refined in an iterative design process as shown in Figure 2.2.1.

Designing a processor involves making design choices between a number of possible microarchitectural and implementation features. Because of the richness of potential design choices and the fact that many design choices are sometines counter-intuitive, it is necessary to model various alternatives and then measure the performance actually obtained before a particular option is chosen [58].

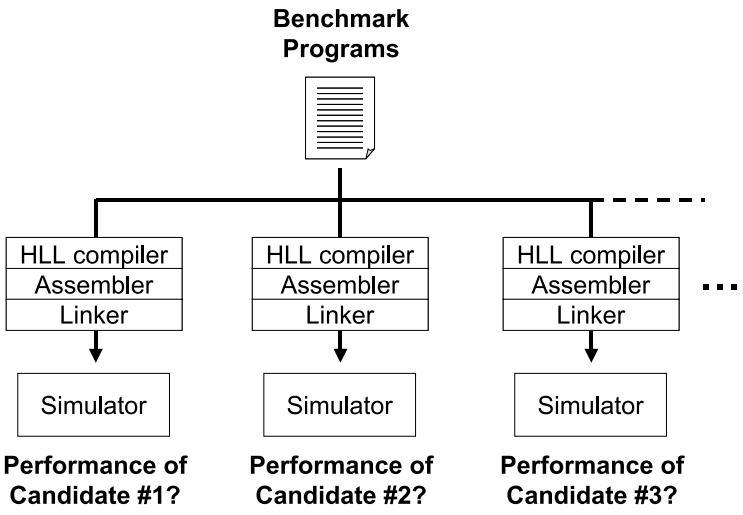

Figure 2.2.1: Architecture exploration

Benchmark application programs are used to evaluate architecture candidates based on the performance measurements and profiling data. The performance models used at this stage of the processor design must be flexible, fast and accurate enough to measure the cycles-per-instructions (cpi) count but also fast enough to quickly provide the performance measures. Since the benchmark program suite is written in a HLL, a complete set of software development tools consisting of HLL compiler, assembler, linker and simulator must be available. In order to enable short design iterations, the software tool set must be adapted quickly to the respective candidate processor.

But processor models involved in processor design are not limited to performance models. The design methodology used in modern microprocessor development projects requires modeling at various levels of abstraction for the different phases of processor design [10]. The *performance models* cover the pipeline mechanisms and instruction semantics. In the end of the architecture definition phase, an instruction-accurate, functional simulator is created that serves as a "golden model reference for functional validation of pre-RTL or RTL implementations.

As soon as the global microarchitecture is frozen, the phase of logic implementation can begin. The design entry, based on a hardware description language (such as VHDL), provides a RTL-level simulation model incorporating full function and timing of the processor. However this model is typically extremely slow (tens to hundreds of cycles per second) and cannot be used for performance studies.

Although the design flow of the microarchitecture implementation is mostly hardware-oriented, the processor models developed in the definition and exploration phase are mixed hardware/software models.

2.2.2 Systems-on-Chip Integration

The complexity and heterogeneity of system-on-chip design are main drivers for the research work on hardware/software co-design. Numerous approaches have been published that solve issues related to hardware/software co-design of system implementation [39, 15].

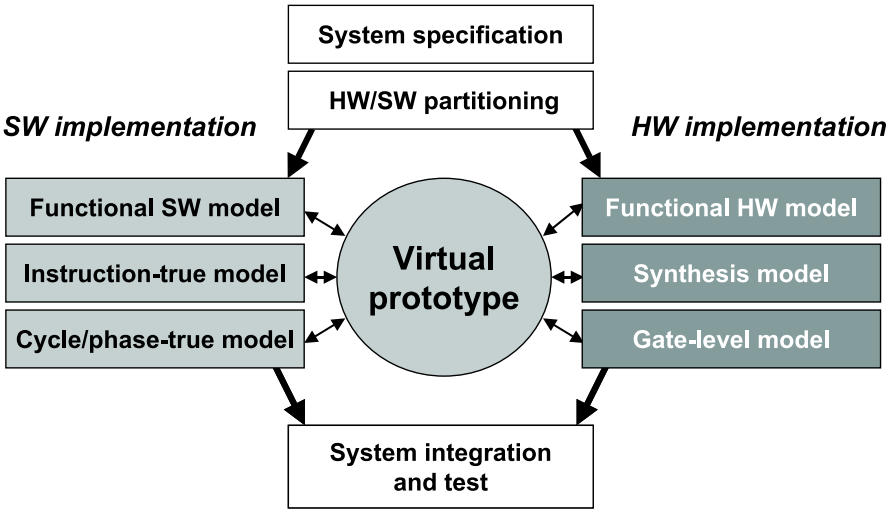

Figure 2.2.2: Hardware/software co-design methodology

Most approaches target components of the design flow illustrated in Figure 2.2.2. Initially, the specified system functionality is partitioned into hardware and software components. In the following sections, the hardware architecture and software implementation are developed in parallel. Since all changes on either side of the implementation path affect the opposite side, a *virtual prototype* [66] based on hardware-software co-simulation tools must be used to evaluate and verify the complete system at each implementation step. The virtual prototype consists of a configuration of models that are refined, thus involving new sets of increasingly detailed models. In the established design flow of non-programmable hardware components, less detailed models can be produced by means of synthesis tools.

Embedded processors however, require models at the different abstraction levels discussed in section 2.1 that cannot be synthesized from each other. Software prototypes that are generated or written in a high-level programming language like C or C++ are translated with HLL-compilers into assembly language of the target processor. The execution speed and the memory consumption of the assembly code of the application software must now be optimized to fulfill the real-time requirements and the limitations of on-chip memory. At this stage of design, instruction-set simulators that accurately determine the executed cycles are needed for software development and debugging. In case of simple processor architectures, instruction-accurate simulators can be employed here, but for more complex processors cycle-accurate or even phase-accurate simulators are required.

Accurate processor models are needed especially as soon as the detailed behavior of the processor and its interfaces, represented by signals and buses must be modeled. For the interaction with hardware components connected to the processor, cycle- and phase-accurate models are necessary [28, 17].

Since simulation speed is critical for the verification of the software in the hardware environment, a common approach is to integrate instruction-set simulators into the system-level simulation environments is based on bus-interface models (BIM) [17]. Bus-interface models are created to embed simulators that are not accurate enough to reproduce cycle- or phase accurate behavior at the interfaces. Typically, BIMs are very complex and modern processor architectures with complex pipelines can hardly be modeled based on this approach. Thus, the whole processor simulator

must be modeled cycle- or phase-accurate.

2.2.3 Processor Simulation

In order to simulate a processor, the abstract model must be translated into an executable processor that implements the simulation of the application program and external stimuli. The execution of the model is performed on a computing platform that simulates the state of the target processor and the application of transition functions. Possible simulation platforms are software simulators, hardware-based emulators and the processor hardware itself. Naturally, the processor hardware is only available after fabrication of the device. In our target applications, processor design and design of systems-on-chip, this is not the case and therefore the processor hardware cannot be used as a simulation platform. Emulators are dedicated hardware components, like e.g. FPGA boards that can be configured to perform the simulation of the target processor or parts hereof. However, emulators are very expensive, hard to configure, and they provide a very centralized and unique computing platform that can be used exclusively by only one user at a time. For this reasons, most designers prefer to use software simulators that are executed on general purpose computers. These simulators reproduce the behavior of the target processor based on a software implementation of the abstract processor model on a host computer (virtual machine). Since embedded processors are parts of systems with limited memory space, it can be assumed in general practice that the host has the required resources to perform this task.

2.3 Retargetable Software Development Tools

Each embedded processor requires a set of processor-specific tools that support the process of software development and verification. These *software development tools* are described in the following section. In section 2.3.2, requirements and methods of generating tools from a processor specification are discussed.

2.3.1 Software Development Tools

In the past, designers have been extensively using assembler language for the programming of embedded software to optimize execution speed and code size. Programming with HLLs was mainly used for prototyping. However, programming on the assembly level is very error-prone and hard to verify resulting in long development times. As the size and complexity of software increases and reusability of software components is becoming very important, HLL programming will become inevitable for embedded software development [61]. The code generation tools typically comprise HLL compilers, assemblers, disassemblers and linkers. The compiler translates the application HLL code into an assembler program (see Figure 2.3.1).

The assembler in turn generates object code from the assembly program. This step is a one-to-one translation of assembly statements to binary encoded instruction words. So far, the binary code is not yet assigned to physical addresses. The linker maps the code objects generated by the assembler into the address space of the target processor and produces the executable program. This

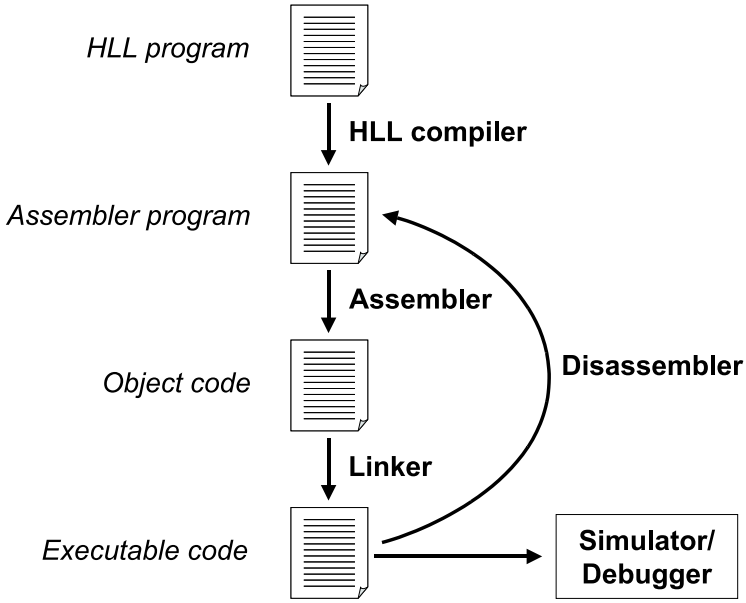

Figure 2.3.1: Software development tools

program can now be loaded and run in the simulator. The debugging interface is a visualization tool that allows the designer to watch and control program execution in the simulator. The state of the processor represented by registers and memories can be observed and changed during the simulation. Frequently, the source code (either the HLL or assembly program) is not available to the designer. In order to watch the program currently being executed in the simulator, the designer wishes to see the assembler code. If it is not available, it must be obtained from the executable program. The reverse process of converting object or executable code into assembly programs is performed by disassemblers.

2.3.2 Retargetability

The tools used for code generation and simulation of embedded processors are highly target specific. The importance of code compatibility is marginal compared to the world of GPPs. Even DSP processors of the same semiconductor vendor have vastly different instruction sets. Therefore, all software development tools must be developed specifically for each processor. However, these tools consist of processor specific and target-independent parts. Building custom tool sets for each new processor architecture is a very error-prone and tedious design process. Compilers, assemblers, linkers and simulators each have to be matched to specific components of an abstract model of the processor architecture. For example, whole simulator designs teams are working over many years to develop and maintain the simulators for modern DSP processors like the C5xx and the C6xx of Texas Instruments. Although simulator design is started early, simulator “bugs” are reported still years after the device is on the market. Furthermore, embedded processors of most semiconductor vendors represent a whole family of devices that feature different peripheral and memory configurations that must be supported by simulators. New system-level simulation tools of EDA vendors require adaptation to their specific interfaces. Therefore, the design of development tools is an ongoing project that does not finish with a stable release. Most of all, designers are faced with the issues of inconsistency between the different development tools, the compiler,

assembler, linker and simulator.

The problem of inconsistency can be solved by using a *retargetable* tool set that is adapted to a target processor based on a single machine description. With respect to simulation we define the following:

> *A retargetable processor simulator is a tool that can simulate different processor architectures by means of a processor template and a machine description that is provided for each target.*

Analogous definitions can be made for compilers, assemblers and linkers. Thus, the processor model is composed of the machine description that is individual for each processor instance and the *generic machine model* that defines common characteristics for the whole class of processor architectures that can be covered. The individual tool set for a processor instance can be implemented based on these two components that are illustrated in Figure 2.3.2.

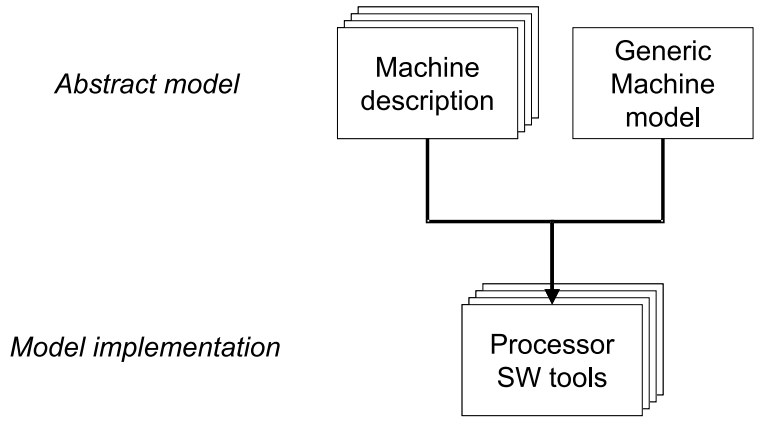

Figure 2.3.2: Retargetable software tools

Depending on the extend of the generic machine model, a trade-off can be made between generality of the processor class and simplicity of the description. The class of processor architectures that can be covered is determined by the properties of the generic model. A generic model based on primitive elements will enable designers to cover a wide range of processor architectures at the cost of long-winded descriptions. Very detailed generic models allow lucid processor descriptions and high modeling efficiency. In the latter extreme, the class of architectures may be limited to derivatives of a template processor [18].

The different software development tools for a specific processor are implementations of the abstract machine model defined by the generic model and the machine description (see Figure 2.3.2). However, each tool implements only a subset of the properties defined in the processor model. These model components will be discussed in the next section.

2.3.3 Model Components of Software Development Tools

Compiler, assembler, linker, disassembler, simulator and debugger are tools that each perform a specific task in the software design flow. For this reason, the required information for the creation

of the respective tool is specific as well. The architectural information can be captured in five model components that are illustrated in Figure 2.3.3. A machine description that is suitable to retarget the usual software tools should provide information consisting of the following model components.

	memory model	*resource model*	*instruction set model*	*behavioral model*	*timing model*
compiler	register allocation	instruction scheduling	instruction selection	-	instruction scheduling
assembler	-	-	instruction translation	-	-
linker	memory allocation	-	-	-	-
simulator	simulation of storage	hazard detection	decoder/ disassembler	operation simulation	operation scheduling
debugger	display configuration	profiling	-	-	-

Figure 2.3.3: Model requirements of software development tools.

- The *memory model* lists the registers and memories of the system with their respective bit widths, ranges, and aliasing. It provides the available registers and memory bank information to the compiler. The memory configuration needed by the linker to determine the mapping of the object code to physical address spaces. During simulation, the entirety of storage elements represents the state of the processor. Finally, the debugger windows dedicated to registers and memories can be configured based on the information of the memory model.

- The *resource model* describes the available hardware resources and the resource requirements of processor operations. Resources reproduce properties of hardware structures which can be accessed exclusively by one operation at a time such as functional units, buses, pipelines, etc. The instruction scheduling phase of the compiler depends on this information. In the simulator the same information can be used to detect pipeline hazards and in the debugger to create profiles of the resource usage.

- The *instruction set model* identifies valid combinations of hardware operations and admissible operands that form the instructions of the processor. The syntax and grammar of the assembly language is defined in this model. Furthermore, the instruction word coding, the legal operands and addressing modes for each instruction are specified here. The translation process of the assembler requires the assignment of assembly statements to binary coding patterns. The same type of information is needed for the reverse translation of the disassembler that is part of the simulator. Finally, the assignment of arithmetical operators to instructions is provided to the compiler, that needs to know the target instructions of the processor in order to select appropriate patterns during code generation.

- The *behavioral model* describes transition functions that represent the abstracted hardware activities for simulation. The transition functions update the processor state at the execution of the simulated program. The accuracy can range between the abstraction levels described in section 2.1.

- The *timing model* specifies the activation sequence of hardware operations and units. The instruction latency information lets the compiler find an appropriate schedule and provides timing relations between operations during simulation.

2.4 Requirements of Hardware/Software Co-Design

The scope of this work is limited to the application domains of hardware/software co-design discussed in section 2.2. A successful implementation of retargetable software development tools for embedded processors based on a machine description must fulfill the following requirements:

- A language must be used for the machine descriptions to achieve the necessary expressive power and flexibility that allows descriptions of new processors and capturing of real-world processor details. It is essential to cover the class of processor architectures that appear in embedded systems ranging from simple single-issue processors to advanced VLIW (very long instruction word) and certain superscalar processors. In particular, the real world (commercial) processors represent the most relevant architectures because they feature idiosyncrasies that are not found in clean designs of pure academic research.
- The software development tools are automatically generated from the machine description in order to enable short design iterations and creation of consistent tools. If we assume the machine description to be correct, then the generated tools are correct by construction.
- The retargetable environment must produce fast simulators. Fast simulation is essential to achieve short evaluation cycles during architecture exploration. At later stages of design, the fast simulation is even more important for verification and validation to process the load of huge test vectors as fast as possible. For this reason, a behavioral description style of the processor model should be supported. A particular technique for the fast simulation of DSP processors has been investigated recently [102]. The compiled simulation helps to significantly improve simulation speed over traditional techniques. For this reason, the machine description should explicitly support compiled simulation techniques.

Beyond these general requirements, there are further requirements concerning the behavioral and timing model on the one hand and the instruction set model on the other hand.

2.4.1 Behavioral and Timing Model

The behavioral model should be described in C or C++. Especially C is well-established in the community of embedded system designers. In general, the acceptance of new languages is low. Therefore, it is advisable to rely on common standards as far as possible. The SystemC initiative for example proposed the use of C++ to bring hardware design to higher levels of abstraction which is a traditionally a domain of designers that are experienced with software. Using C++ also has to important advantage that existing models written in C or C++ can be easily integrated.

The combination of behavioral and timing model must be able to cover several levels of abstraction such as instruction-set models, cycle-based models and phase accurate models. In particular the

structure of instruction and data pipelines and pipeline operations are inevitable components of many processor models. Therefore, capabilities to describe pipeline mechanisms such as stalls and flushes are essential.

2.4.2 Instruction Set Model

The need for different levels of abstraction makes it necessary to distinct between *behavior* and *operator semantics*. In opposition to the behavior that captures activity of the processor hardware with an accuracy that is chosen by the model designer, the operator semantics just specifies the functionality of operators. An example would be an add instruction. This instruction can be described at different levels of accuracy. In very detailed models, the behavioral and bit-true description of the add instruction may consist of large section of C code. However, the operator semantics only provides the operators involved, effectively reducing the long-winded behavioral description into a single algorithmic statement, like $a = b + c$. The latter, high-level type of representation must be described in the model to enable the instruction selection phase of code generators. Although the duplicate representation introduces a certain amount of redundancy into the model, the distinction between behavior and operator semantics is necessary for accurate processor descriptions.

In order to save encoding bits or simplify decoding, many embedded processors feature splitted bit fields or involve particular modes affecting the semantics of certain bit fields. The machine description language should support any type of instruction word coding that is found with real-world (commercial) embedded processors, like instruction aliasing and complex instruction coding schemes.

2.5 Limitations and Assumptions

The approach of the LISA language and the retargetable simulator is targeted to embedded processors. In this work we will particularly focus on DSP processors. Therefore, the class of processors supported by this approach can be limited to the range of architectures that are typically found in embedded systems. A further limitation affects the temporal accuracy. Because it is advantageous to use zero-delay models for fast simulation we will restrict the accuracy to these types of models (cf. section 2.1.2).

2.5.1 Class of Processor Architectures

In general, computer machines can be classified following a popular taxonomy [23, 24] that distinguishes four machine structures with different grades of parallelism:

- Single instruction stream, single data stream (SISD) – This is a uniprocessor.
- Single instruction stream, multiple data streams (SIMD) – The same instruction is executed by multiple processing elements using different data streams. Embedded processor frequently feature particular instructions that execute subdivided operands in parallel. For

example the TigerSharc DSP manufactured by Analog Devices [3] is a 32-bit machine featuring instructions for two 16-bit or four 8-bit operations.

- Multiple instruction streams, single data stream (MISD) – There is currently no commercial implementation of this type.
- Multiple instruction streams, multiple data streams (MIMD) – Each processor fetches its own instructions and operates its own data.

Due to their relevance for embedded systems, only SISD and SIMD type processor architectures are addressed by the LISA approach. The generic processor model of LISA does not allow for more than one instruction stream. However, MIMD processors can be build by combining two or more instances of LISA processor models.

Processors executing a single instruction stream can be categorized into single-issue and multiple-issue processors. Instruction-issue is the process of moving instruction from the decode stage into the execution stage of the pipeline. Therefore, multiple-issue processors execute several instructions in parallel. Since single-issue processors can be seen as a degenerate case of the latter one, only multiple-issue processors shall be considered in the following discussion of the processor class that is covered by LISA. There are two types of processors implementing multiple-issue processors: VLIW and superscalar processors.

VLIWs issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet. The main disadvantage of these architectures is the high resulting program memory size. Recent designs of commercial, embedded processors try to avoid this disadvantage by allowing for configurability [37, 94]. Superscalar processor architectures, in contrast, issue multiple instructions of varying number per cycle. The issue criteria are checked dynamically at run-time, while the corresponding criteria are checked by the compiler in case of VLIW processors. Furthermore, superscalar appear in two flavors:

- In case of the statically scheduled types, the HLL-compiler will compose the order of instruction packets to be executed in parallel.
- The processors with dynamic scheduling determine the composition of instructions at run time of the program based on scoreboarding and appropriate algorithms [95].

Because the number of cycles required for the execution of a block of code cannot be predicted for superscalar processors, there are currently few such processors known in the embedded market that is largely characterized by real time constraints. For this reason, processors with out-of-order execution are not regarded in this work, although most of these architectures can be probably covered.

2.5.2 Zero-Delay Models

Current logic simulation systems capture the abstraction of time based on two main approaches – cycle-based and event-driven simulation [40]. In the comparison, cycle-based simulation achieves higher simulation speed than its event-driven counterpart because scheduling of non-synchronous

events is much more complex [7]. Since high simulation speed is one of the main goals, the concept of the LISA language and its generic machine model is limited to zero-delay models. This means that the state of the processor can only be observed at the end of each clock cycle and intermediate states are not accessible such as in even-driven simulation systems. This limitation does not affect the requirements of hardware/software cosimulation.

2.5.3 Code Generation

The LISA approach is suited to support automatic generation of prototype HLL compiler backends. However, optimization strategies necessary for efficient code generation are extremely difficult to obtain from behavioral processor specifications. Thus, further work on retargetable compilation must solve the present issues of efficient code generation for non-orthogonal processor architectures like DSPs. The efficiency problems of compilers for DSP processors are commonly known even for handwritten and customized commercial compiler products [99, 101, 82]. In recent years, the issues of compiler design has led to a new generation of DSP architectures with much more regularity and orthogonality [94].

A prototype compiler generated with the LISA approach will produce 100% correct code – an important quality that is very valuable for architecture exploration and performance estimation. Furthermore, the prototype compiler can be used as a starting point for a custom compiler implementation with specific optimizations.

2.6 Related Work

In the following, previous work on machine description languages and related areas will be reviewed. Hereby, the survey will focus on the requirements defined in section 2.4. The approaches regarded here can be divided into two main categories: Hardware description languages and machine description languages.

2.6.1 Hardware Description Languages

Hardware description languages (HDLs) like VHDL or Verilog are widely used to model and simulate processors, but mainly with the goal of developing hardware. Using these models for cycle-based or instruction-level processor simulation has a number of disadvantages. They cover a huge amount of hardware implementation details which are not needed for performance evaluation, cycle-based simulation and software verification. Moreover, the description of detailed hardware structures has a significant impact on simulation speed [62, 85]. Another problem is that the extraction of the instruction set is a highly complex, manual task and some instruction set information, like e.g. assembly syntax cannot be obtained from HDL descriptions at all.

An approach of translating event-driven VHDL models into C++ simulators is reported [42], but the achieved simulation speed-up is comparatively low and absolute speed far from the requirements of architecture exploration or verification of application code.

2.6.2 FlexWare

The FlexWare [63] environment comprises the code generator CodeSyn [67] and the simulator Insulin [91]. Retargetability is achieved by a partially reconfigurable VHDL model of a generic processor. The designer can configure the desired number and types of functional units. Additionally, the target instruction set can be defined based on the generic assembler instructions of the VHDL model. The main structure of the processor such as the instruction pipeline cannot be changed by the designer. The advantage of this approach is fast configurability at the cost of a limited class of processor architectures. Furthermore, the reported simulation speed of 500-800 instructions on a Sparc 2 workstation is rather low for application software design.

2.6.3 SystemC

SystemC [64]is a modeling platform consisting of C++ class libraries and a simulation kernel for design at the system-, behavioral- and register-transfer-levels (RTL). It provides semantics of hardware description languages to describe the interfaces and interconnect of components that are specified in C or C++ [88]. SystemC offers the ability to describe both hardware and software in the same high-level language in order to explore potential system partitionings. However, it does not provide any mechanisms to specify and describe processors the software shall be running on. In SystemC models, the specification of the instruction set is completely missing.

2.6.4 MIMOLA

MIMOLA (machine independent microprogramming language) [8] is a hardware description language designed to describe processor architectures and configure a set of related tools, such as processor hardware synthesis, code generation and simulation [54]. The semantics and expressive power is very similar to other hardware description languages such as VHDL or Verilog except some extensions that support retargetable code generation.

MIMOLA models describe the details of the microarchitecture implementation. The instruction set that is required for code generation or simulation must be derived from the implementation structure [46, 47]. The advantage of MIMOLA is to have the same description for hardware implementation, code generation and simulation. However, the low abstraction level makes MIMOLA not very useful for architecture exploration because the microarchitecture implementation is usually not yet defined in an early stage of design and modifications to processor become very laborious.

There are no results published on the simulation efficiency. However, the pure structural approach of MIMOLA is not well suited for the generation of fast simulators.

2.6.5 Machine Description Languages

There are many publications on machine description languages that provide instruction-set models. Most approaches using such models are addressing retargetable code generation. Very commonly used for general purpose processors (GPP) of desktop computers is the GNU compiler [90] that is

retargetable on a functional level. There are many approaches of code generation for embedded processors, e.g. [16, 50, 4, 19, 42]. The following sections provide a review of approaches that address both, retargetable code generation and simulation.

2.6.6 nML

The language nML was developed at TU Berlin [20, 25] to adapt the retargetable system of the code generator CBC [21] and the instruction set simulator Sigh/Sim [51]. A revised version of nML was presented in 1995 [22] and adopted in several projects [31, 27].

It is a well-structured and understandable language based on an attributed instruction grammar. The description of machine instruction consists of three sections:

- **action** specifies the behavior of the instruction in form of register transfers;
- **image** describes the coding;
- **syntax** defines the mnemonics and formal representation of the assembly command.

A hierarchical description style is very well supported. Derivatives of instructions can efficiently be described by the means of or-rules and and-rules provided in `nML`. Or-rules list valid alternatives in order to form non-terminals. And-rules combine already defined objects in order to share certain properties and to produce a new object.

The underlying execution model of nML is rather simple. The machine executes a single thread of uniform machine instructions which are held in the (single) program memory. For this reason processors with variable instruction word size [93] and superscalar architectures [52] cannot be described. Another significant restriction of nML is that the same information (action attribute) is used to identify the functional operators and the description of operation behavior. Consequently, instruction semantics and instruction behavior are merged and described at the same abstraction level. Thus, processors descriptions based on nML are instruction set models. More detailed models including instruction pipelines cannot be covered.

Extensions to nML

Due to the obvious problems with detailed processor models some extensions have been discussed. However, annotation of latencies or reservation tables [25] cannot satisfy the requirements of cycle-accurate simulation of most embedded pipelined processors. Another extension of nML [81] was proposed introducing a resource model. However, the approach of nML is still limited by the underlying instruction sequencer. Processors with more complex execution schemes and instruction-level parallelism like the Texas Instruments TMS320C62x DSP [94] cannot be described, even at the instruction-set level, because of the numerous combinations of parallel and sequential instruction execution within an instruction fetch packet.

2.6.7 ISDL

ISDL [29] is a machine description language targeting the description of ASIPs with emphasis on VLIW architectures. The language is part of a system for architecture exploration that includes a retargetable code generator, assembler, simulator and hardware synthesis. The structure of ISDL is very similar to nML. The main difference is a particular section for the description of explicit *constraints*. In addition to (positively) describing the features of the processor architecture, the constraints are used to restrict admissible operation combinations with the goal of a more intuitive description style. Although this is helpful for code generation, it does not help with the description of resource conflicts that can be detected only at run-time of the simulation.

Due to the concept of latency annotation and an execution model that is very similar to nML, the same limitations apply as well. Like in nML, there is no distinction between behavior and semantics. Thus, cycle-based models of many commercial pipelined DSP processors cannot be created.

2.6.8 EXPRESSION

The EXPRESSION language [30] is designed to support high-level architecture exploration and generation of simulators and code generators. Besides of the specification of the instruction set model, EXPRESSION descriptions mainly focus on the processor structure that is mainly represented by data paths, pipelines, buses and memories. Emphasis of this approach is put on the capability of adapting to architectural changes with only small modifications of the description. But no results are published on the resulting simulation speed.

Although pipelines are supported in EXPRESSION, the bit-accurate operation description including all side-effects is limited by the joint description of behavior and semantics. Furthermore, variable instruction word length that are found in DSP processors [93] are not supported.

2.6.9 RADL

The language RADL [89] is derived from earlier work on LISA [100] and includes two main extensions focused on the description of pipelines. One extension is the introduction of pipeline *phases* that subdivide the control steps (clock cycles) commonly used in LISA. There are no publications that document results of the generated tools or efficiency of this approach.

2.6.10 Maril

The machine description Maril is part of the retargetable code generation system Marion [11]. It focuses especially on load-store RISC processor architectures that do not feature dynamically scheduled superscalarity. The main application domain is retargetable compilation but the authors intended to comprise simulation and synthesis of hardware as well. The Marion system has been used to produce code generators for the Motorola 88000, the MIPS R2000, and the Intel i860.

Maril is based on latency annotation and reservation tables for code generation. To overcome the restriction of fixed latency specification for every instruction, auxiliary directives are used to override the latency specification for specific instruction pairs. So, the pipeline hazards are described explicitly. The disadvantage of this approach is that all hazards for all possible instruction pairs have to be specified explicitly. For highly pipelined processors this can turn out to be a major problem because even after the design of a processor some hazards may be discovered which occur in very special circumstances.

Since operation behavior in Maril is specified by single assignment expressions, machine instructions can produce only one result. This is an restriction excluding SIMD processors.

2.6.11 ISPS

The formalism of the instruction set processor specification (ISPS) was published by M. Barbacci in 1981 [6]. It is based on the language ISP which was published by C. Bell in 1971 [9] and includes some improvements to allow the description of local units and processes. The application domain comprises the simulation and synthesis of hardware as well as retargetable compiler and assembler. The ISPS description is parsed and feeds a global data base that all related tools are based on.

ISPS descriptions allow the specification of behavior elements (*entities*) as well as their interconnect (*interface*). The behavioral part is based on register transfers descriptions. Basically, the proposed formalism used to describe the behavioral part of the model does mainly provide features which also can be found in programming languages like C. Additionally, there are language features that allow the synchronization of operations and further control mechanisms. Consecutive steps of operations are separated by the *next* statement which refers to the basic step of the processor. This simple mechanism makes it very hard for the designer to model complex pipelines.

ISPS has a limited capability of describing instruction word coding. Embedded processors tend to have fragmented coding fields that cannot be described with ISPS. Furthermore, due to the uncommon syntax for the description of behavior, ISPS cannot be used directly in combination with standard compilers of programming languages.

2.6.12 SimpleScalar

The tool set of the SimpleScalar architecture provides five fast simulators with different accuracy levels [13]. But the retargetability of this tool set is restricted to derivatives of the MIPS architecture.

2.6.13 SimOS

In the SimOS project [84, 96] processor models on three different abstraction levels are used to simulate complete operating systems. However, the strategy of abstraction by direct execution

requires binary code compatibility between host and target which does not apply to software development for embedded processors.

2.6.14 Summary

The existing description languages for the modeling of embedded processors can be divided into software-oriented and hardware oriented languages. Most software-oriented languages provide an instruction-set centric view of the processor that only superficially describes architectural details based on latency annotation. The hardware models are typically structure oriented and capture a lot of hardware implementation details that are not needed and significantly slow down simulation. Furthermore, the instruction set model is completely missing in HDL models. The approach of the LISA language is to bridge this gap with a joint hardware/software model. Finally, our interest in compiled processor simulation is addressed by the introduction of explicit statements that support compiled techniques.

2.6.15 Why a New Machine Description Language?

In general, only few new languages seem to be accepted by designers. A main motivation for the acceptance of new approaches is the lack of (better) alternatives. As we saw from the review of existing approaches, there is no machine description language that fulfills our requirements of hardware/software cosimulation. Therefore, a new approach should be based as much as possible on existing methodology. From our experience, many designers are already using C for modeling processors and writing simulators. For that reason, we chose C (resp. C++) for the behavioral model and the memory model. Nevertheless, C does not provide a formal way to describe the timing model, the assembly syntax and grammar or the instruction word coding. Consequently, the LISA language is a framework with embedded components of C code. The basic idea of attributes in nML was implemented in LISA to combine the formal description of timing, syntax and coding with the C code components.

Chapter 3

LISA – Machine Description Language and Generic Model

3.1 Overview

LISA (language for instruction set architectures) is a machine description language that is designed to support the generation of software development tools for processor architecture exploration, application software development and system-level integration of processor cores [100, 74, 71]. The framework of the LISA language lets designers create formal processor descriptions that are easy to understand and well maintainable. LISA processor models provide all information necessary for the automatic generation of software development tools such as simulators, debugger interfaces, assemblers, disassemblers and linkers. A consistent set of these tools can be generated from a single processor model specified in LISA as shown in Figure 3.1.1.

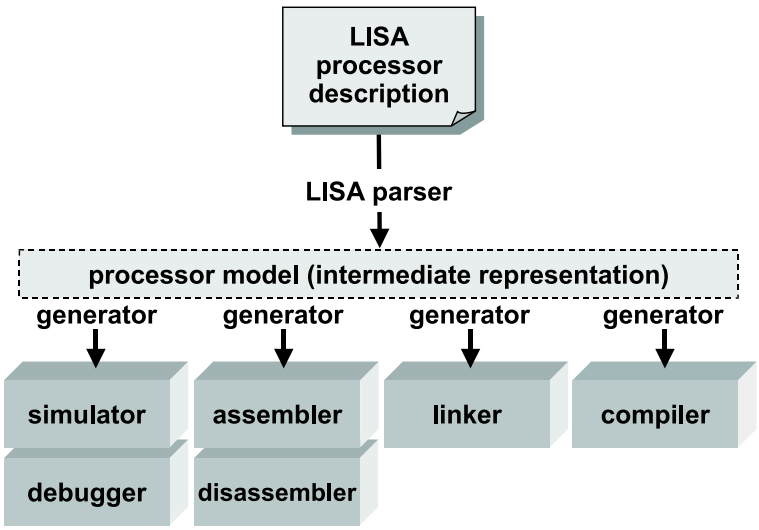

Figure 3.1.1: Development tools generated from a LISA processor specification.

Hierarchical modeling style is supported to allow good structuring and easy maintenance of the code. Due to its C-like syntax, LISA can be easily and intuitively used by designers. LISA descriptions are non-ambiguous specifications of the target architecture that can be exchanged be-

tween design groups (e.g. product definition group, processor hardware designers and application software developers). Such specification is a valuable replacement for the textual documentation written by designers which is usually faulty and not up-to-date. The specification in LISA is the basis for generation of documentation formatted in HTML for online use or formatted for word processing tools. Furthermore, the generated simulator itself can serve as an executable specification [87]. The advantage of such a generic approach is obvious: the documentation is correct by construction and consistent with all tools generated from the same processor description.

3.1.1 Abstraction Levels

The LISA language combines two views of the processor: The programming model of the software designer (software view) and an abstract model of the processor hardware architecture (hardware view). The design of the LISA language was influenced by the view of a software designer who needs to add hardware properties and timing to the model in order to provide the required accuracy (see Figure 3.1.2).

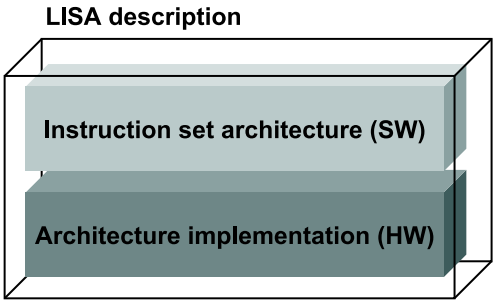

Figure 3.1.2: Joint HW/SW model

Similar to using programming languages, the user has a high degree of freedom to describe his view of the architecture and to create a model at the desired abstraction level. For example, one instruction of a processor may be represented by just one operation in case of an instruction set model (see Figure 3.1.3).

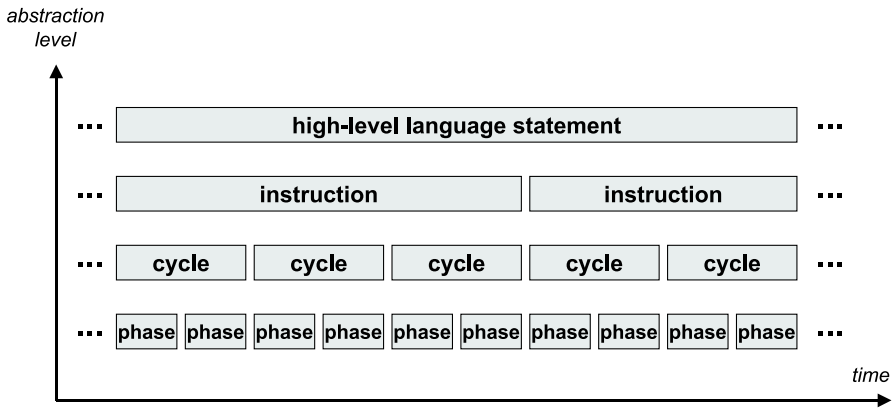

Figure 3.1.3: Abstraction levels of LISA processor descriptions

After refinement of the model, it may be described by a whole sequence of operations that represent the separate actions between clock cycles in case of a phase-accurate model. Common to all models

described in LISA is the underlying zero-delay model. This means that all transitions are provided correctly at each control step. Control steps may be clock phases, clock cycles or instruction cycles as illustrated in Figure 3.1.3. Events between these control steps are not regarded. However, this property meets the requirements of current cosimulation environments such as Seamless of Mentor Graphics [56], Eaglei of Synopsys [92] or VCC of Cadence [14] on processor simulators [28, 17].

3.2 Language Structure

The basic structure of the LISA language is adapted from the information found in conventional text-book descriptions of processors which are frequently called "Programmer's Manual". Programmer's Manuals describe:

- the storage elements – the state of the processor and
- the bit-true behavior of instructions – the transition functions between two states of the processor.

Thus, LISA descriptions are composed of *resource declarations* and *operations*. The declared resources represent on the one hand the storage objects (e.g. registers, memories, pipelines) that capture the state of the processor and on the other hand hardware resources (e.g. functional units, buses) that model the limited availability for operation access. Section 3.3 provides the detailed description of resource declarations.

Operations are the basic objects in LISA. They represent the designer's view of data paths, the instruction set and parts of the underlying architecture. Operation descriptions collect several attributes that contribute to the model. The operations are typically structured hierarchically in a tree. Operations in nodes of the tree describe properties that are common to all operations in inferior nodes or leaves. The complete description of the LISA language is provided in [73].

3.3 Resources

3.3.1 Memory and Resource Model

The resource section lists the definitions of all objects which are required to build the *memory model* and the *resource model.* Besides the data storing elements like memories, registers, and flags on the one hand also signals, buses, and virtual resources can be declared. The totality of resources holds the state of the processor. Object definitions in the resource section contribute to both models, automatically receiving the properties of a memory element and a resource element. Thus the object is represented by a two-tuple (v, s): The data values v based on the data type that is part of the object declaration and the binary state s of the resource that indicates availability or occupation by some operation. It depends on the operations (the behavioral model) if both or only one of these properties are used. Whereas the data values parameterize the transition function

that drives the processor into a new state, the occupation state of the resources determines the admissible operations and the selection of the transition function itself.

The resource section lists all declarations of resources. The declarations follow the style of variable declaration in C and data values of resources are treated like variables in C. However, they can only be declared outside the scope of operations. Consequently, all resources are defined globally and visible for all operations. This resembles the properties of hardware components that are global by their nature.

The resource section comprises four types of objects:

- Simple resources, such as single registers and flags as well as vectors hereof such as register files and memories
- Pipelines structures for instructions and data paths
- Pipeline registers that resemble the data stored in latches between each pipeline stage
- Memory maps that locate resources in the address space

3.3.2 Simple Resources

Simple resources are the storage elements of the memory model that directly correspond to variables in a programming language. Resource declarations can be supplemented with *resource categories.* The specification of these categories is not mandatory but frequently used to assign resources to one of the following categories:

- *REGISTER*,
- *CONTROL_REGISTER*,
- *PROGRAM_COUNTER*,
- *DATA_MEMORY* or
- *PROGRAM_MEMORY*.
- *PORT*.

With respect to simulation, users of graphical debugger interfaces expect the registers, control registers, memories, etc. to be logically arranged. Resource classifications in LISA are responsible for the configuration of the debugger interface and placement of resources into appropriate subwindows. Beyond that, the resource attribute *PROGRAM_COUNTER* has a particular meaning. It identifies the resource carrying the address of the program line to be highlighted in the debugger front-end and is essential for breakpoint management. The resource section of a very simple processor model is provided in example 3.3.1.

```
RESOURCE {
  PROGRAM_COUNTER  int pc;
  CONTROL_REGISTER int ir;                  // instruction register
  REGISTER         mreg[16];                // register file
  PROGRAM_MEMORY   int prog_mem[0x3FFFFF];
  DATA_MEMORY      int data_mem[0xFFFFFF];
}
```

Example 3.3.1: Resource declaration of a simple processor model.

3.3.3 Pipeline Structures

The LISA language provides designated mechanisms to model multiple instruction and data pipelines of a processor architecture. The generic machine model of LISA allows that operations are explicitly assigned to pipeline stages or pipeline phases. Thus, the respective pipelines must be declared in the resource section. Pipeline declaration starts with the keyword *PIPELINE*, followed by an identifying name and the list of stages, phases, sub-phases, etc. Example 3.3.2 depicts three pipeline declarations.

```
RESOURCE {
  PIPELINE pipe = { FE; DC; EX; WB };
  PIPELINE detailed_pipe = { FE; DC; EX; WB }{ read; write };
  PIPELINE detailed_pipe2 = { FE; DC; EX; WB }{ read; write }{ ph1; ph2 };
}
```

Example 3.3.2: Pipeline declaration.

The pipeline *pipe* consists of four stages. The stage names are separated by semicolons. The stages are ordered with the first stage first. The second pipeline *detailed_pipe* has the same stages, however it provides a more detailed description by subdividing the stages into two phases *read* and *write*. Subdivisions have the same syntax as stage definitions. Even further subdivisions can be added to arbitrary refine the model. The pipeline *detailed_pipe2* gives an example for such a declaration.

3.3.4 Pipeline Registers

The purpose of processor pipelines is to improve the instruction throughput. The execution of instructions is split into several parts. Each pipeline stage performs a part of the complete execution. Instructions enter the pipeline at one end, progress through the pipeline and exit at the other end [33]. Thus, the pipeline must remember the different instructions in the pipeline, each in a different stage of completion. In the pipeline the instructions are held in pipeline registers or latches that separate the pipeline stages and store the context of instructions in the pipeline. Figure 3.3.1 depicts these registers of a four stage pipeline. The stages between the pipeline registers represent the actual logic that performs the partial execution of instructions.

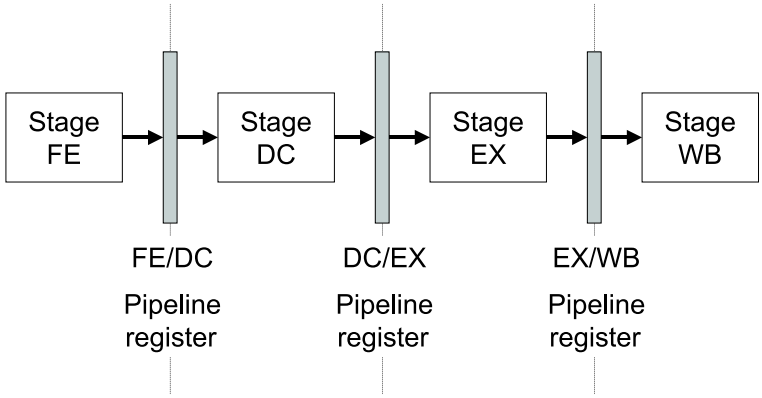

Figure 3.3.1: Pipeline registers/latches

LISA provides the resource type of *PIPELINE_REGISTER* declarations to resemble pipeline registers. Pipeline registers must be assigned to a pipeline of the processor model thus producing multiple instances of the register. This is the main difference to simple resources that are instantiated only once. Example 3.3.3 shows the declaration of the pipeline *pipe* and several pipeline registers that are assigned to it.

```
RESOURCE {
  PIPELINE pipe = { FE; DC; EX; WB };
  PIPELINE_REGISTER IN pipe
  {
    int instruction_register;
    int src1, src2, dest;       // source and destination operands
  };
}
```

Example 3.3.3: Declaration of pipeline registers.

The pipeline registers can be accessed in the C code of the behavior section just like all other resources. However the specific syntax for pipeline register accesses must be used which starts with the keyword *PIPELINE_REGISTER* followed by the name of the pipeline register in parenthesis, a dot and the name of the resource. The pipeline register name is composed of the pipeline name and the identifiers of the two stages that are separated by this register, as for example:

```
PIPELINE_REGISTER(pipe.DC/EX).src1
```

The pipeline registers are inserted between all pipeline stages as illustrated in Figure 3.3.1. Thus, only one register is produced for the first and the last stage of the pipeline. All other stages have a pipeline register to read from and to write to. As the pipeline is advanced, the data stored in the pipeline registers is forwarded as well. All other operations that affect the pipeline or stages of it, also apply to the pipeline registers.

The pipeline registers are not affected by phase definitions. The phases split the pipeline stages into several parts. However, register instances are only inserted between stage boundaries. Figure 3.3.2 illustrates the pipeline of example 3.3.2 that is subdivided into read and write phases. If data shall be transferred between two operations that are assigned to the same pipeline stage but to

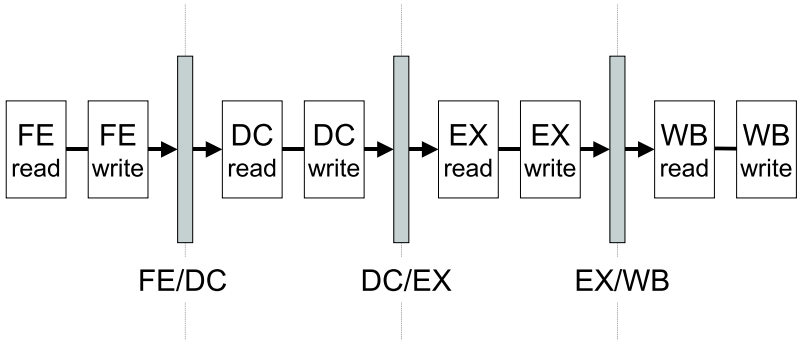

Figure 3.3.2: Pipeline registers/latches with phases

different phases, the designer has to declare and use his own resources that store the data.

3.3.5 Memory Maps

Modern DSPs and Microcontrollers may have huge address spaces. 32-bit processors are very common today and those may address up to 4 GByte of memory. Typically, only small portions of the space actually address physical memory. Obviously, it is efficient with respect to simulation size to model only the respective portions of the address space. The memory maps in LISA describe the mapping of resources in the address space.

```
MEMORY_MAP uaddress BYTES(1)
{
  0x00000000..0x00FFFFFF  =>  prog_mem, BYTES(4);     // 32-bit data
  0x01000000..0x13FFFFFF;                             // -empty-
  0x01400000..014000FFFF  =>  prog_rom, BYTES(1);     // 8-bit data
  0x80000000..0x82FFFFFF  =>  data_mem, BYTES(4);     // 32-bit data
  0xA0000000..0xA0000000  =>  mreg,     BYTES(4);     // memory-mapped
}                                                     //     registers
```

Example 3.3.4: Memory mapping of resources.

Example 3.3.4 shows the declaration of a sample memory map that locates the three resource vectors *prog_mem*, *prog_rom* and *data_mem* in the address space named *uaddress*. The resource *mreg* represents memory-mapped registers. The address space itself is organized byte-wise (*BYTES(1)*). Thus, addresses 0x0 to 0x3 are mapped onto *prog_mem[0x0]* that features 32-bit wide memory (*BYTES(4)*).

3.3.6 Issues with Real Processors

Commercial processors show idiosyncrasies and properties that demand particular consideration. Some common issues that concern resource declarations are discussed in this section.

Special Data Types

The host platform frequently provides arithmetic and native data types that differ from the target and exceed the maximum bit width supported on the host. Especially long registers and accumulators are frequently found in DSPs. In order to cover these particular registers, LISA provides a particular data type in addition to standard C/C++ data types. The scalable fixed point data type *bit* can be sized to any number of bits and provides signed and unsigned arithmetic. For example, the declaration of a 75 bit wide register has the following form:

```
REGISTER bit[75] accu;
```

Aliasing

Frequently, DSPs feature registers which are combined of multiple bit-fields (resp. sub-registers). These bit-fields can be accessed independently as well as a whole. In order to simplify these different types of accesses, LISA allows the definition of aliases to reference these bit-fields. Aliases do not define new states. They define pointers to resources that are defined in the model.

```
RESOURCE {
  int accu[0..1];                  //  declaration of a combined register
  int lo_word ALIAS accu[0];       //  alias for lsb
  int hi_word ALIAS accu[1];       //  alias for msb
}
```

Example 3.3.5: Definition of aliases.

Example 3.3.5 defines the resource *accu* as an array of two elements. The aliases *lo_word* and *hi_word* each refer to these components. Write accesses to one of the aliases leaves the value of the other element intact.

3.4 Operations

Operations are formed by a header line and the operation body. The header line consists of the keyword OPERATION, its identifier and possible options:

```
OPERATION name_of_operation [options]
{
  section list...
}
```

Operation definitions collect the description of several attributes capturing model components that have been discussed in section 2.3.3. These attributes are defined in the following sections:

- The DECLARE section contains local declarations of identifiers, operation groups and references to other operations used in the context of the current operation.
- The CODING section describes the binary image of the instruction word (instruction set model).
- The SYNTAX section captures the notation of mnemonics and other syntax components of the assembly language, such as operands, and execution modes (instruction set model).
- In the SEMANTICS section, the functional operators combining source and destination operands are specified based on simple register transfers (instruction set model).
- The BEHAVIOR and EXPRESSION sections describe components of the behavioral model. During simulation, the operation behavior is executed and modifies the values of resources which drives the system into a new state.
- In the ACTIVATION section, the timing of other operations is defined relative to the current operation (timing model).

Beyond these sections, that are predefined in LISA, the designer may add further sections in order to describe other attributes, like e.g. power consumption or documentation.

3.5 Local Declarations

The DECLARE section provides declaration of identifiers with a scope local to the current operation. The items of the declare section are used in the later sections (e.g. coding, syntax, behavior). There are four types of declarations:

- Instances,
- groups,
- references and
- labels.

The purpose of these declarations is to declare relations between operations within the operation tree. Figure 3.5.1 shows parts of such a tree for a small processor. Starting with operations *main* and *decode* the description is distributed into inferior operations. The operations at branches of the tree reference alternative operations. For example, the *opcode* lists the alternatives *control*, *arithm* and *move*.

Instances

Instance declarations announce direct references to inferior operations. An instance produces a single branch originating from the current operation and pointing to the inferior operation. Regarding the operation *main* in the tree of Figure 3.5.1, the operation *decode* in an inferior operation that must be declared as an instance.

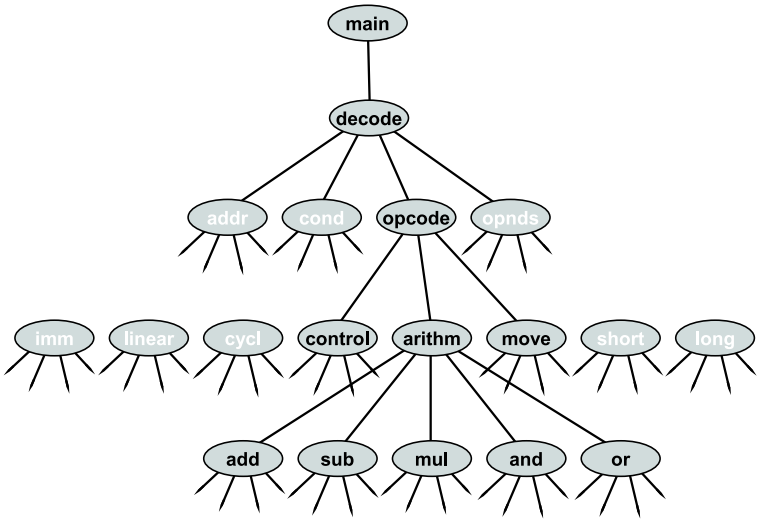

Figure 3.5.1: Operation tree

Groups

The purpose of operation groups is to list alternative operations that are used in the same context. The members of a group represent possible options that cannot be selected at model entry time of the LISA specification. The selection is rather made at compile-time, based on the individual parameters of the current instruction. Hence, the selection is done based on either the specific instruction coding or the assembly statement, depending on the direction of translation. Therefore, groups represent non-terminals in the scope of the operation holding the group definition. In the operation tree, groups produce branches to inferior operations. The operations *decode*, *opcode* and *arithm* in Figure 3.5.1 contain such group declarations. The basic concept of groups in LISA is similar to the or-rules in nML. The differences between both approaches are discussed in section 3.5.1.

References

In many cases, the hierarchical structuring of operations makes it necessary to provide access to groups or instances that are not defined in the scope of the current operation. LISA allows to reference groups that are declared in superior operations. Furthermore, references can be passed on from one operation to the next. Distant groups can be reached by concatenating references along an operation chain. Figure 3.5.2 illustrates an example based on a subset of operations that form a LISA processor description. Operations are shown in ellipses. Elements of groups are connected by lines to the operation holding the group declaration. The group elements that are actually selected by a sample instruction are shown with solid lines, the other elements have dashed lines.

In our example, we regard the small set of three operations that belong to one group of operation *alu*. The instruction *add* can be executed in two different functional units, either *unit1* or *unit2* which form the second group of operation *alu*. We assume, that the actual instruction that we regard here selects operation *add* from the first group and *unit2* from the second. In the scope of *add*, the group comprising *unit1* and *unit2* is unknown. Operation *add* must have a REFERENCE

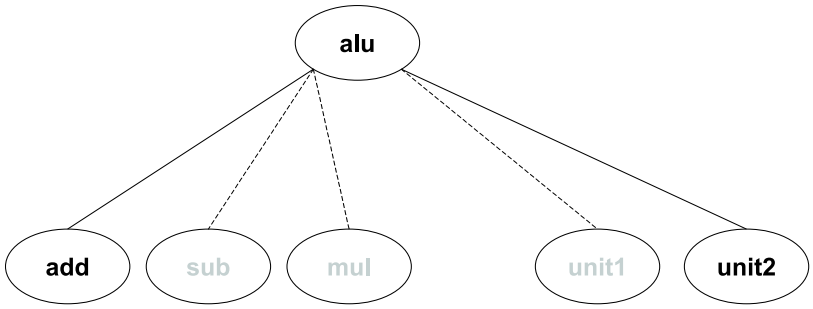

Figure 3.5.2: References

that imports the group definition from operation *alu* to make it available in the local scope.

Labels

Operation sections may have more than one terminal element. Labels are used in LISA to create cross references between terminal elements of different sections. Example 3.5.1 shows the use of the label *immediate* in operation *addc*.

```
OPERATION addc {
  DECLARE {
    LABEL immediate;
    GROUP dest, src1 = register;
  }
  CODING   { dest  src1  immediate=0bx[14] }
  SYNTAX   { "ADDC"  dest  ","  src1  ","  immediate=(int) }
  BEHAVIOR { dest = src1 + immediate; }
}
```

Example 3.5.1: Section cross reference with labels.

The label identifies those elements in the coding, syntax and behavior section that are directly correspond to each other.

3.5.1 Modeling Issues

The language nML and its derivatives like ISDL demand that the topology of the operation tree is identical for all attributes (coding, syntax, behavior). This mechanism is caused by the language concept. The or-rules of nML apply to the level of operations. Therefore, all attributes of the operations produce the same tree topology. The combination of several attributes in a single operation is very useful in many cases and allows the description of clearly structured instruction sets, such as the Analog Devices ADSP21xx DSP. However, the restriction turns out to be not suitable for complex processor architectures like the C62xx DSP of Texas Instruments.

A simplified example taken from the C62xx model shall illustrate the restriction. This DSP has two registers banks and two sets of functional units that are nearly identical. Therefore, many

instructions can execute in the units on both sides. The respective unit is determined at compile time and coded in the instruction word. An assembly statement like “ADD.L1 dest, src1, src2” will produce an add instruction that is executed in the L-unit (the ALU) number one using the operands dest, src1 and src2. In a suitable description, the operation *add* is the selected element of a group in *alu* (see Figure 3.5.3).

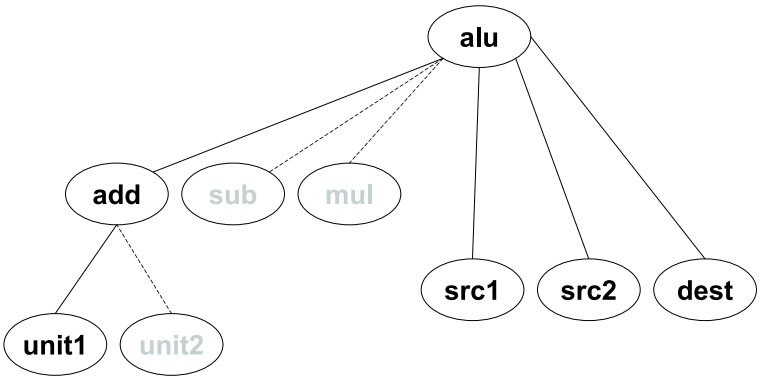

Figure 3.5.3: Operation tree for syntax attributes

Since the operation *add* can be executed in both units, those become elements of a group declaration in this operation. Since the other instructions (*sub* and *mul*) use the same operands, *dest*, *src1* and *src2* are groups of *alu*. Figure 3.5.3 shows the resulting tree for the syntax attributes. The syntax of the assembler language binds the units to the operation *add*. A different tree arises from the description of the instruction word coding. In the coding, there is no distinction between add instructions that are executed on either set of units. However, the selected unit decides which registers bank has to be selected. The resulting tree for the coding is shown in Figure 3.5.4. Because the units shall be visible for all other instructions, such as *sub* or *mul*, *unit1* cannot become an inferior operation of *add*.

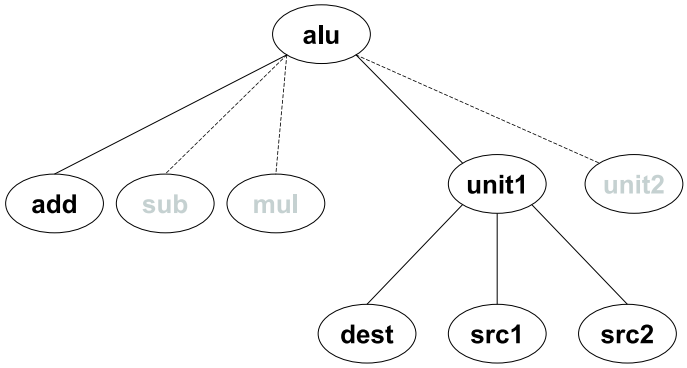

Figure 3.5.4: Operation tree for coding attributes

In this rather simple example, it is still possible to find ways of descriptions that result in the same tree topology for all attributes. However, only at the cost of repetitive description of the same structures in several places. Furthermore, the full model of the C62xx contains additional complexity that hardly can be described with nML.

3.6 Behavioral Model

In the instruction set model, instruction word coding as well as assembler statements define configurations of *operations*, *operands* and *execution modes.* The corresponding elements in the behavioral model are defined in two sections:

- The BEHAVIOR section describes the operations.
- The EXPRESSION section defines the operands and execution modes used in the context of operations.

3.6.1 Behavior Section

The behavior section describes the behavior of operations based on the programming language C. The code of each operation is transformed into a C function body. Combined with an appropriate scheduling that is defined in the timing model, these functions build the architecture specific simulator.

Calls to other operations have the same syntax as calls to C functions. The only difference is the limitation that no parameters can be passed to LISA operations. The reason for this limitation is that the resources like registers and memories are globally visible to all operations and therefore, parameters are not necessary. A further reason comes from the implementation of the compiled simulation technique. Allowing individual parameter lists for the behavior sections of different operations would spoil the efficiency of compiled simulation. A detailed discussion of this issue is provided in section 4.4.1. This restriction does not apply if an ordinary C function is called instead of a LISA operation. Besides the parameters, the main difference between C function calls and LISA operations are the mechanisms of groups and references. Calls of the behavior code of other operations can use the identifiers of groups or references. The actual group element will be selected at compile-time. The same mechanism is provided for operands. The use of groups and references chooses the operands (like registers, memories, etc.) or calculates the actual values of immediate operands at compile-time.

3.6.2 Access Attributes

Implementations of most processors allow reading and writing of registers in the same cycle. The execution order of two LISA operations accessing the same register is undefined if both operations are assigned to the same pipeline stage. In this case, the designer should use the access attributes that specify the “ports” of behavior sections. Ports can be attributed as input (*IN*), output (*OUT*) or bidirectional (*INOUT*) in the port specification as shown in example 3.6.1.

It is not required to specify the ports of all operations in LISA but in this case the processor model would provide all information that is required to assemble a complete netlist of the interconnect between the operations.

```
OPERATION ADDER {

  BEHAVIOR ( IN src1, src2;  OUT dest;  INOUT status; )
  {
    dest = src1 + src2;
    status |= ((src1 > 0x7FFF) && (src2 > 0x7FFF));
  }

}
```

Example 3.6.1: Specification of inputs and outputs.

3.6.3 Expressions

The EXPRESSION section defines an expression object that returns either a resource object or a numeric expression. The objects in the expression section provide operands to the behavior section of an operation. In the case of an add instruction that reads the source operands from a register file and writes the result back, the behavior code of this operation would be placed in the behavior section of some operation and the registers would be placed in the expression section of some other operation.

3.6.4 Reset

The operation *reset* is a particular operation that is executed at reset of the processor. Its purpose is to initialize all resources and to start all operations that have to be performed during reset.

3.7 Timing Model

Most DSP processors and microcontrollers have pipelined architectures. In cycle- and phase-accurate machine models, the effects of pipelines in the processor architecture become visible and must be described. There are even many processors that expose the pipeline at the instruction-set level. To describe pipelines and operation timing, the LISA language uses a generic machine model.

3.7.1 Generic Machine Model

A generic machine provides common characteristics of the designated processor class. The generic model of LISA is based on control steps that can be interpreted by the designer as HLL instructions, cycles or phases. The granularity of control steps is determined by the designer based on:

- pipeline declarations,

- operation assignment to pipelines,
- activation of operations and
- explicit advancement of time.

3.7.2 Declaration of Pipeline Structures

The declaration of pipelines (see chapter 3.3.3) is the basis for cycle- and phase-accurate processor models in LISA. Under the condition that the generic processor model shall be used and the processor's instruction cycles do not generally coincide with its clock cycles, a LISA description without pipeline declarations cannot be more accurate than the instruction-set level. Nevertheless, it is possible to describe a cycle-count accurate model without pipelines.

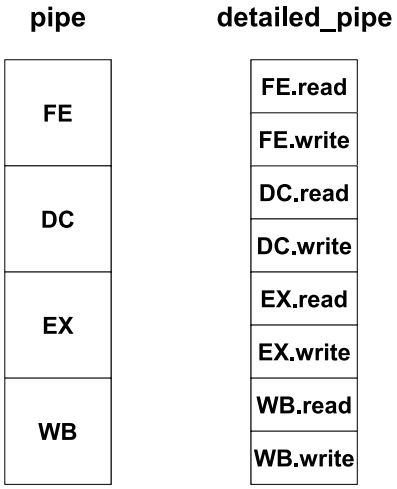

Figure 3.7.1: Granularity of pipelines

Based on the generic machine model of LISA, pipelines can be declared with the granularity of stages, phases and sub-phases. Figure 3.7.1 shows the granularity of two pipelines. Pipeline *pipe* consists of four stages and *detailed_pipe* provides additionally two phases per stage. The highest possible temporal accuracy of a processor model is defined by the finest pipeline granularity that appears in the description. However, the actual accuracy of the model also depends on the assignment of operations to pipeline stages or phases and on appropriate operation activation.

3.7.3 Operations in the Pipeline

In LISA processor models, instructions are composed of operations. In case of cycle- or phase-accurate models, some of the operations may be assigned to pipeline stages and other not as depicted in Figure 3.7.2. Here, the instruction is composed of seven operations O_1 to O_7. Except of O_4, all operations are assigned to the stages s to $s + 2$ of the pipeline $pipe$.

The lines represent activations that connect activating operation with activated operations. All operations that are assigned to the same stage also execute in the same control step independent from control step or pipeline stage of the activating operation. E.g. operations O_5, O_6 and O_7

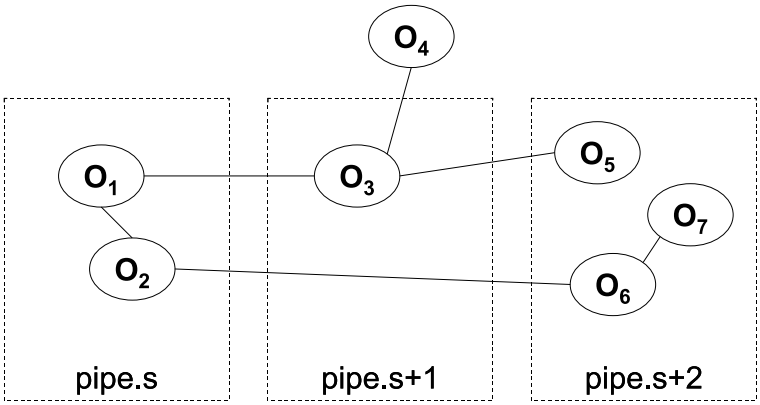

Figure 3.7.2: Instruction composed of operations

execute in the same control step, although O_5 is activated two pipeline stages earlier than O_7. Operations that are not assigned to a pipeline stage execute in the control step of the activation, i.e. O_3 and O_4 execute in the same control step.

3.7.4 Operation Assignment to Pipelines

The assignment of operations to a pipeline stage or phase is specified in the header line of the respective operation description by appending the keyword *IN* and the identifier of the pipeline stage. The LISA syntax to assign the operation *decode* to the stage *DC* of pipeline *pipe* is as follows:

```
OPERATION decoder IN pipe.DC {...}
```

Assigned operations will always execute in the stage or phase they are assigned to. If multiple operations are assigned to the same pipeline stage or phase, there is no precedence ordering between these two operations, because these are operations that execute in parallel like two independent hardware units. In the simulation, there will be a schedule that orders the execution of the two operations, but it is not predictable which operation will execute first. Figure 3.7.3 depicts the assignment of operations to the two pipelines of the previous section.

In pipeline *pipe*, the two operations *address* and *read* are assigned to stage FE. As long as these operations are assigned to the same stage in a cycle-accurate model, no execution ordering is possible. The ordering can be achieved with the pipeline *detailed_pipe* that defines two phases per stage. In this case, the detailed pipeline structure based on phases provides higher accuracy. However, the operation assignment alone will not execute any operation. In order to execute operations, they have to be activated.

3.7.5 Activation of Operations

The ACTIVATION section of an operation lists operations to be executed in subsequent control steps according to their respective pipeline assignment. Thus, activated operations do not execute immediately. They rather wait for the control step the activating operation enters the pipeline stage.

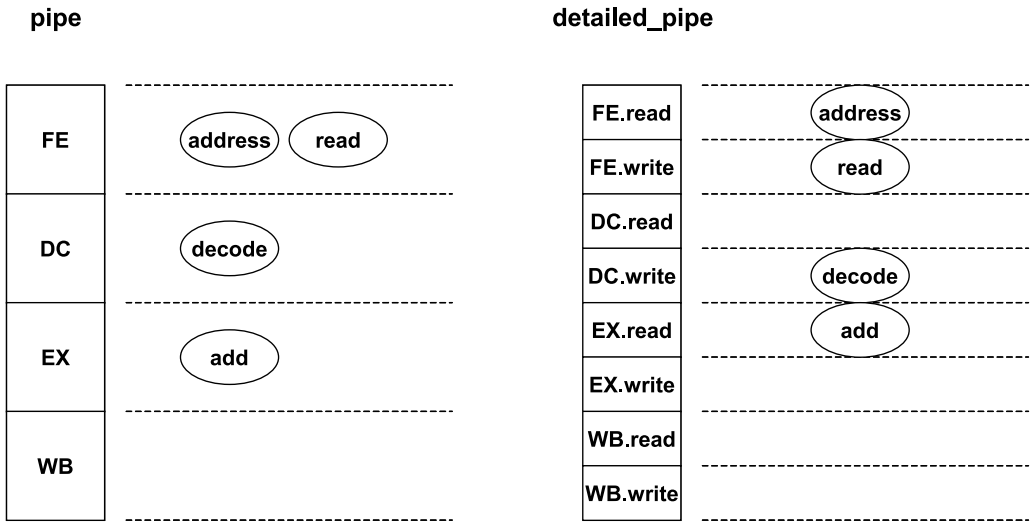

Figure 3.7.3: Operation assignment

This mechanism resembles the shifting of instruction words in the pipeline and the execution of operations in designated pipeline stages. Usually, the activated operations activate further operations in turn such producing an activation chain. Thus, the timing of instructions is described relative to each based on operation assignment and operation activation.

```
OPERATION fetch IN pipe.FE {
  DECLARE    { INSTANCE  decode; }
  BEHAVIOR   { instruction_register = prog_mem[pc]; }
  ACTIVATION { decode }
}

OPERATION decode IN pipe.DC {
  DECLARE    { INSTANCE  add, writeback; }
  ACTIVATION { add, writeback }
}

OPERATION add IN pipe.EX {
  DECLARE    { GROUP  Src1, Src2, Dest = register; }
  BEHAVIOR   { result = Src1 + Src2; }
}

OPERATION writeback IN pipe.WB {
  DECLARE    { REFERENCE  Dest; }
  BEHAVIOR   { Dest = result; }
}
```

Example 3.7.1: Activation of operations.

The example 3.7.1 shows such an activation chain for the instruction *add* that consists of four operations. The operation *fetch* is assigned to the stage *FE* and activates operation *decode* which in turn activates *add* and *writeback*. The simple activation chain in this example lets the instruction *add* complete execution within four cycles assuming that the instruction execution is not affected by pipeline stalls or other control operations. Figure 3.7.4 illustrates the movement of the instruction through the pipeline for this case. In cycle k, the instruction enters the pipeline and operation

fetch is executed. The arrows denote the activations that the execution of the other operations in the following cycles.

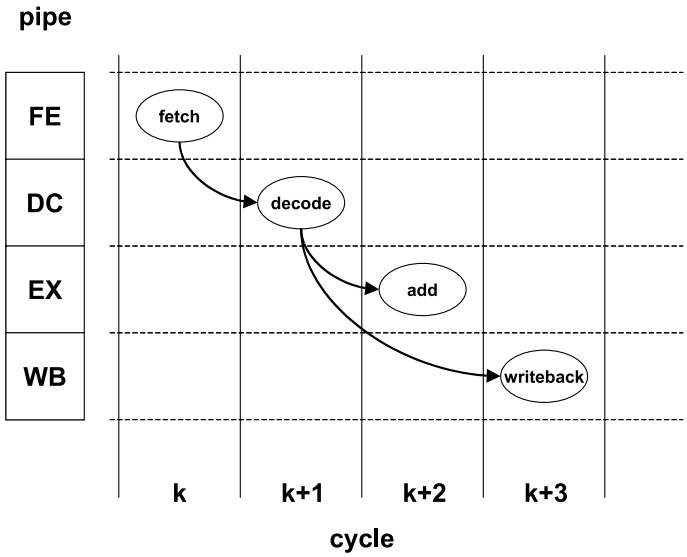

Figure 3.7.4: Operation activation

The actual instruction latency between entering the pipeline and finishing execution not only depends on the number of pipeline stages that have to be passed. Control hazards, data hazards or structural hazards can cause multiple pipeline stalls or even cancellation of the instruction execution. The total delay measured in control steps k_{total} between execution of two LISA operations during simulation is defined by the operation assignment to pipeline stages (*spatial delay*), the mode of activation (*temporal delay*) and the *pipeline control* operations stalling the pipeline:

$$k_{total} = k_{spatial} + k_{temporal} + k_{stall} \tag{3.1}$$

3.7.6 Spatial Delay

The activation of operations that are assigned to pipeline stages or phases that are downstream in the pipeline (in subsequent pipeline stages) produces an execution in a later control step. The delay between the moment of activation and the earliest possible execution is caused by the spatial distance in the pipeline. We will call it *spatial delay*. The spatial delay is statically defined through the number of pipeline stages between the activating and the activated operations. In Figure 3.7.4, the operation *fetch* is assigned to stage *FE* and *writeback* is assigned to *WB*. If the pipeline is shifted forward in each cycle, the operation *writeback* will execute three cycles after *fetch*.

3.7.7 Temporal Delay

The temporal delay is an explicit mechanism of the LISA generic model to delay operations beyond the spatial delay of pipeline assignments. The activation section consists of a list of operations that are separated by *activation operators.* There are two types of activation operators:

- concurrent activation operator: comma (,) and
- delayed activation operator: semicolon (;).

All listed operations separated by commas are activated in the current control step. The execution order of operations separated by semicolons is defined as “left before right”. For operations that are separated by commas, the written order has no effect. Each semicolon specifies a further delay of one control step before all following operations are activated. If there are several delayed activation operators, the total temporal delay accumulates upon moving towards the end of the activation list. Multiple delays can be specified by using brackets with the number of delays:

```
ACTIVATION { mul1 ; mul2 ;[3] mul3 }
```

Here, the operation *mul1* will be activated in the same control step without delay. Operation *mul2* will be delayed by one control step and *mul3* receives a further delay of three control steps. We will now assume that this activation list is part of the operation *decode* that is assigned to stage *DC* and that all activated operations are assigned to stage *EX* of pipeline *pipe*. The resulting operation schedule is depicted in Figure 3.7.5. The first delay between *decode* and *mul1* is a pure spatial delay. The operations *mul2* and *mul3* receive additional, temporal delays. Thus, *mul3* executes five cycles after *decode*.

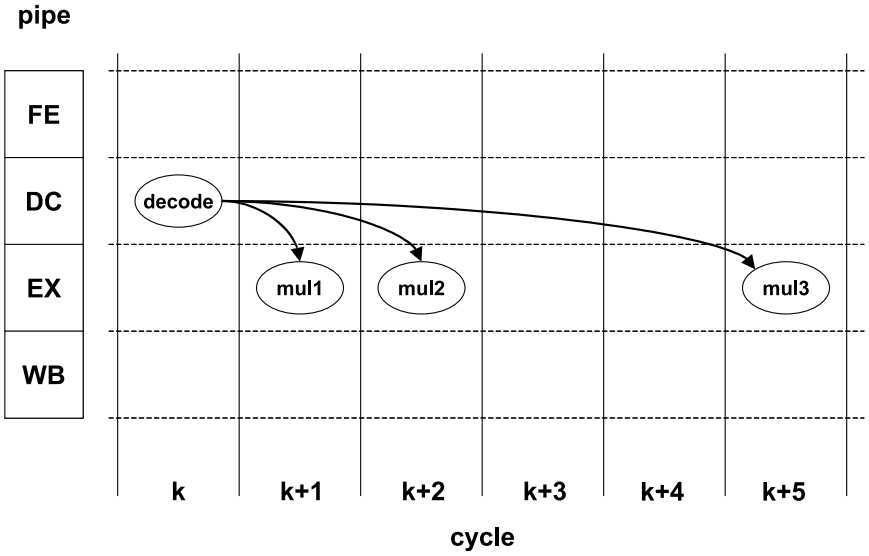

Figure 3.7.5: Operation activation with temporal delay

So far we have considered the ideal case of undisturbed execution of instructions in the pipeline which is certainly the desired mode of pipeline operation. However, pipelines involve different types of hazards that have to be handled with appropriate pipeline control mechanisms.

3.7.8 Pipeline Control

Spatial and temporal delay alone disregard control operations that may affect the flow of instructions in the pipeline due to pipeline hazards. Hazards are situations that prevent instruction in

the pipeline from executing during its designated clock cycle. According to [33], there are three classes of hazards:

- *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
- *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- *Control Hazards* arise from the pipelining of branches and other instructions that change the program counter.

The elimination of hazards often requires that some instructions in the pipeline are allowed to continue execution while others are delayed. The delay of instructions is produced by stalls of the respective pipeline stages. In same cases, the detection of pipeline hazards occurs so late that instructions already have entered the pipeline that are not meant to complete execution. They have to be removed from the pipeline which is implemented by the mechanism of pipeline flushes. The generic machine model of LISA provides four types of pipeline operations to control the stream of instructions:

- The *execute()* operation brings all activated operations to execution that have reached their designated pipeline stage.
- The *shift()* operation transfers the contents of pipeline registers from one pipeline stage to the next.
- A *stall()* inhibits shift operations on the respective pipeline stage.
- The *flush()* removes all activated operations from the pipeline that are in the respective stage.

Although it would be theoretically possible to achieve the same functionality with only the three pipeline operations *shift*, *execute* and *flush*, it is a practical and very reasonable approach to use the *stall* operation as well. Contrary to the other operations, the *stall* does not cause any action. It inhibits the *shift* operation from execution. Since stalls cause the pipeline performance to degrade from the optimum, they represent unwanted exceptions of the ordinary operation. Thus, stalls are situations that only arise under particular conditions. Therefore, it is much easier and much more efficient to model the particular behavior explicitly by using the *stall* pipeline operations rather than building complex condition expressions that list the cases of exceptions in the ordinary pipeline flow.

All pipeline operations can be applied to single stages as well as to whole pipelines. However, the pipeline operations do not execute all at the same time. Two phases have to be distinguished that resemble the behavior of synchronous processor hardware structures.

In phase 1, the embedded, mostly combinational logic between the pipeline registers is active – the actual functionality of the processor in the pipeline stages. The pipeline operation *execute* performs the execution of the operations that have reached their designated pipeline stage. This is illustrated on the left side of Figure 3.7.6 for a four-stage pipeline and execution of all pipeline stages.

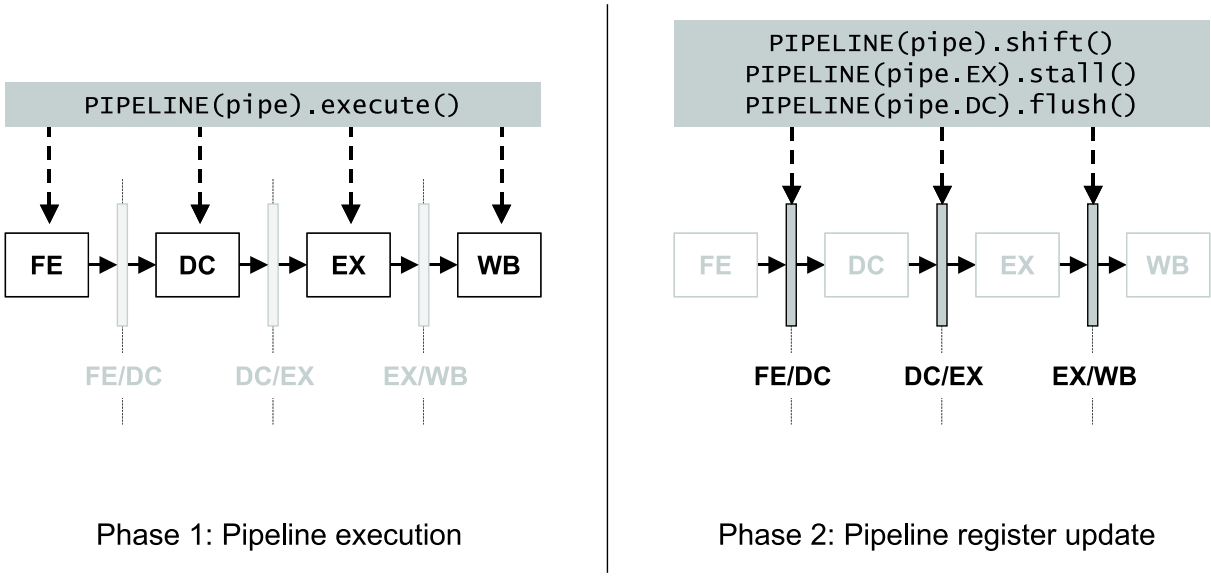

Figure 3.7.6: The two phases of pipeline operations

In phase 2, the pipeline registers are updated. Hardware design requires that the activities of phase 1 complete prior to the end of the current cycle. The reason is that with the clock cycle boundaries the register contents are updated which requires settled inputs. The pipeline operations *shift*, *stall* and *flush* are active in this phase. The right side of Figure 3.7.6 shows three pipeline operations that update the pipeline registers at the end of the cycle.

Execution of both phases simulates one cycle of the processor. For continuous simulation, the two phases of the pipeline must be processed alternately. Regarding the pipeline operations, the *execute* and the pipeline register update operations must not be randomly placed into the LISA model. However, the order of pipeline register update operations is not relevant for the resulting behavior. Shifting the whole pipeline and stalling a single stage afterwards or vice-versa will stop the shift operation in this specific stage. The syntax for pipeline operations is as follows:

```
PIPELINE(pipe).shift();
```

```
PIPELINE(pipe.EX).stall();
```

All common pipeline operations can be reduced to combinations of the primitive pipeline operations in LISA. For example, inserting operations in a pipeline stage and simultaneously replacing operations in this stage can be traced back to a flush followed by the activation of the respective operation.

The additional delay caused by pipeline operations contributes to the total delay between two operations. Figure 3.7.7 depicts the *add* instruction of example 3.7.1. In cycle $k+2$, the instruction enters the *EX* stage of the pipeline. Due to a stall, the instruction resides in the stage (and the instruction word in the pipeline register DC/EX. Therefore, the writeback operation is not executed until the instruction enters the *WB* stage.

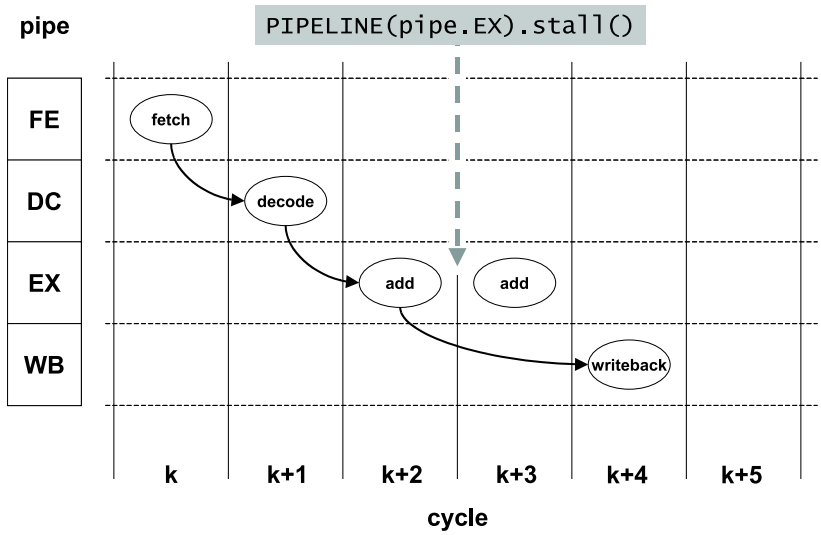

Figure 3.7.7: Delay of operation execution caused by a pipeline stall

3.7.9 Main Operation

In the processor hardware, each instruction word that enters the pipeline initiates a new instruction execution sequence. The supply of new instructions is supervised by the general processor control. The processor control can be seen as a continuous process that is active in each clock cycle. The corresponding mechanism in the LISA model is the operation with the reserved name *main*. The *main* operation is the root of the operation tree. The contents of the behavior and activation section of this operation are executed automatically in each control step. Thus, the *main* operation is the instance that defines the general processor control and determines the mechanisms of launching new instructions. Example 3.7.2 shows the *main* operation taken from a model of the DLX processor with pipeline interlock mechanism that is described in [33].

```
OPERATION main {
  DECLARE { INSTANCE fetch; }
  ACTIVATION {
    if (!MEM_data_hazard) { fetch }
  }
  BEHAVIOR {
    PIPELINE(dlx_pipe).execute();
    if (MEM_data_hazard) {
      PIPELINE(dlx_pipe.IF).stall();
      PIPELINE(dlx_pipe.ID).stall();
    }
    PIPELINE(dlx_pipe).shift();
    cycle++;
  }
}
```

Example 3.7.2: The main operation of DLX with interlocking.

The operation description contains a declaration of the operation *fetch* that represents the execution in *ID* – the first stage of the DLX pipeline. The activation of this operation is conditional and checks

the signal *MEM_data_hazard*. If the hazard arises, no new instruction may enter the pipeline. The behavior section contains the pipeline control operations. All activated operations are executed and afterwards the pipeline is shifted. If a hazard occurs, the stages *IF* and *ID* are stalled (and not shifted).

3.7.10 Interrupts

Interrupts – also called exceptions – are situations where the instruction execution order is changed in unexpected ways. Interrupts can be triggered by the software (breakpoints, arithmetic over- and underflows, undefined instructions, etc.) or caused by memory accesses and peripherals (e.g. timers or I/O devices). Most of these events require a prompt reaction or service and partially executed instructions in the pipeline must be aborted. This behavior can be described based on pipeline control operations that flush instructions from certain pipeline stages and the activation of the interrupt operation in the appropriate stage.

3.7.11 Processor Pipeline Description Issues

Complex Pipelines

The primitive pipeline declarations of LISA always refer to simple pipelines without branches and junctions. However, pipeline implementations in real-life processor architectures are frequently much more complex or even combined of several pipelines. Such pipeline structures must be composed from several pipelines instances. In the generic model of LISA, new instructions enter a pipeline through the activation of an operation that is assigned to the first stage (see Figure 3.7.8).

Figure 3.7.8: Primitive pipeline structures

The advancement of the instruction in the pipeline is described by the activation chain of operations. Consequently, two pipelines can be combined through activation links. This shall be shown for an example based on the pipeline of Figure 3.7.8. We extend the processor by an additional multiplier unit that needs three cycles to execute. The multiplier is inserted as a parallel path to the *EX* stage of the pipeline as shown in Figure 3.7.9.

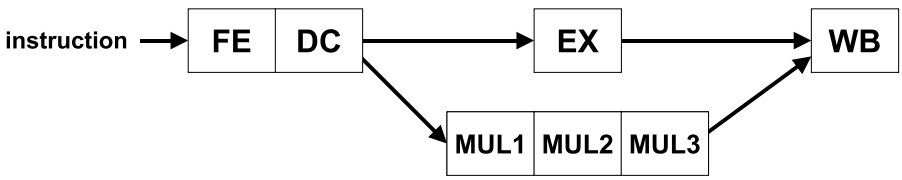

Figure 3.7.9: Composed pipeline structure

In addition to the main pipeline, a multiplier pipeline must be declared in the resource section. Depending on the type of instruction that enters the *DC* stage of the pipeline, an operation is activated that either is assigned to the *EX* stage or assigned to the first stage of the multiplier

pipeline *MUL1*. The multiply instructions complete execution after three cycles and return to the main pipeline based on the activation mechanism. Based on the activation mechanism and the pipeline structures, arbitrary complex or multiple pipeline structures can be described.

VLIW Processors

VLIW processors issue a fixed number of instructions that are scheduled by the compiler. Since also the burden of choosing instructions to be issued simultaneously falls on the compiler, the hardware structures of forwarding instructions to their designated functional units can be quite simple compared to the required logic in superscalar processors. Typical VLIW processors issue up to eight instructions per machine cycle [33]. Especially if the assignment of instructions in the long word to specific functional units is static, the possible variety of instruction issue is not very rich. The instructions to be issued simultaneously can be described by multiple operations that are activated concurrently.

Recent processor implementations replace the rigid execution scheme of traditional VLIW processors by configurable packets or types of instructions that are issued simultaneously as defined by the compiler or assembly optimizer. Since the traditional VLIWs can be considered a subset of the configurable VLIWs, we regard in the following an example from the latter group.

The C62x DSP of Texas Instruments can issue between one and eight instructions per machine cycle. The processor always reads packets of eight instruction words and uses a dispatch unit to issue the required instructions depending on a single bit of the respective instruction word coding. This bit determines for each instruction of the packet, if the following instruction will be issued in the same or in the next machine cycle. An adequate LISA description of this dispatch unit is shown in example 3.7.3.

```
OPERATION dispatch IN pipe.DP {
  DECLARE {
    GROUP Insn1, Insn2, Insn3, Insn4,
          Insn5, Insn6, Insn7, Insn8 = SeqInsn || ParInsn; }
  CODING { ... }
  SYNTAX { ... }
  IF (Insn1 == SeqInsn) { ACTIVATION { Insn1; } }
                   ELSE { ACTIVATION { Insn1, } }
  IF (Insn2 == SeqInsn) { ACTIVATION { Insn2; } }
                   ELSE { ACTIVATION { Insn2, } }
  IF (Insn3 == SeqInsn) { ACTIVATION { Insn3; } }
                   ELSE { ACTIVATION { Insn3, } }
  IF (Insn4 == SeqInsn) { ACTIVATION { Insn4; } }
                   ELSE { ACTIVATION { Insn4, } }
  IF (Insn5 == SeqInsn) { ACTIVATION { Insn5; } }
                   ELSE { ACTIVATION { Insn5, } }
  IF (Insn6 == SeqInsn) { ACTIVATION { Insn6; } }
                   ELSE { ACTIVATION { Insn6, } }
  IF (Insn7 == SeqInsn) { ACTIVATION { Insn7; } }
                   ELSE { ACTIVATION { Insn7, } }
  ACTIVATION { Insn8 }
}
```

Example 3.7.3: Description of the C62x dispatch mechanism.

The operation *dispatch* declares eight instructions as group elements that are either executed sequential *SeqInsn* or parallel *ParInsn*. Depending on the actual coding of some instruction that is not shown in the example, a selection of the following activation sections becomes valid[1]. The selection of valid activation sections is based on compile-time statements. Unlike the if-else or switch-case structures of the C code in the behavior section that are evaluated at run-time of the simulation, the compile-time statements IF-ELSE and SWITCH-CASE of LISA are evaluated at compile-time. Therefore, the arguments of the compile-time statements are not values rather than group assignments. The assignments of the instructions *Insn1* to *Insn7* decide on the selection of a segment of sequential or parallel operation activation. The complete activation section becomes composed from the concatenation of all valid segments. The advantage of this decomposition of the activation sections is that the $2^7 = 128$ possible combinations of sequential and parallel instructions can be described without listing them explicitly.

Superscalar Processors

In statically scheduled superscalar processors, the instruction schedule is defined by the HLL compiler. However, the dynamic issue capability of superscalar processors involves run-time decisions to determine the simultaneous instructions. If we compare that mechanism to the C62x example in the previous section, then the outcome from these decisions is equivalent to the encoded bits that determine the sequential and parallel instructions. We can make turn the model of example 3.7.3 into a superscalar processor architecture by converting the IF-ELSE compile-time statements into run-time statements.

LISA allows the use of run-time statements in the activation sections. Example 3.7.4 shows the converted dispatch unit in the static superscalar version. Since the encoding of parallelism has disappeared, the instructions have turned into instance declarations instead of groups. The logic that decides on the instruction issue is hidden in the C function *seqInsn()* that returns a one if simultaneous execution is not possible and a zero otherwise. The result is taken by the if-else statement in the activation section that issues the eight instruction accordingly.

In dynamically scheduled superscalar processors, also the scheduling of instruction issue and execution is handled dynamically. The number and type of instructions to be issued is selected at run-time based on score-boarding techniques. The order of instructions being fetched from the instruction memory in these processors is not necessarily the order of execution and the order can also change with each iteration a certain block of code is executed. The instructions to be issued and executed are selected based on the available hardware resources and general limitations of parallelism.

If such processors shall be described in LISA, the decisions in the activation sections can become quite complex. Since these types of processors are not common in embedded systems, the issues of describing reservation stations, reorder buffers and speculative execution have not been explored in this work.

[1] A description of the coding section for this example is provided in section 3.8.3.

```
OPERATION dispatch IN pipe.DP {
  DECLARE {
    INSTANCE Insn1, Insn2, Insn3, Insn4,
             Insn5, Insn6, Insn7, Insn8 = Instruction; }
  CODING { ... }
  SYNTAX { ... }
  ACTIVATION { if (seqInsn(1)) { Insn1; }  else { Insn1, }
               if (seqInsn(2)) { Insn1; }  else { Insn2, }
               if (seqInsn(3)) { Insn1; }  else { Insn3, }
               if (seqInsn(4)) { Insn1; }  else { Insn4, }
               if (seqInsn(5)) { Insn1; }  else { Insn5, }
               if (seqInsn(6)) { Insn1; }  else { Insn6, }
               if (seqInsn(7)) { Insn1; }  else { Insn7, }
               Insn8 }
}
```

Example 3.7.4: Superscalar version of the C62x dispatch unit.

3.8 Instruction Set Model

The instruction set model reflects the programmers view of the processor – the software-oriented part of the model. There are two manifestations of instructions in use with processors: assembler statements and binary coded instruction words. LISA captures the description of these manifestations in two different sections: *Coding* and *syntax*.

3.8.1 Coding Section

The CODING section describes the bit fields of the instruction word coding. The coding section C is a concatenation of coding elements c_k. Using the concatenation operator ($+$), this is denoted as follows:

$$C = c_1 + c_2 + ... + c_n$$

The coding element are either

- terminals t_c specified as binary or hexadecimal constants or
- non-terminals n_c that reference coding sections of other operations (based on groups or instances).

$$c ::= t_c \mid n_c$$

Terminal coding fields are binary strings composed from the alphabet a which is the set of zeros, ones and don't care bits $a ::= 0 \mid 1 \mid x$. The zeros and ones are used to encode and decode instruction words. The don't care bits describe bits that provide immediate operands or other fields that cannot used for the decoding decisions.

The non-terminals refer to inferior operations that provide a coding section itself. We must demand from all inferior operations that the coding sections specify the same bits width. Finally, the leafs

of the tree must not have non-terminals. This means that successively following the branches of the operation tree reduces the non-terminals to terminal codings.

A common problem in the description of instruction coding that is not solved in most machine description languages are split coding fields. These are multiple coding elements that build a logical unit. Usually, consecutive bits build logical units and can be composed to coding elements. However, split coding fields c_{split} are composed of several coding fields that are distributed over distant positions in the instruction word. This means that a split coding field are produced from the logical concatenation of sub-strings of the coding section:

$$c_{split} = c_{k1} + c_{k2} + ... + c_{kn}$$

The logical field can be composed from the sub-strings in random order. Figure 3.8.1 shows a 24 bit wide instruction word with seven bit fields. The logical field *cond* is a five bits wide split coding field. The most significant bit is bit 03 and least significant is bit 16.

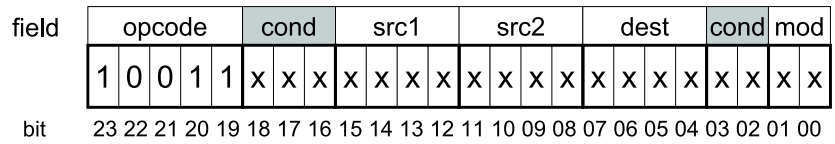

Figure 3.8.1: Split coding fields

The first field *opcode* is a five bit wide terminal coding. It unambiguously identifies the particular instruction, e.g. *add*. The other fields *cond*, *src1*, *src2*, *dest* and *mod* are non-terminals. The split field *cond* is described by specifying the sub-range of bits for each part to define the order of composition. The corresponding coding section for the instruction would look as follows:

```
CODING { 0b10011 cond[2..0] src1 src2 dest cond[4..3] mod }
```

Root of the Coding Tree

The hierarchical ordering of operations in LISA description produces operation trees. In most cases, the tree of the behavioral model covers more operations than the instruction set model. Only after its decoding the properties of instructions become visible and contribute to the instruction set model. During the addressing and fetching of instructions, instructions are treated uniformly. This means that the respective operations describing this process cannot have a coding or syntax section. The first operation that provides such information is the *coding root operation.*

Instructions and their operands are recognized in the processor hardware by gradually decoding the logical bit fields of the instruction word that is held in the instruction register. The corresponding structure in LISA descriptions begins in the coding root operation. In order to decode instructions, the coding patterns defined by the totality of all described operations must be compared to the actual value of the current instruction word (or even multiple instruction words). Therefore, the resource holding the instruction word, the bit width and the address it is fetched from is specified in the coding root operation.

Example 3.8.1 shows a description of the coding root operation of the DLX processor. The instruction set of the DLX can be structured into three groups of instructions that are defined here in the declare section. The resource *pc* contains the address the instruction is fetched from and the

```
OPERATION decode IN pipe.DC {
  DECLARE {
    GROUP instruction = i_type || r_type || j_type;
  }
  CODING WIDTH (32) AT (pc) {
    PIPELINE_REGISTER(pipe.IF/DC).instruction_register => instruction
  }
  SYNTAX   { instruction ; }
  BEHAVIOR { instruction(); }
}
```

Example 3.8.1: Coding root operation of the DLX processor.

32 bit wide instruction word itself is taken from the pipeline register. The assignment operator => separates the instruction register and the non-terminal that represents the root of the coding tree.

3.8.2 Syntax Section

The purpose of the SYNTAX section is to describe the textual representation of instructions on the assembly-level. The syntax section S is composed from syntax elements s_k:

$$S = s_1 + s_2 + ... + s_n$$

The syntax elements are either:

- Terminal strings t_s describing mnemonics, operands, etc. or
- non-terminals n_s that reference other operations.

$$s ::= t_s \mid n_s$$

The terminals are either character strings which are enclosed in quotation marks (“) or numbers that directly correspond to coding elements. The correspondence between the coding element and the number of the syntax elements is provided by casting rules. The casting rule is a function that defines a one-to-one translation between bits and their numeric interpretation. LISA provides the following casting rules that are taken from the C language:

- s_k = (int) c_k
- s_k = (unsigned int) c_k
- s_k = (float) c_k
- s_k = (double) c_k

Example 3.8.2 shows the description of the instruction *ADDI* of the DLX processor. This instruction adds the value of a register (*rs1*) and an immediate value (*immediate*) and assigns the result to a register (*rd*). The syntax section begins with the terminal mnemonic string `''ADDI''` and is followed by the operands which are separated by commas (`'',''`). The first two operands are

the two registers involved and the last element is the immediate value with a casting rule (*immediate=(int)*) that describes the translation of the coding bits into the appropriate textual representation of DLX assembler statements.

```
OPERATION ADDI IN pipe.DC {
  DECLARE {
    GROUP rs1, rd = fix_register;
    LABEL immediate;
  }
  CODING { 0b001000  rs1  rd  immediate=0bx[16] }
  SYNTAX { "ADDI"  rd  ","  rs1  ","  immediate=%d }
  BEHAVIOR { rd = rs1 + immediate; }
}
```

Example 3.8.2: ADDI instruction of the DLX processor.

3.8.3 Instruction Set Description Issues

Non-orthogonality of Coding and Syntax

The instruction word coding schemes or assembly languages of DSPs often have elements that are non-orthogonal to other elements. This means in the assembly process that a coding field cannot be encoded independently from some further parameter provided in the assembler statement. In case of the disassembler it means that a coding field cannot be decoded independently from the decoding result of some other field. This is a problem for processor descriptions since they decompose the complete coding – respectively the complete assembler statement – into gradually smaller parts. Within the scope of these parts (resp. operations) the decoding result of other parts is not visible. LISA provides the mechanism of reference declarations (cf. section 3.5) to transfer the results of compile-time decisions between operations. The alternatives of coding, syntax or behavior depending on the compile-time decision can be listed by means of the compile-time statements IF-ELSE and SWITCH-CASE that are available in LISA. These conditional structures allow to select different lists of sections. The selector is a group or a reference. In contrast to the corresponding control-flow statements of the programming language C, these statements are evaluated at compile time.

VLIW Processors

VLIW processors encode the instruction to be issued parallel in long instruction words that can have up to several hundred bits. Since LISA provides the scalable data type *bit*, there are two approaches possible to describe VLIW instructions. In some cases it may be useful to place the whole, long instruction word into a single register based on the scalable data type. In our description of the C62x however, it was more convenient to split the long word into eight instructions that each represent one instance of the same coding tree. Example 3.8.3 lists the description of the

dispatch unit which is the coding root operation of the model[2].

```
OPERATION Dispatch IN pipe.DP {
  DECLARE {
    GROUP Insn1, Insn2, Insn3, Insn4,
          Insn5, Insn6, Insn7, Insn8 = Ser_Insn || Par_Insn;
  }
  CODING WIDTH (256) AT (PIPELINE_REGISTER(pipe.PR/DP).pce1 & PFC_MASK) {
       (IR[0] => Insn1)  && (IR[1] => Insn2)
    && (IR[2] => Insn3)  && (IR[3] => Insn4)
    && (IR[4] => Insn5)  && (IR[5] => Insn6)
    && (IR[6] => Insn7)  && (IR[7] => Insn8)
  }
  SYNTAX {
    Insn1 ; Insn2; Insn3 ; Insn4 ; Insn5 ; Insn6 ; Insn7 ; Insn8 ;
  }
  ACTIVATION { ... }
}
```

Example 3.8.3: Description of the C62x dispatch mechanism.

The eight instructions are stored in the array of instruction registers *IR*. They are each assigned to one element of the same group declaration such defining the eight identical coding trees. The and operators (&&) between the assignments require that all eight assignments have to be made at once. Since the instructions are 32 bit wide, the complete long word of 256 bit is consumed. In the syntax section, the elements of each instruction must be terminated with semicolons to separate assembler statements that belong to different instructions.

Variable Instruction Word Length

In many processors, the instruction word length is identical for all instructions. However, there are some processors like the C54x DSP of Texas Instruments that have different instruction word lengths. Due to the pipelined execution, instructions are typically encoded in multiples of the shortest instruction words. The actual length of the current instruction must be derived from the coding of the word that is fetched first.

A simplified description of the coding root of the Texas Instruments C54x DSP that has 16 bit, 32 bit and 48 bit instruction words is provided in example 3.8.4. The enumeration (*ENUM*) lists three alternative instruction types that are described in the compile-time switch statement. The cases provide the description of coding roots that define three different coding trees (*Type1*, *Type2* and *Type3*). The second and the third instruction words are treated as separate fields which can be accessed from within the coding trees by means of references.

Instruction Aliasing

While the instruction word coding non-ambiguously identifies one instruction of a processor, the same does not necessarily apply to the opposite direction. The assembly languages of many proces-

[2]The activation section is left out here because it is already described in section 3.7.11.

```
OPERATION Decode IN pipe.DC {
  DECLARE {
    ENUM InsnType = { Type1, Type2, Type3 };
    GROUP InsnGroup1 = InsnType1;
    GROUP InsnGroup2 = InsnType2;
    GROUP InsnGroup3 = InsnType3;
    GROUP Addr1, Addr2 = address;
  }
  SWITCH (InsnType) {
    CASE Type1: {                              // 16 bit instruction
      CODING WIDTH(16) AT (PIPELINE_REGISTER(pipe.FE/DC).pc)
        { PR.DC => InsnGroup1 }
      SYNTAX { InsnGroup1 ; }
      ACTIVATION { InsnGroup1 } }
    CASE Type2: {                              // 32 bit instruction
      CODING WIDTH(32) AT (PIPELINE_REGISTER(pipe.FE/DC).pc)
        { (PR.DC => InsnGroup2) && (PR.FE => Addr1) }
      SYNTAX { InsnGroup2 ; }
      ACTIVATION { InsnGroup2 } }
    CASE Type3: {                              // 48 bit instruction
      CODING WIDTH(48) AT (PIPELINE_REGISTER(pipe.FE/DC).pc)
        { (PR.DC => InsnGroup3) && (PR.FE => Addr1) && (PR.PF => Addr2) }
      SYNTAX { InsnGroup3 ; }
      ACTIVATION { InsnGroup3 } }
  }
}
```

Example 3.8.4: Coding-root operation of the C54x describing variable instruction word length.

sors comprise different instructions which are mapped onto the same instruction word. In Figure 3.8.2, two assembler statements are shown that translate into the same instruction coding.

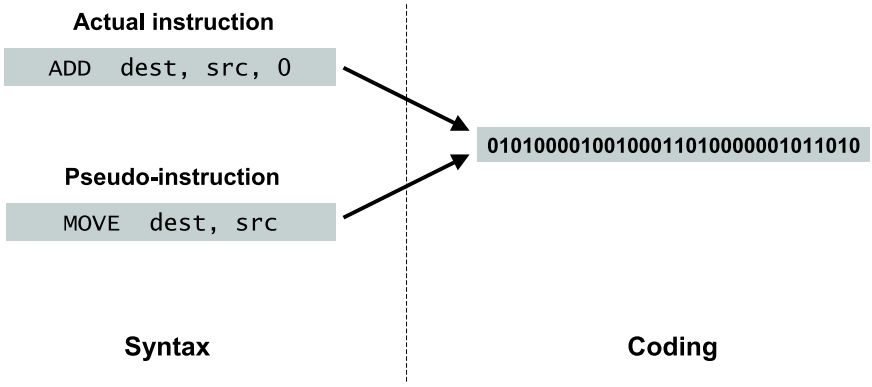

Figure 3.8.2: Ambiguous relation between coding and syntax

The first statement is an special case of an add instruction that takes an immediate source operand. In this case, a zero is added to the source register *src* and the result is assigned to *dest*. The second assembler statement, the move instruction, is a pseudo instruction that is not supported explicitly by the processor hardware – the data path available to transfer register values is routed through the ALU unit. The disassembler must translate the instruction word coding into the appropriate assembler statements and requires a translation rule for this case. In LISA, the translation rule is described by *instruction aliasing*. Instruction aliases are separate operations that describe a special case of the coding section found in the aliased instruction as shown in example 3.8.5.

```
OPERATION ADD
{
  DECLARE  { GROUP dest, src = register;  }
  CODING   { 0b01100  dest  src  immediate }
  SYNTAX   { "ADD"  dest  ","  src  ","  immediate }
  BEHAVIOR { dest = src + immediate; }
}

ALIAS OPERATION MOVE
{
  DECLARE  { GROUP dest, src = register;  }
  CODING   { 0b01100  dest  src  0x00000000 }
  SYNTAX   { "MOVE"  dest  ","  src }
  BEHAVIOR { dest = src; }
}
```

Example 3.8.5: Aliasing of instructions.

The coding section of both operations contains the same opcode and operand fields, however the *MOVE* instruction requires the immediate value to be zero. The *ALIAS* keyword achieves that the operation is checked first in the pattern matcher of the disassembler. This rule removes the ambiguity of the translation from coding to syntax.

Access Variants

A similar problem affects the relation between behavior/expression and the coding section. In some instructions of the TI C62x, the access to objects that are selected by instruction coding appears in variants in the same operation. The problem shall be explained based on example 3.8.6.

```
OPERATION SHR_long {
  DECLARE {
    GROUP dest, src = Register;
    LABEL shift;
  }
  CODING { 0b0101101  dest  src  shift=0bxxxxx }
  SYNTAX { dest "=" src ">>" shift; }
  BEHAVIOR {
    dest.high = src.high >> shift;
    dest.low  = ((src.high << (32 - shift)) | (src.low >> shift));
  }
}
```

Example 3.8.6: Access variants in the behavior section.

The operation *SHR_long* describes an instruction that shifts the source operand right by the number of bit positions that are specified by the immediate value *shift*. The instruction operates on long operands that each span over two consecutive registers of the register file. Therefore, the instruction reads the source registers *src.low* and *src.high* and writes the two registers *dest.low* and *dest.high* in the behavior section. However, the coding is provided only for the registers with the lower

index. Therefore the selection between both registers involved must be made in the behavior section. In the example, the suffixes perform the selection. Consequently, the operation *Register* must describe both variants that are accessed from operation *SHR_long*. Example shows a suitable LISA description of the registers. The two required variants are placed into sub-operations.

```
OPERATION Register {
  CODING { index=0bxxxxx }

  SUBOPERATION low {
    SYNTAX { "R"  ~index=%u }
    EXPRESSION { R[index] }
  }
  SUBOPERATION high {
    SYNTAX { "R"  ~(index+1)=%u  ":"  "R"  ~index=%u }
    EXPRESSION { R[(index+1)] }
  }
}
```

Example 3.8.7: Description of the register access.

Sub-operations allow to create several operations which are grouped under the same main identifier. These operations have their own sub-names to permit access. In operation *Register* both sub-operations share the same coding. But the syntax and expression section is specific to the sub-operations. Without sub-operations or an equivalent mechanism it is hardly possible to describe these type of instructions.

3.9 Summary

In this chapter, the machine description language LISA and its generic model was presented. The language covers all required model components for the generation of software development tools. Numerous examples and description problems derived from real processor architectures have been examined and suitable LISA descriptions have been presented. The generic machine model was discussed with regard to complex processor pipelines and their mechanisms.

Using retargetable tools and machine descriptions enables designers to explore processor architectures and analyze candidates with accurate performance measurements and profiling based on benchmark applications. LISA processor models can be refined to very accurate models that can be used for the verification of application software and co-verification in hardware/software co-design environments. Fast simulation is a key factor for successful architecture exploration and verification complete systems [34, 72]. The next chapter will present the implementation of fast simulators that can be retargeted based on LISA processor descriptions.

Chapter 4

Retargetable Processor Simulation

This chapter explores principles of retargetable processor simulation based on LISA machine descriptions with focus on compiled simulation techniques. A retargetable simulator is generated from two components: On the one hand the generic machine model that collects the properties that are common to all machines of the target class of processors and on the other hand the processor description that defines the specific properties of the machine that shall be simulated (see Figure 4.0.1). The product of the generation process is the *processor specific* simulator.

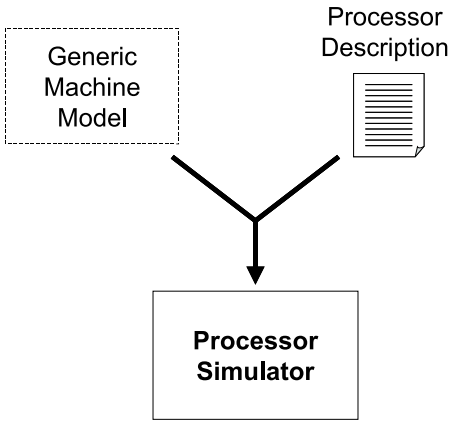

Figure 4.0.1: Retargetable simulator

Since the goal of this work is the generation of fast processor simulators based on compiled simulation techniques, the simulator is not a single software program. It is a generation process itself. The following section will discuss the compiled simulation principle and the techniques used in this work. In the following sections, the implementation of the LISA generic machine model and the simulator generation process are explored.

4.1 Compiled Processor Simulation

The objective of compiled simulation is to reduce the simulation time. In general, efficient run-time reduction is achieved by accelerating frequent operations. The principle of compiled simulation

is to take advantage of a priori knowledge and to move frequent operations from simulation run-time to compile-time with the goal of improving simulation speed. Since this approach requires a translation step to be performed before the actual simulation can be run, a speed-up of compiled simulation over interpretive simulation is only achieved if the time spent for the translation step t_t is outweighed by the shorter simulation time t_c of the compiled simulator. The compiled simulation is the better choice, if the time consumed by the interpretive simulator is greater that the total time required by the preprocessing and the compiled simulation itself: $t_i > t_c + t_t$.

Regarding the compiled simulation of programmable architectures, the technique for accelerating operations is to take advantage of a priori knowledge by translating the program code of the target processor into simulation code for the host computer. Compiled simulation of programmable DSP architectures was introduced to speed up instruction set simulation [102] and was extended to cycle-accurate models of pipelined processors [98]. The principle of compiled simulation for DSPs corresponds to the ideas that are already successfully implemented in the simulation of synchronous VLSI circuits [7], constant propagation in high-level language compilers [1], and that are used for static multi-processor scheduling [44].

The instruction processing of programmable architectures is typically performed in several steps:

- Each instruction cycle is initiated by *instruction addressing* that determines the new address and sends it to the program memory.
- At *instruction fetch*, the instruction word is read from program memory and written into the instruction register.
- During *instruction decode*, the operations that are assigned to this specific instruction are determined.
- The order of instruction issue is determined in the step of *operation sequencing*.
- *Operation instantiation* collects and computes the parameters of the instruction like the involved registers, addresses and conditions to parameterize the sequenced operations.
- The *operation execution* performs the state update of the processor.

A fully interpretive simulator performs all these steps at run-time of the simulation. In a compiled simulator, the predictability in all steps but the last can be exploited and performed at compile-time. According to [98], several compiled simulation techniques can be distinguished that differ in their extend of predicting these steps such implementing several levels of compiled simulation. The different compilation levels allow trade-offs between required preprocessing time and the simulation speed that can be achieved. The main levels of preprocessing are depicted in Figure 4.1.1.

- The step of *compile-time decoding* determines the instructions, operands and modes from the respective instruction word. Since memory space is a valuable resource in embedded systems, the processors designed for this application area typically use compact instruction encoding. Therefore, the decoding of instruction can require complex decoding operations. The pipeline structures found in modern DSP processors illustrates the fact that instruction decoding consumes a significant amount of time. If we take for example the Texas Instruments TMS320C62x DSP, most instructions actually execute within only one pipeline stage

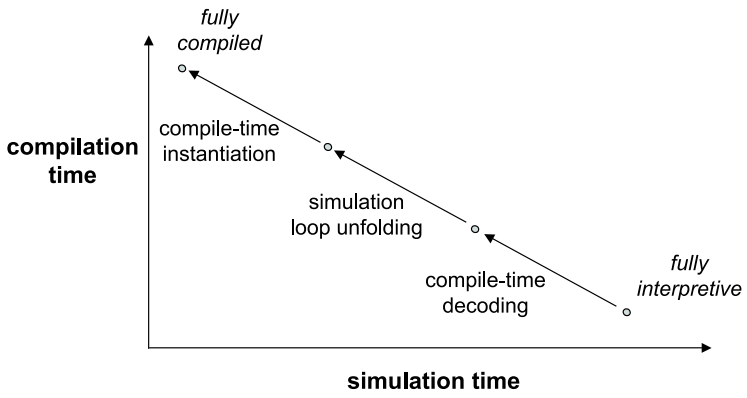

Figure 4.1.1: Levels of compiled simulation.

(or cycle), whereas fetching and decoding operations require six pipeline stages. The experience has shown that this step is the main source of simulation speed-up.

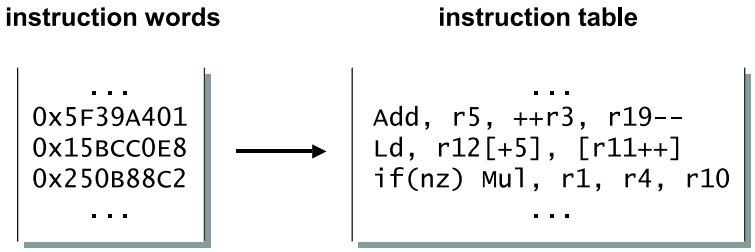

Figure 4.1.2: Compile-time decoding

Compile-time decoding translates the raw instruction words of the program memory into an evaluated representation that explicitly lists the type of instruction, operands, addressing modes, condition codes, etc (see Figure 4.1.2). The evaluated representation of the program memory is captured in an instruction-table that describes the decoded instructions. The simulation process is driven by the contents of the table. Re-compilation due to changes in the program memory means updating respective parts of the instruction table.

- Interpretive simulators use a main top-level loop that is iterated for the simulation of each instruction. The step of *simulation loop unfolding* creates simulation code for each instruction word in the program memory which unfolds the main simulation loop. In Figure 4.1.3, for each initialized address of the program memory one case of a large switch statement is generated. Each case is the label for a call to the function that uses the simulation table to select state update functions and the appropriate operands. A true advantage with respect of simulation speed can usually not be achieved based on this step. In many cases, it rather spoils the simulator performance because the large simulation programs typically do not fit in the caches of the host computer and are difficult to handle for the compiler.

  The motivation for using this technique is either the wish to implement even higher levels of simulation compilation or the requirements of debugging. In [98], the unfolding was a prerequisite of using an ordinary C source-level debugger. This approach has the advantage that no new debugger interface must be created. The execution of the compiled simulation can be controlled by the C source level debugger and breakpoints are set to the specific line of simulation code that corresponds to the program memory address, the program counter is pointing to.

```
case 0x8000: SimulateInstruction();
case 0x8001: SimulateInstruction();
case 0x8002: SimulateInstruction();
case 0x8003: SimulateInstruction();
                    ...
```

Figure 4.1.3: Simulation loop unfolding

- In *compile-time instantiation*, individual simulation code parameterized for each instruction of the application program is generated. Figure 4.1.4 shows instantiated instructions that have replaced the *SimulateInstruction* statement. An instruction table becomes obsolete at this level of compilation.

```
case 0x8000: Add(&r5, ++r3, r19--);
case 0x8001: Ld(&r12, 5, mem[r11++]);
case 0x8002: if(nz) Mul(r1, r4, r10);
                    ...
```

Figure 4.1.4: Compile-time instantiation

  This technique implements the highest possible level of compilation at the cost of even higher compilation time than simulation loop unfolding [76]. In case of cycle-accurate models of pipelined processors, the generated code must be specific to the overlapping of instructions in the pipeline. Since control flow instruction like branches and loops change the overlapping of instructions, traces have to be generated that cover all possible instruction combinations at the specific address in the program memory. Simulation code is generated for each trace and at simulation run-time, the appropriate trace is chosen. In case of a medium pipelined processor like the C54x DSP, the maximum possible number of traces that have to be followed concurrently does not exceed 32 traces and the mean trace count ranges between two and six, depending on the density of branches in the application code. However, the simulation code size caused by the multitude of traces is not acceptable for processors with longer or more complex pipelines.

4.1.1 Simulation Compilation

For the translation of application code of the target processor into simulation code of the host a simulation compiler is used. The simulation compiler can produce simulation code in several formats. In [98], the direct translation into host binary code and the generation of intermediate C code are proposed (see Figure 4.1.5. The advantage of direct translation is the high translation speed that is achieved at the cost of a very difficult implementation of the simulation compiler software. Furthermore, the simulation compiler is specific for a single host platform. The software must be ported to each new computing platform the simulation compiler and the compiled simulator shall be run on. The limitation to a single processor and the implementation issues are the reasons, why this approach has no practical relevance.

Generating C code has the key advantage of producing simulation code that is independent from the host platform. The C code becomes an intermediate format as depicted in Figure 4.1.5. The

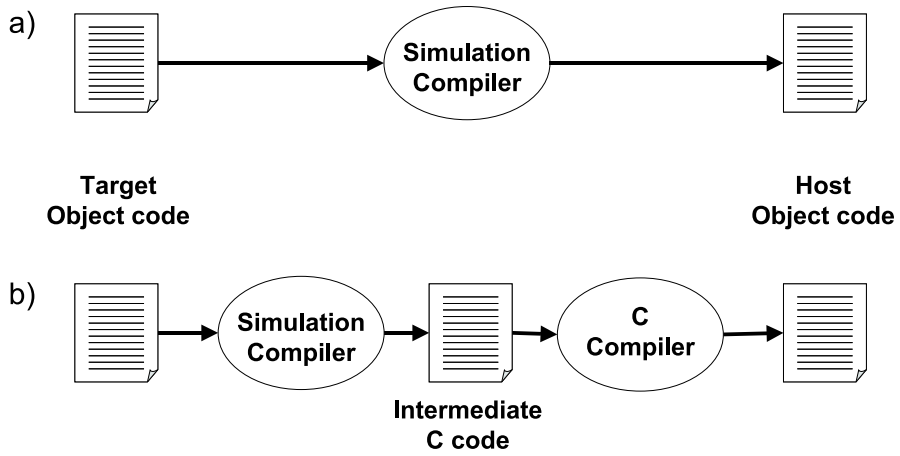

Figure 4.1.5: Simulation compiler

simulation compiler produces C source code of the simulator which is then translated on any host computer where a C compiler is available. In addition to the portability, this approach has the advantage that the simulation compiler implementation is less complex. The simulation compiler performs only the first step of the whole translation – generation of C code. The second step of translating the C code into optimized machine code of the host is covered by the available HLL compiler. The cost of the flexibility and reduced implementation complexity is the translation speed of simulation compilation.

Translation Speed

With higher levels of simulation compilation, also higher preprocessing time is required (cf. Figure 4.1.1) which is equivalent to lower translation speed of simulation compilation. If the highest level of compiled simulation – compile-time instantiation – shall be used, the translation speed can become a main issue. Unfortunately, available C compilers are not able to deal with the produced C code of the unfolded loops very well. The reason are the enormous switch-statements that comprise cases for each instruction of the program memory. Measurements of the simulation compilation of the C54x DSP have shown, that even when all C compiler optimizations are switched off, the HLL compilation time grows exponentially with the size of the target object code. The translation speed drops from 135 instruction words per second (i/s) for small kernels down to 26 i/s for the GSM speech coder/decoder program [75] and even down to 1 i/s if compiler optimizations are used. However, this low speed that corresponds to 42 minutes for simulator compilation is unacceptable. Subdividing of the basic blocks is a possible approach of reducing unacceptable growth of preprocessing times for large applications which is discussed in [75]. Here, the compilation time grows slower, but still exponentially.

The second level of compiled simulation – simulation loop unfolding – is not suitable to speed-up simulation because the large resulting simulation programs do not fit into caches of most host computers. Therefore, we will focus in this work on the technique of compile-time decoding and table-driven simulation to avoid long compilation times.

4.1.2 Previous Work

So far, the implementations of compiled simulator for embedded processors are restricted to specific processor architectures. The implementation of compiled simulators for some DSPs was carried through at the Laboratory for Integrated Signal Processing Systems (ISS). Simulators for DSPs have been realized for processor architectures like the Analog Devices ADSP21xx [69], Rockwell RSP [86] and Texas Instruments C54x [76]. All these compiled simulators consist of two parts: The simulation library that consists of the transition functions of the simulator and the simulation compiler (see Figure 4.1.6).

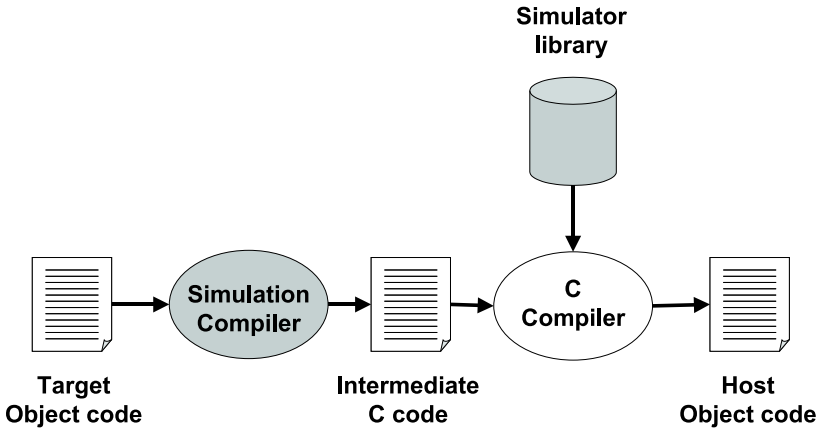

Figure 4.1.6: Simulation compilation

Since both parts are highly architecture-specific, it was not possible to reuse relevant parts from previous compiled simulator implementations. Therefore, the simulation libraries and the simulation compilers have been written from scratch. The development of the instruction-set accurate ADSP21xx compiled simulator took about six man months, while the development of the cycle-accurate, compiled C54x simulator took more than 18 man months although the source code of the commercial, interpretive simulator from Texas Instruments was available. The experience has shown that the co-design of the two parts simulation library and simulation compiler is tedious and very error-prone due to consistency problems. It is a very lengthy process of implementing an abstract model of the processor architecture. The design efforts can be reduced significantly and the consistency problem can be eliminated by using a retargetable simulator which is generated from a machine description and a generic machine model.

4.2 Implementation of the Generic Machine Model

The mechanisms of the generic machine model described in section 3.7.1 consist of assignment of operations to pipelines, operation activation and pipeline control operations. The following section will describe the activation tables – an extension of reservation tables – and suitable operation scheduling that implement the generic machine model.

Standard reservation tables and some extensions are commonly used for the detection of resource conflicts in pipeline design and scheduling [41]. These tables describe resource allocation over time assuming that each resource can be allocated by only one operation at a time. In the generic

machine model of LISA, operations are also assigned to pipeline stages or phases. However, tables are used here to reproduce the timing based on a given processor description. Due to instruction-level parallelism, multiple instructions – represented by LISA operations – may execute in the same pipeline stage or phase. Furthermore, pipeline control makes it necessary to e.g. remove operations from these tables. Therefore, the standard reservation tables are converted into *activation tables* that allow multiple operations to be assigned to a single resource and the application of control operations. The next sections will introduce the activation tables and discuss the mechanisms of pipeline control and operation scheduling.

4.2.1 Activation Tables

LISA processor descriptions allow the assignment of operations to pipeline stages or phases. The activation tables keep track of the current set of operations originating from overlapping instructions that are processed in the pipeline. Therefore, the dimensions of activation tables are determined by the structure of the pipelines declared in the resource section of the LISA description. Activation tables are created for each pipeline that is declared in the resource section of the LISA description. A separate activation table A^s is required for each pipeline stage or phase s. The activation table of the first stage (*FE*) of the pipeline *pipe* as declared in example 3.3.2 is shown in Figure 4.2.1.

FE	
FE	
DC	
EX	
WB	

Figure 4.2.1: Activation table for a single pipeline stage

It provides a field for each stage of the pipeline to list activated operations that are assigned to the respective stage. Instructions in the pipeline are assumed to move always downstream through the pipeline. Therefore, the operations that belong to the current instruction can only activate operations that are assigned to the same stage or to one of the following stages. Activation tables have fields for the current pipeline stage and for all following stages or phases. Consequently, the number of fields f in the activation tables are a variable of the position s in the pipeline and depend on the total number of stages or phases S in the pipeline: $f(s) = S - s + 1$.

The activation table collects the activated operations in the respective fields that are created for the other stages of the pipeline. Similar activation tables are created for the other stages, thus creating a set of activation tables as shown in Figure 4.2.2 for the four-stage pipeline *pipe*.

According to the four stages of this pipeline, a set of four activation tables must be generated. Within the activation chain of one instruction, operations can only be activated downstream of the pipeline. This means that operation in the first stage of the pipeline can activate operations in all further stages. However, operations in the last stage cannot activate operations in other pipeline stages as long as this is meant to be the same instruction. All activations of operations that are assigned to upstream pipeline stages initiate new instructions in the pipeline. For this reason, the size of the activation tables as shown in Figure 4.2.2 shrinks with each pipeline stage.

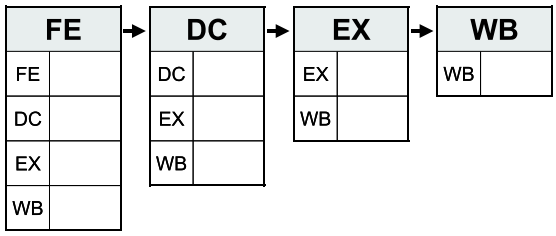

Figure 4.2.2: Set of activation tables for a four-stage pipeline

If the pipeline declaration defines a sub-division into phases, the set of activations tables is multiplied by the number of phases per stage and one table is necessary for each pipeline phase. In case of the pipeline *detailed_pipe* from example 3.3.2 with two phases, eight activation tables are necessary. Analogous to the set of tables for a pipeline based on stages, the tables shrink with each phase as depicted in Figure 4.2.3.

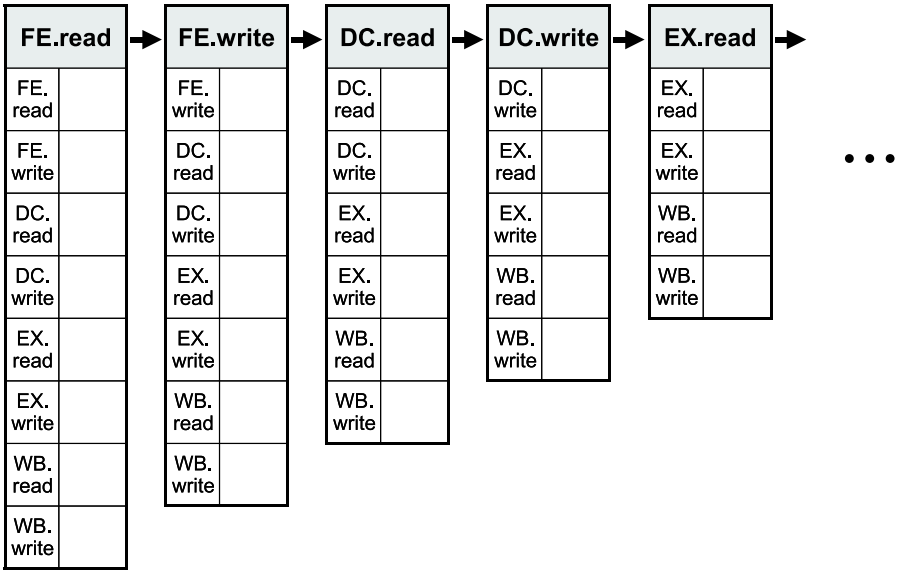

Figure 4.2.3: Activation tables for pipeline phases

Since multiple operations assigned to the same pipeline stage or phase can be activated from the same operation, each field of the activation tables contains a list of the activated operations. The mechanism of operation activation is described in the following section.

4.2.2 Operation Activation

The mechanism of operation activation allows to execute operations in subsequent control steps. The activation tables are used to list the activated operations until the instruction enters the designated pipeline stage or phase the operations are assigned to. In this control step, the operations are executed.

We assume an operation O_a with an activation section that is assigned to the pipeline stage s which shall be denoted as O_a^s. Upon execution of O_a^s, the identifiers listed in the activation section are entered into the activation table A^s of the current stage. Depending on the assignment to pipeline stages, the operations are listed in the field of the respective stage or phase. As the instruction

advances to the next pipeline stage $s + 1$, the activation table entries are transferred to the next activation table A^{s+1}. All operations that have reached their designated stage or phase are executed in this control step and removed from the activation table.

For example the operation *fetch* of the LISA description 3.7.1 is assigned to the stage *FE* of pipeline *pipe* and activates *decode* that is assigned to stage *DC*. As depicted in Figure 4.2.4, operation *decode* executes in the following control step $k + 1$ and is removed from the activation table of pipeline stage *DC*.

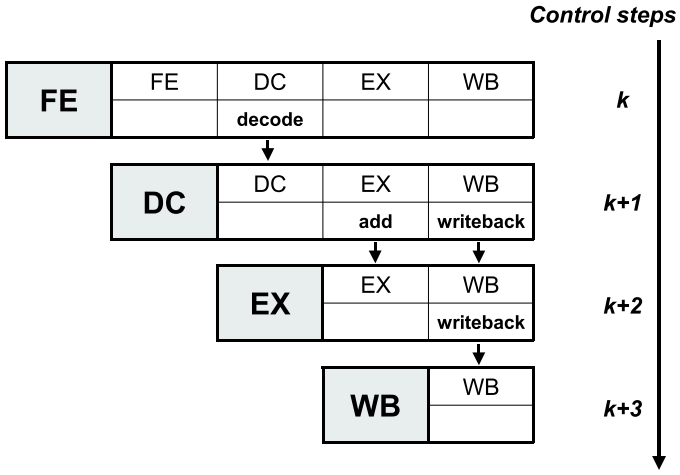

Figure 4.2.4: Activation of operations

Operation *decode* in turn activates *add* and *writeback*. *Add* executes in control step $k + 2$ and operation *writeback* remains in the activation table until control step $k + 3$ is reached.

When an operation is executed that contains an activation section, the names of the operations to be activated are written into these tables. Operation *decode* is the first entry in the activation table of the stage FE in the pipeline. Upon entering the DC stage in control step k+1, *decode* is executed and writes the operations *add* and *writeback* in the table of stage DC. At the same time the *decode* operation is removed from the table because it comes to execution. Add executes in the next control step and only *writeback* remains in the activation table to wait for execution in stage WB.

4.2.3 Temporal Operation Delays

As discussed in section 3.7.7, the execution of operations can be delayed by multiple control steps. The delay is specified in the activation section of the activating operation. Since the LISA language supports conditional statements in the activation section, it is possible to describe variable delays, such as the LISA code example 3.7.3 that describes the dispatch mechanism of the C62x DSP. Here, the temporal delay can range between zero and seven, depending on the configuration of parallel and sequential instructions in a fetch packet. However, the maximum delay count can be determined for a given processor model based on the activation sections of all operations. The fact that there is a upper bound for the delays for any processor model makes the implementation much easier. The maximum activation delays can be determined separately for each pipeline stage.

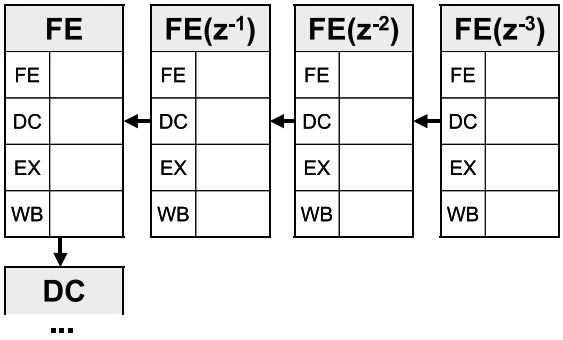

Figure 4.2.5: Multiple activation tables for temporal delays

The mechanism of temporal delays introduces a further dimension to the activation tables described in section 4.2.1. In order to resemble the delay of activations, multiple instances of the activation tables are needed for those pipeline stages that include operations with delayed activations. Figure 4.2.5 displays the resulting activation tables for the stage *FE* that includes a maximum activation delay of three. Delayed activations are entered in the activation table with the corresponding delay. After each control step, the contents of the delayed activation tables are moved to the next activation table with a lower delay. Only the content of the activation table without delays is forwarded to the next stage (which is stage *DC* in the example depicted in Figure 4.2.5). This means that the *activation* of the operations is delayed rather than the *execution.* The reason are the pipeline control operations. Without the mechanisms of pipeline control operations, the delay could also be bound to the stage of the activated operation. But in case of a pipeline stall or flush, the execution of the respective operations must be stopped. Binding the delay to the stage of the activating operation means that stalls and flushes are applied to all activation tables of the respective stage. Returning to the example of the C62x, a flush that is applied to the *DP* stage of the pipeline *pipe* would remove all instructions of the fetch packet that are currently waiting to be issued.

4.2.4 Pipeline Control Operations

The mechanism of operation activation resembles the decoding of instruction words and the generation of respective control signals in the processor hardware. In the LISA processor model, instructions are represented by pipeline registers and activation lists. Therefore, the treatment of the activation tables is immediately linked to the mechanisms of the pipeline registers. All pipeline control operations apply to the activation tables as well:

- With each pipeline shift, the entries in the activation tables must be moved from one stage to the next.
- In case of a pipeline stall, the activation table entries assigned to the respective stages remain in their table.
- In case of a pipeline flush, all operations in the respective pipeline stages are removed from the activation tables.
- Instructions that are injected into the pipeline, such as e.g. interrupts appear as operations that enter the appropriate table.

4.2.5 Operation Scheduling

All operations that execute in the same control step are parallel events in the processor model. Since these operations may be assigned to different pipeline stages of phases, an appropriate schedule must be used to efficiently simulate execution. First, we will regard the operation scheduling based on pipeline stages before we generalize the scheduling strategy to the level of phases.

Cycle-Accurate Models

LISA does not allow to specify any precedence between two simultaneous operations that are assigned to the same pipeline stage. Therefore, operations assigned to the same stage can be executed in random order and the scheduling can be regarded on the level of pipeline stages. This means that these operations are partially ordered [45]. The ordering is completed by randomly scheduling all operations that are executed in the same cycle.

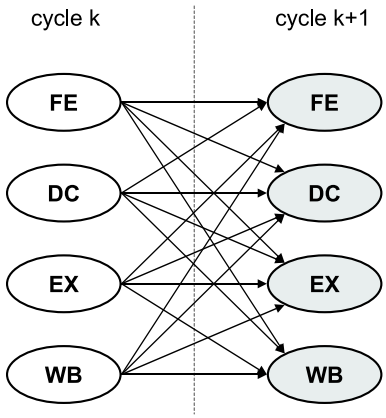

Figure 4.2.6: Precedence constraints between two cycles

Since all pipeline stages operate in parallel, the precedence constraints only exist between two cycles as illustrated in Figure 4.2.6 for a four-stage pipeline. Due to the parallelism no constraints exist between pipeline stages within the same cycle. Because of the flow of instructions that are shifted downstream through the pipeline, the required number of buffers can be minimized by executing the pipelines from the last stage to the first. The buffers are composed of intermediate values and the pipeline registers between the stages. Execution from the rear to the front of the pipeline makes sure that the contents of the pipeline register are read before they are overwritten by the preceding stage. Figure 4.2.7 shows the resulting schedule.

Phase-Accurate

If pipelines are described in phase-accurately, the operations cannot be scheduled simply from the rear of the pipeline to the front. The reason are the precedence constraints between the phases. Figure 4.2.8 shows the constraints for a four-stage pipeline that is subdivided into two phases *read* and *write*.

To all pipeline stages applies that the read phase must precede the write phase and vice versa at

Control Step	Schedule			
k	WB	EX	DC	FE
k+1	WB	EX	DC	FE
k+2	WB	EX	DC	FE
k+3	WB	EX	DC	FE
...	...	...	...	...

Figure 4.2.7: Operation schedule based on pipeline stages

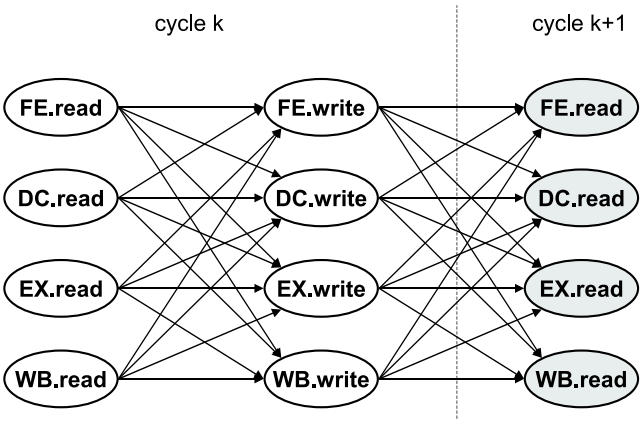

Figure 4.2.8: Precedence constraints between two phases

the cycle boundary. This means that one phase must be finished for all pipeline stages, before operations of the next phase may execute. In addition to that, the constraints between stages as shown in 4.2.6 still apply.

Due to the precedence constraints the execution of phases must start with the first phase (the read phase in our example pipeline). The other phases must be scheduled in their logical order. The scheduling within a pipeline phase can be performed in analogy to the case above that disregards phases – the buffers can be minimized with a schedule that starts with the last stage. The resulting schedule is depicted in Figure 4.2.9. First, the stages are executed from the rear to the front for the *read* phase and second the *write* phase is executed in the same order.

Control Step	Schedule							
k	WB.read	EX.read	DC.read	FE.read	WB.write	EX.write	DC.write	FE.write
k+1	WB.read	EX.read	DC.read	FE.read	WB.write	EX.write	DC.write	FE.write
k+2	WB.read	EX.read	DC.read	FE.read	WB.write	EX.write	DC.write	FE.write
k+3	WB.read	EX.read	DC.read	FE.read	WB.write	EX.write	DC.write	FE.write
...	...	...	...	...	...	...	...	...

Figure 4.2.9: Operation schedule based on pipeline phases

4.2.6 Resource Accesses

As defined in earlier sections, all operations that are executed in the same control step must be scheduled appropriately. Operation execution begins with the last stage and continues in the reverse order of pipeline stages. This order avoids resource conflicts which arise from read and write accesses to the same resource in one control step. As opposed to the pipeline hazards described in [33], these conflicts can be caused in the simulation by the sequential execution of operations that are parallel events in the processor hardware. In synchronous hardware circuits, resources like registers are read in the beginning of the cycle and written in the end of the cycle.

However, the schedule of operations that are assigned to the same pipeline stage cannot be predicted in the model. If these operations read from and write to the same registers within one control step, resource conflicts are produced. The conflicts can be avoided by generally buffering all write accesses, but this approach introduces unnecessary overhead caused by the necessary update operation at the end of each control step that copies the buffered values of all registers to the designated register. The overhead of this copy operation can be reduced significantly, if only those registers are updated that are actually written. The LISA language provides the access attributes to identify the respective registers (cf. section 3.6.2). All write accesses to resources that are listed as outputs of the operation are buffered and the actual update is performed at the end of the control step. All other write accesses are performed immediately to the resource to speed up the simulation.

4.3 Simulator Generation

Retargeting a compiled simulator involves two generation processes. In the first step, the generic model is converted into a processor specific model. Based on the generic machine model and the processor description, the processor specific simulation compiler is produced (see Figure 4.3.1).

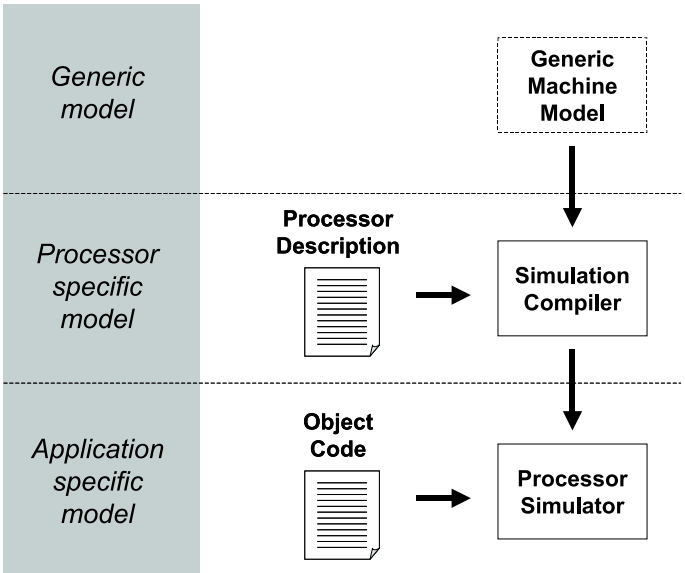

Figure 4.3.1: Retargeting a compiled simulator

Compiled simulators must be generated for each given application program and they are therefore

application specific. Consequently, the processor specific model must be converted into an application specific model in the second step. The simulation compiler takes the object code as input and produces the actual application specific processor simulator. In the following sections the two generation steps will be presented.

4.4 Generating the Processor Specific Model

The conversion of the generic model and the processor description into the processor specific model is performed by means of the LISA compiler. It produces the processor specific simulation compiler and the simulation library. The LISA compiler consists of a front-end and a back-end (see Figure 4.4.1).

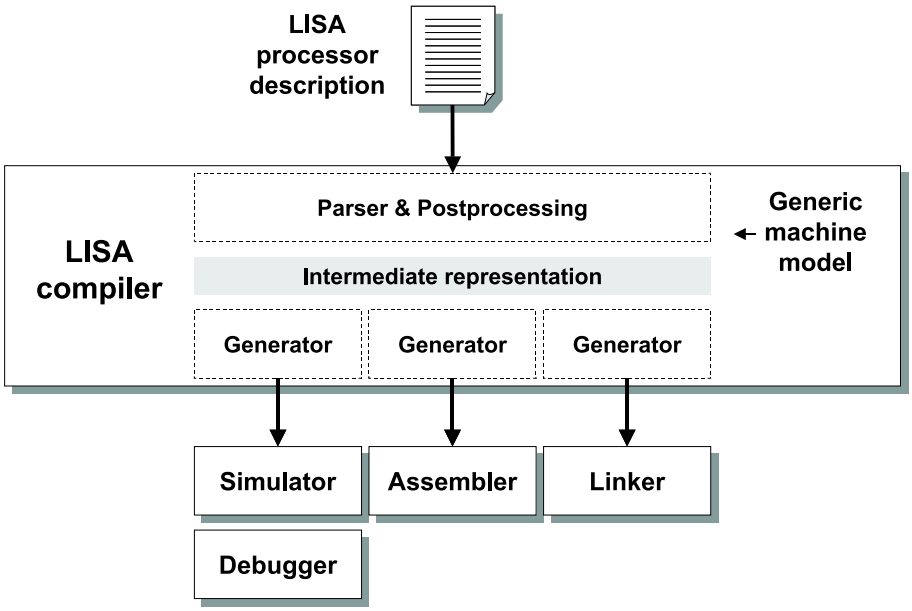

Figure 4.4.1: LISA compiler and tools generation

The front-end is the language parser that reads the processor descriptions and translates it into the intermediate model representation. It is constructed by means of a lexical scanner and parser generator [79, 77] using the BNF description of the grammar for the LISA language that is provided in the appendix B. In the post-processing phase of the LISA compiler front-end, the generic machine model is parameterized with the processor-specific pipelines and the consistency and completeness of the description is checked, i.e. especially the different operations trees for the coding, syntax and behavior sections as well as the declared relations between the operations. Furthermore, the expression sections are distinguished between lvalues and rvalues. Lvalues are those expression sections that can represent the left-hand side of assignments in the behavioral model. With respect to the processor model, lvalues are always resources while rvalues represent constants and numeric expressions.

The back-end comprises several generators which produce the software development tools: The simulator components, debugger information, assembler and linker. The back-end for the assembler generation was implemented within the scope of a diploma thesis [60]. In the following

sections, we will focus on the generation of the simulator components and the debugger. The generated simulator components consist of the simulation library and the simulation compiler. Both parts are used to compose the application specific, table-driven simulator as depicted in Figure 4.4.2.

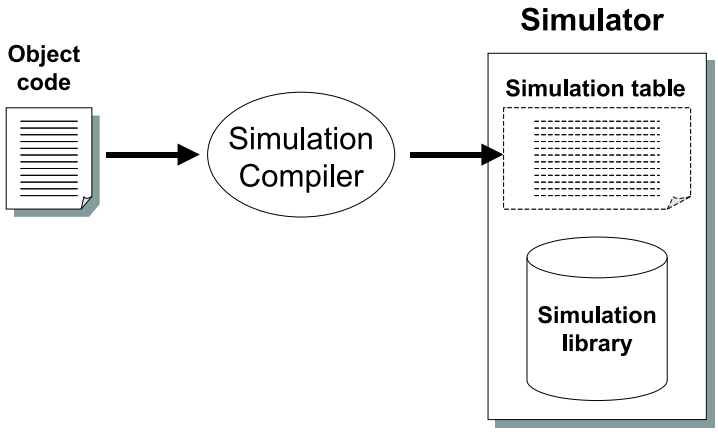

Figure 4.4.2: Compiled simulator

4.4.1 Simulation Library Generation

The simulation library contains the application-independent part of the processor model. In particular, it consists of the memory and resource model, behavioral model, timing model and simulator interface which will be described in this section.

Memory and Resource Model

The memory model is completely generated from the set of elements declared in the resource section of the processor description. All simple resources are converted into (state) variables. The registers based on the scalable data type *bit* instantiate C++ objects of the generic integer type *xbit* that is part of the LISA simulation library [59]. The pipeline register declarations produce multiple instances of the respective variables according to the number of pipeline stages. The pipeline declarations are used to parameterize the generic machine model and to produce the *structure* of the simulation tables. The principle structure of the simulation tables is shown in Figure 4.4.3.

Address	Instruction word	Function	Operand	Operand	Function	...
0x00F000	D40368C1	&Load	&a[5]	0x54FD	&MAccess	...
0x00F001	040523BA	&Add	&b[12]	0	0	...
0x00F002	1BF0C720	&Multiply	&mreg	2	&ChkFlag	...
...	...	...	...	...	...	...

Figure 4.4.3: Simulation table

The simulation tables are partially evaluated representations of the program memories. It contains the decoded information generated from the respective application code. For each address of initialized program memory, one row in the simulation table is produced. This means that multiple simulation tables are required for processors with multiple address spaces. The contents of the table are structured as follows. The first column contains the address of the respective memory location followed by the value of the instruction word in the second column. The remaining columns contain three types of elements:

- Addresses of state update *functions* that are references to the behavioral model, like the columns three and six of the table shown in Figure 4.4.3.
- *Operands references* that point to elements of the memory model providing the addresses of resources like registers, memories, etc. These elements are lvalues that parameterize the state update functions (see column four of the table in Figure 4.4.3).
- *Immediate operands* are numeric constants or expressions – rvalues – that directly parameterize the state update functions (see column five).

For each group declared in the processor description, a separate column is generated that provides a pointer to the target object of the decoding operation or immediately the object.

Behavioral Model

The behavioral model is generated from the C code that can be extracted from the behavior sections in the LISA operations. The behavior section of each operation is translated into a separate C function that can be executed to update the state of the processor model as depicted in Figure 4.4.4. Here the behavior code of the LISA operation *add* is translated to a C-function with the same name. The C-code provided in the behavior sections of LISA operations must be translated to executable C-code of the state update functions because the identifiers of groups that reference operands and other LISA operations must be replaced by legal symbols of the simulator program. The identifiers of these operands and operations (resp. state update functions) are converted into pointers aiming at specific positions (columns) in the simulation table structure.

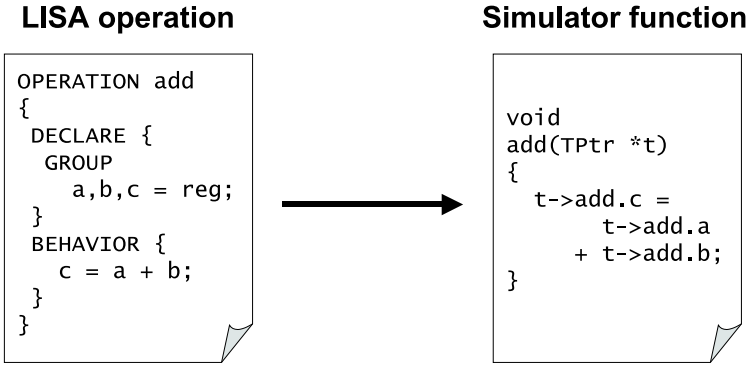

Figure 4.4.4: Translation of behavior sections to state update functions

This means that the behavior code addresses the operands indirectly through the reference provided in the simulation table. The respective set of operands is chosen by assigning a particular address

(row) of the simulation table. In effect, each row of the simulation table parameterizes the set of operations that compose the execution of one instruction. The reference to this particular row of the simulation table is provided by a pointer which is the only parameter that is passed to the state update function. In Figure 4.4.4, the parameter has the data type *TPtr** that represents on row of the simulation table. The three operands *a*, *b* and *c* of the operation *add* are operands representing group identifiers that are replaced by pointers to the respective field in the simulator table structure using the base pointer that is passed to the state update function.

Similar to the operands, the state update functions are also called indirectly using the simulation table (see columns three and six of the table in Figure 4.4.3). As discussed in section 3.6.1, the behavior code of the LISA operations cannot have parameters. This makes the construction of the simulation table much less complex because all functions generated from the behavior sections have the same parameter list. The crucial advantage is that only one data type (a function pointer) is necessary to implement the indirect calls to the state update functions. For the same reason, these functions do not provide return values.

The simulation table captures the results of all expressions in the model that can be evaluated at compile-time. In the LISA language there are two mechanisms that produce such expressions: Group declarations and compile-time statements. In the context of the behavioral model, group declarations list multiple alternative behavior sections. Since the behavior code is translated to the equivalent set of simulator functions as depicted in Figure 4.4.5, the simulation table provides a pointer to the respective simulator function that is selected from the decoding operation of the current instruction.

The compile-time statements IF-ELSE and SWITCH-CASE in the processor description are production rules that enclose multiple behavior sections. Operations can comprise concatenations and encapsulations of multiple compile-time statements. At instruction decoding, the expressions in the compile-time statements are evaluated. Depending on the result of decoding operations, different compositions of behavior sections are produced. In order to cover all decoding results, separate state update functions for all possible combinations of behavior code must be generated. This mechanism is illustrated for a simple example in Figure 4.4.5. Here, the operation *add* contains a compile-time SWITCH that branches into three cases *M1*, *M2* and *M3*.

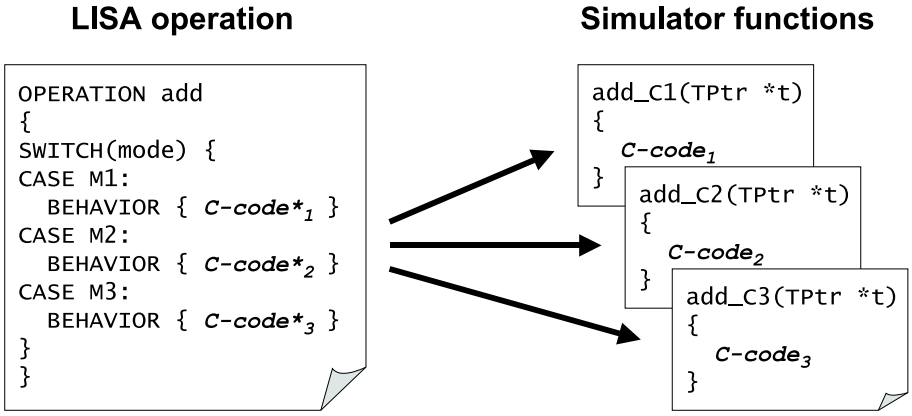

Figure 4.4.5: Translation of behavior sections with compile-time statements

Instead of a single state update functions three functions are generated here. The function names are extended by generated suffixes *__C1..3* to distinguish the three possible productions in this

example. The complexity of compile-time statements found in real LISA descriptions is typically much higher than this simple example. Through concatenation and encapsulation of compile-time statements, more than 100 different cases are produced for a single operation in the model of the Analog Devices ADSP21xx DSP.

Timing Model

The generated simulator components of the timing model consist of the activation tables (cf. section 4.2.1), the pipeline control functions (cf. section 4.2.4) and the operation scheduler that are parameterized by the pipelines declared in the processor model. The operation scheduler is activated from the particular operation *main* that is the controlling instance of the whole simulation. The main operation is executed every control step before any other operation and calls the operation scheduler by means of the pipeline control function *execute()*. The operation scheduler executes all operations that have reached their designated pipeline stage or phase as described in section 4.2.5.

In general, the activation sections are translated into C-code that enters the respective function names in the appropriate field of the activation table. Only those activations that refer to operations which are not assigned to the same pipeline or not assigned to any pipeline are treated differently. In this case, the activations are translated to ordinary functions calls because they are expected to execute in the same control step.

Simulator Interface

The simulator generated by the LISA compiler can be translated by the C-compiler to a stand alone executable program or a dynamically linked library object. The stand-alone simulator can be used with ordinary source-level debuggers that are available on the host computing platform. This was the approach used in [98]. In a retargetable simulation environment, it has the disadvantage that long registers that exceed the bit widths of the native data types on the simulation host platform cannot be displayed. In order to display such data, a specific debugger is necessary. Furthermore, the overhead of simulation loop unfolding (cf. section 4.1) shall be avoided in this work. This approach also requires a specific debugger. Since the debugger is application independent, it is useful to link the application specific, compiled simulation object dynamically as depicted in Figure 4.4.6.

The simulator is controlled by means of an application programming interface (API) which allows to advance the simulation and to obtain or manipulate the state of the processor model. The API provides the following types of functions that are completely listed in appendix C:

- Identification of the processor target,
- initializing the processor model (reset),
- advancing the simulation based on control steps,
- setting and clearing breakpoints and
- retrieving and setting the values of registers and memories.

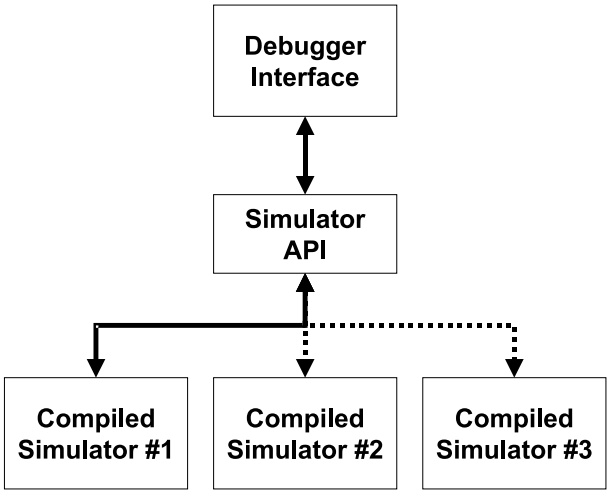

Figure 4.4.6: Dynamically linked simulator controlled by the API

4.4.2 Simulation Compiler Generation

The simulation compiler has to perform two tasks: The generation of the simulation table and the disassembler listing of the program that is displayed in the debugger (cf. section 4.4.3). The simulation compiler takes the executable object code of the target processor as input in order to produce the this output. The translation rules between the input and output of the simulation compiler are provided by the operation trees of the processor description. The instructions of the object code are recognized by means of the binary representation in the coding sections. The output patterns are defined by the elements of the behavioral model and the syntax sections for the simulation table and the disassembler listing respectively.

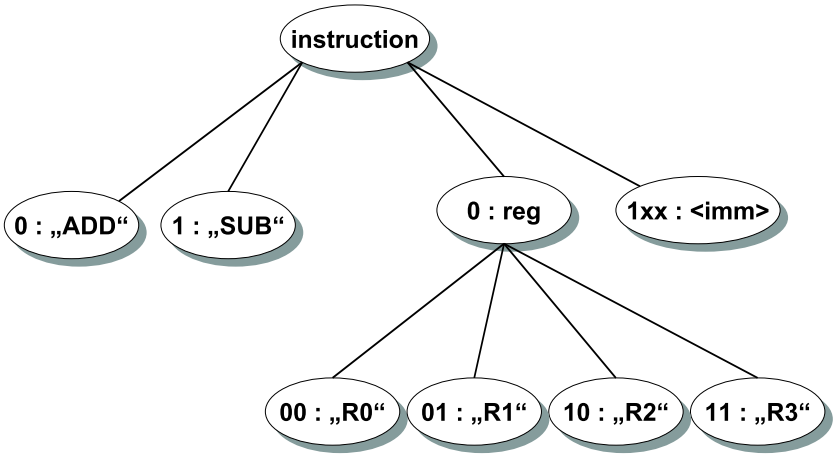

Figure 4.4.7: Coding tree

In order to perform the translation, a search through a decision tree must be performed. The starting point for this search is the coding root operation. In Figure 4.4.7 this is the operation *insn*. It consists of two or-rules that select the respective instruction (*ADD* or *SUB*) and the type of operand (*reg* or *imm*). The operation of searching the appropriate operations assigned to a certain binary instruction word is called decoding. The alternative binary patterns for the decision tree are provided by the coding fields of the alternative elements of LISA operation groups. Once the appropriate

elements in the operation groups are selected, the output of disassembly by using the corresponding syntax sections is quite straightforward. For example in the tree shown in Figure 4.4.7, the binary pattern "1010" translates into the disassembly "SUB R2". The table entries are produced differently for the elements of behavior and expression sections. In case of behavior sections, the operation name is entered in the table which is the name of the corresponding state update function. The expression sections however specify either resources of the model or immediate operands. Thus, the main problem of the translation is the decoding.

The decoding starts at the coding root operation of a LISA processor description. It provides the initial coding pattern C_0 that is used for the comparison with all bits of the actual instruction word $I = i_0..i_n$ that shall be decoded. The coding root operation provides a concatenation of coding fields

$$C_0 = c_1 + c_2 + ...c_n$$

where all non-terminal coding fields are roots of subtrees which in turn may contain further subtrees $C_1, C_2, ..., C_m$. According to the position and the size of the respective bit fields in the subtrees, the instruction word is broken into increasingly smaller parts for the comparison. The decision tree is produced from the coding tree by generating the decision function

$$D_k = \{I_k = C_k\}$$

for each node $k = 0..m$ starting with the coding root operation. The decision function evaluates to 1 if the subset of bits $i_{k,0}..i_{k,n}$ in the instruction word equals the bits described in the coding section of the operation under test.

The simulation compiler performs this tree search for each instruction that must be translated by successively evaluating the decision functions. After finishing the search, all necessary group elements are selected and the output can be produced. The compile-time statements of all selected operations are now evaluated from the results of the search and the respective syntax section and behavior/expression section is chosen in order to generate the disassembly and one row of the simulation table.

4.4.3 Retargetable Debugger

The debugger is the graphical user interface that allows it to the designer to control simulation execution and the observation and modification of the state of the processor model – the registers and memories. Due to the specific register sets, memory configurations and pipelines, the debugger must be adapted to the processor target. The currently available debuggers for embedded processors are currently only available in form of target-specific tools that are incorporating the processor simulator as well. These tools are parts of the software development tools provided by semiconductor vendors for the respective device. Debuggers for high-level languages like gdb [80] or ddd [78] do no provide target-specific information, like the register set. Using this type of debugger for the compiled simulation requires to design an interface specifically for the target processor that extracts the state from the model and that controls simulation execution. Since the simulation loop unfolding is essential for their use with simulators in general, this option cannot be used for our work. Therefore, we have developed a retargetable debugger that is adapted to the target processor as depicted in Figure 4.4.8.

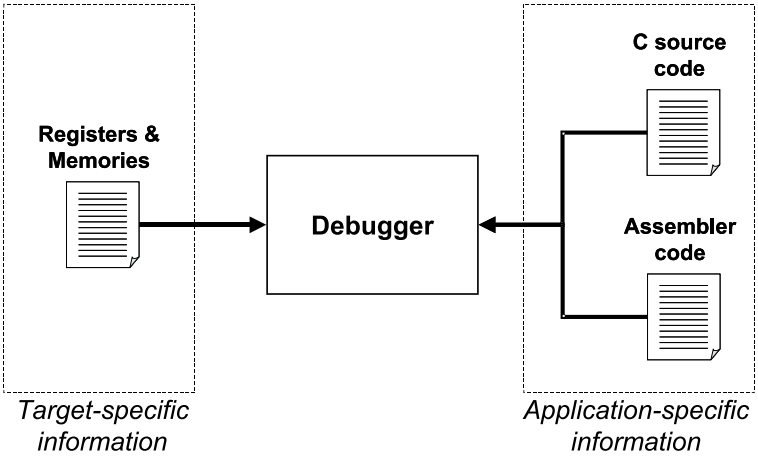

Figure 4.4.8: Debugger retargeting

With each model compilation, a specific interface is generated that provides access to the register sets with the individual data types, bit widths (the scalable data types) and assignment to certain register groups such as arithmetic-, control- or pipeline registers. The register groups are assigned to the appropriate register windows. The same applies to the memory configuration. For each memory structure, a separate window is generated.

Finally, the reference pipeline stage must be determined. This is the stage that is considered to execute instructions from the software designers perspective. This is typically the stage of the functional units. Although the instruction execution is split into several parts, in most processors the reference stage plays a particular role. In this stage the actual arithmetical function of the instruction is computed, conditions are evaluated, flags are set and most registers are accessed.

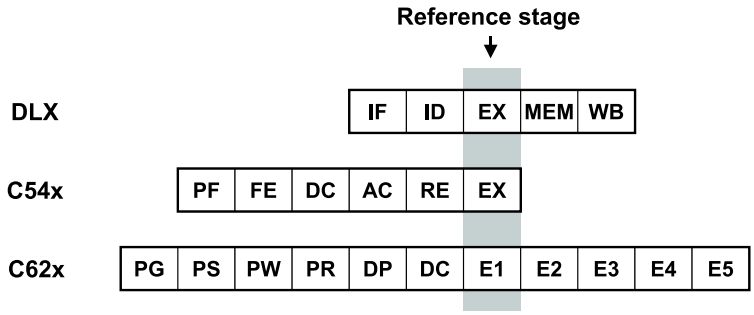

Figure 4.4.9: Reference pipeline stage

Program execution and the full processor state can be observed by loading the application program code and a dynamically linked library object which contains the compiled simulator for the application. Figure 4.4.10 depicts a screen-shot of the debugger.

The debugger controls simulation execution through an API to step through the program, run freely, handle breakpoints, and read and write registers and memory contents. The C source-level program, the assembly program or the numeric representation of the program memory is displayed. Furthermore, the debugger allows to collect detailed profiling data in order to observe operation activity, use of address spaces, and resource accesses such as registers, memories and buses.

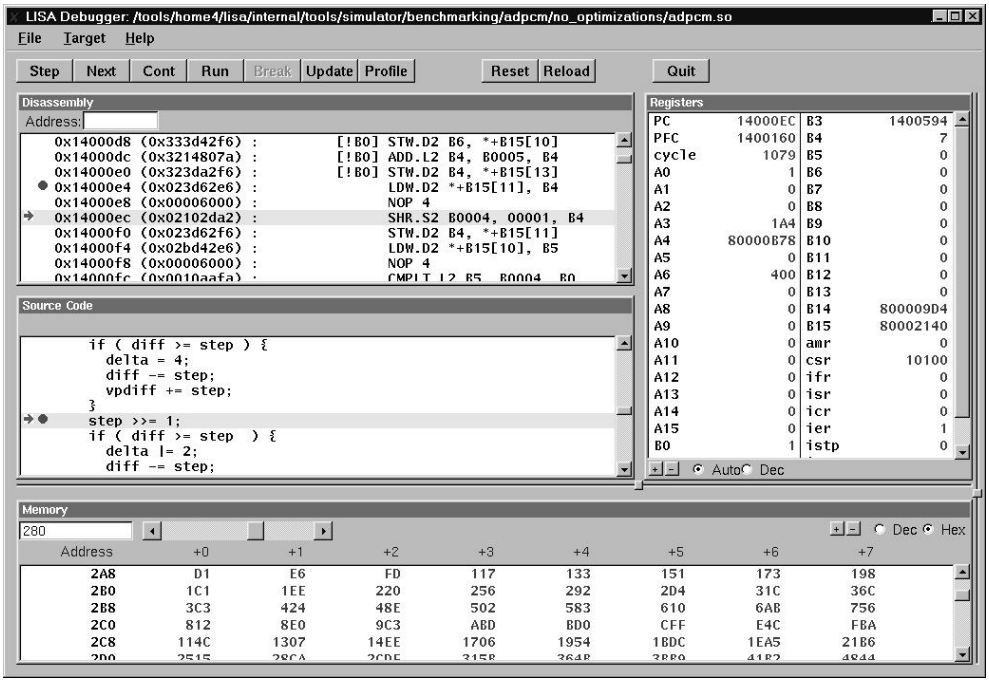

Figure 4.4.10: Target-independent debugger

Chapter 5

Case Studies

This chapter presents case studies of three processors that have been described with LISA. The LISA compiler was used to generate assembler, linker and simulator for these processor models. Based on application programs for the respective processor, the simulation speed is determined and compared to commercial processor simulators implementations that use interpretive simulation techniques. Furthermore the model compilation time was measured.

5.1 Processor Models and Modeling Efficiency

Several processor models have been completely described in LISA to explore the applicability of the LISA language for modeling different processor architectures and generating compiled simulators. Table 5.1 lists processors that have been successfully described. Because of the processor complexity, the different accuracy levels and significant differences in the designer's modeling experience, the model design time and numbers of lines required for the LISA description vary substantially.

Processor	Accuracy level	Pipeline	LISA description	Design time
DLX	Cycle-based	5 stages	37 kB	1 week
ADSP 21xx	Instruction-set	–	144 kB	6 weeks
TI C54x	Cycle-based	6 stages	303 kB	8 weeks
TI C62x	Cycle-based	11 stages	321 kB	6 weeks

Table 5.1: LISA processor descriptions

The first model is the DLX processor described in [33] without the floating-point pipeline. The model of the DLX is cycle-based covering the five-stage pipeline including interlocking and register forwarding mechanisms. Since the arithmetic can be directly mapped onto the data types of C, the behavioral model is quite simple. The very orthogonal instruction set of this academic RISC processor results in a considerable small LISA description that comprises 37 kBytes of code which was built by an experienced designer in less than one week.

The second example is the model of the Analog Devices AD21xx DSP. It describes the processor instruction-accurately. This is a processor with 16bit arithmetic and three functional units. To the programmer, no pipeline is visible. Two undergraduate students without specific knowledge on DSPs or the LISA language worked both for three weeks on the model implementation [57, 36]. In comparison, the implementation of a custom compiled simulator for this DSP processor during a diploma thesis took more than 18 weeks [69] because the model and the specific simulation compiler had to be implemented (see Table 5.2). This means that designer's efficiency using the LISA methodology is three times higher in this case and in addition, it provides the automatic generation of assembler and linker. Furthermore, changes to the custom simulator implementation are very difficult due to the compact instruction word coding.

Processor	Accuracy level	Pipeline	Source code	Design time
ADSP 21xx	Instruction-set	–	112 kB Processor model 114 kB Simulation compiler	18 weeks
TI C54x	Cycle-based	6 stages	kB Processor model kB Simulation compiler	58 weeks

Table 5.2: Custom compiled simulator implementations

The third model listed in Table 5.1 is the TMS320 C54x DSP of Texas Instruments. This processor has a six-stage pipeline and 16bit arithmetic. The implementation of the cycle-accurate model was implemented within eight weeks as the result of a diploma thesis [12]. In an other project, the custom implementation of a compiled simulator with the same accuracy for this processor took 58 weeks [75]. This means that the design effort for this complex model could be reduced by 86%!

The most complex processor architecture described with LISA so far is the TMS320C62x DSP of Texas Instruments. It is a processor with 32bit arithmetic. This processor has two pipelines that are concatenated to eleven stages. It was modelled cycle-accurately within six weeks [70].

The design times listed in Tables 5.1 and 5.2 do not include the time required for verification of the model. It is only the time required to describe the processor using the given documentation. The verification procedures of the processor models have been very different because of the availability and type of reference models. In case of the C54x, the source code of the commercial simulator of Texas Instruments was available while the implementation of the AD21xx model was made based on the printed processor documentation provided by Analog Devices [2]. Furthermore, some simulators provided by the semiconductor vendors allow it to produce traces and state dumps which significantly accelerates the verification process. If this is not possible, the comparison of the generated simulator to the reference simulator is much more costly with respect to the resulting design cycle.

From these processor models described in LISA, the DLX, the Analog Devices ADSP21xx and the Texas Instruments C62x will be examined in more detail providing the experimental analysis of the achievable simulation speed and comparison to currently available simulation technology. Furthermore, the model and simulation compilation times will be discussed for these processors in the next sections. The LISA descriptions of the DLX and C62x models are provided in E and F.

5.1.1 Measurement Conditions

For the following measurements we used a Sun Ultra Sparc 10 with a 333 MHz Ultra Sparc IIi CPU, 256 MB memory and 2 MB cache. The host was part of the networking system and runs under the operating system Solaris 2.7. The execution times were measured using the UNIX command *time*, that distinguishes between real elapsed time for the process, the user-time which is the actual processing time the job spends on the CPU and time spent on operating system calls. The measurement results presented here always refer to the sum of the user and system time.

The compiler chosen for the compilation of the files comprised in the simulation library and of the simulation-table was the GNU C++-compiler *g++*, v2.91.60. The GNU tools are generally practical to use, since they are available on nearly all platforms (Solaris, Linux, Windows NT, etc.) and thus make it easy to port code from one platform to the next.

To eliminate the influence of different IO and CPU loads on the benchmark results, every simulation is run 10 times. Out of the resulting 10 values for simulation runtime, the shortest time is chosen.

5.2 Model Compilation

If a processor description is created or changed, the model must be compiled to generate the tools for the respective architecture. First, the LISA compiler reads the processor description and generates the C++ source code of the tools. In a second step the generated tools must be translated by means of a C++ compiler to produce the libraries and executable programs. Here, we present the compilation time required to generate the retargetable simulator consisting of the simulator libraries and the simulation compiler.

Processor Model	Compilation time
DLX	24 s
ADSP 21xx	58 s
TI C62x	67 s

Table 5.3: Model compilation time

Table 5.3 lists the model compilation times for the three models with different complexity. Although the description of the C62x is nearly ten times larger than the description of the DLX, the compilation time varies much less. One reason for this behavior is that fact that the complexity of the decoder – and thus its generation time – does not necessarily correspond to the size of the model. However, the compilation time is certainly short enough for to enable short design iterations and to fulfill the requirements of architecture exploration. The measured times are very typical numbers known for the compilation of small software projects.

5.3 Application Programs

The simulation speed of the processor models was determined based on four application programs that differ in size and complexity. Table 5.4 list the application programs and their implementation.

Application program	Implementation
FIR filter	Assembler
FIR filter	C
ADPCM encoder/decoder	C
GSM encoder/decoder	C

Table 5.4: Application programs used for the measurements

The first application is a small, typical kernel of DSP software, a 16-tap FIR filter that is hand-written in assembler for the respective processor. The second application program is the same algorithm implemented in C. It was translated with a C compiler for the respective target processor. Both FIR filter programs process random input data. The ADPCM example is a realization of the ADPCM G.721 speech compression standard as defined by the ITU [38]. It consists of the encoder and the decoder that are also implemented in C and process ITU test sequences as input data. The ADPCM algorithm implementation represents an full application with low complexity. The last algorithm tested is a C implementation of the GSM speech compression standard. It consists of the speech encoder and the decoder. This application program is rather complex and requires a lot of program and data memory. It can be used to examine the influence of large application programs on compiled simulation.

Except the GSM encoder/decoder, all applications are wrapped in a loop repeating the execution in order to obtain reasonable run-time that can be measured. Furthermore, this approach makes sure that the initialization phase of the simulation can be neglected which is important to determine the simulation speed.

5.4 Simulation Speed

Simulation speed was quantified by running an application on the respective simulator and relating the simulation time to the processed number of cycles. In case of the Analog Devices model, one cycle corresponds to one instruction, because only internal memory was used to avoid waitstates.

5.4.1 DLX

The measured simulation speed of the DLX simulator is shown in Table 5.5. The highest simulation speed of 1.42 million cycles per second was measured for the C implementation of the FIR filter. The assembler implementation runs slightly slower. Even the slowest application, the GSM speech compression runs at a speed of nearly 1.09 million cycles per second.

Application program	Simulation speed
FIR filter (asm)	1.34 M cycles/s
FIR filter (C)	1.42 M cycles/s
ADPCM encoder/decoder	1.10 M cycles/s
GSM encoder/decoder	1.09 M cycles/s

Table 5.5: DLX simulation speed

5.4.2 Analog Devices ADSP 21xx

The comparison of the simulation speeds between the compiled simulator generated from the LISA environment and the interpretive simulator *xsim2101* provided by Analog Devices are shown in Table 5.6 for three application examples. The GSM program was too large to be loaded into the simulator *xsim2101* and could not be used for this comparison.

Application program	LISA simulator	AD simulator	Speed-up
FIR filter (asm)	1.79 M cycles/s	14.13 k cycles/s	127x
FIR filter (C)	4.85 M cycles/s	31.89 k cycles/s	152x
ADPCM encoder/decoder	4.01 M cycles/s	28.65 k cycles/s	140x

Table 5.6: ADSP 21xx simulation speed comparison

The compiled LISA simulator runs at speed between 1.79 and 4.85 millions of instruction per second. Compared to the interpretive simulator, this corresponds to speed-up factors between 127 to 152 times fast simulation. The assembler implementation of the FIR filter shows the lowest speed at both simulators. It can be assumed that the high density of operations is responsible for this behavior. In the hand-optimized assembler code for this processor the kernel operations can be realized by a single instruction that performs multiply and accumulate operations. This single instruction is executed in a tight loop to calculate the filter algorithm. This density of operations is not achieved by the C compiled applications. The LISA simulator is even more affected by this effect than the simulator of Analog Devices.

Another interesting comparison can be made between the compiled simulator generated from the LISA tools and the earlier custom implementation of the compiled simulator SuperSim for this processor [69]. Unlike the approach of this work, the SuperSim simulator of this target implements the highest level of compiled simulation, the compile-time instantiation.

Application program	LISA simulator	SuperSim simulator	Speed ratio
FIR filter (asm)	1.79 M cycles/s	2.36 M cycles/s	0.76x
FIR filter (C)	4.85 M cycles/s	14.72 M cycles/s	0.33x
ADPCM encoder/decoder	4.01 M cycles/s	4.29 M cycles/s	0.93x

Table 5.7: Speed of compiled simulators of the ADSP 21xx

Interestingly, the simulation speed of the generated simulator achieves speeds that range in the same order of magnitude as the SuperSim simulator. The highest difference is measured for the C

implementation of the FIR filter. Here, the SuperSim shows remarkable 14.7 million instructions per second simulation speed. This value could be achieved through an efficient simulation of data transport instructions such as load, store and move instructions. An analysis shows that the FIR filter generated by the C compiled consists to large extend of this type of instructions. For the ADPCM, the LISA simulator runs at nearly the same simulation speed as SuperSim. A possible reason for this performance is the simulation size. The SuperSim for this application consists of a relatively large simulation executable program which is not very suitable for the cached architecture of the host computer platform.

5.4.3 TI C62x

The reference simulator *sim62x* from Texas Instruments achieves between 1.6k and 11.9k cycles/s whereas our generated compiled simulator runs with speeds between 228k and 437k cycles/s at the same accuracy level. These boundary values are found for two implementations of the FIR filter kernel. For full applications – the ADPCM and the GSM speech compression encoder/decoder pair – simulation speeds in the range of 250k to 300k instructions/s are achieved (see Table 5.8).

Application program	LISA simulator	TI simulator	Speed-up
FIR filter (asm)	437.4 k cycles/s	11.90 k cycles/s	36.8
FIR filter (C)	228.9 k cycles/s	3.22 k cycles/s	71.0
ADPCM encoder/decoder	259.8 k cycles/s	3.21 k cycles/s	80.8
GSM encoder/decoder	287.6 k cycles/s	1.69 k cycles/s	169.9

Table 5.8: C62x simulation speed comparison

This performance results in a speed-up of 36.8 to nearly 170 times faster simulation provided by the LISA simulator. The extraordinary speed-up of the GSM codec is caused by the low performance of the reference simulator for this large application program that nearly fills the complete internal program and data memory space.

Detailed analysis

In order to investigate the simulator performance and to discover potential for speed improvements, the time shares of different parts of the simulator functionality shall be further analyzed. For this purpose, the LISA simulator was compiled to produce profiling information of the activity of its C/C++ functions while it is running. Figure 5.4.1 shows the 20 most active functions of the simulator for the ADPCM application. The operation *move_on_one_step* which is executed in each clock cycle dominates the histogram. This function implements that operation *main* of the processor description. This operation contains the control mechanisms of the two pipelines and the interrupt detection. The second bar represents the pipeline operation *execute* which is followed by the operation *Decode* of the LISA description.

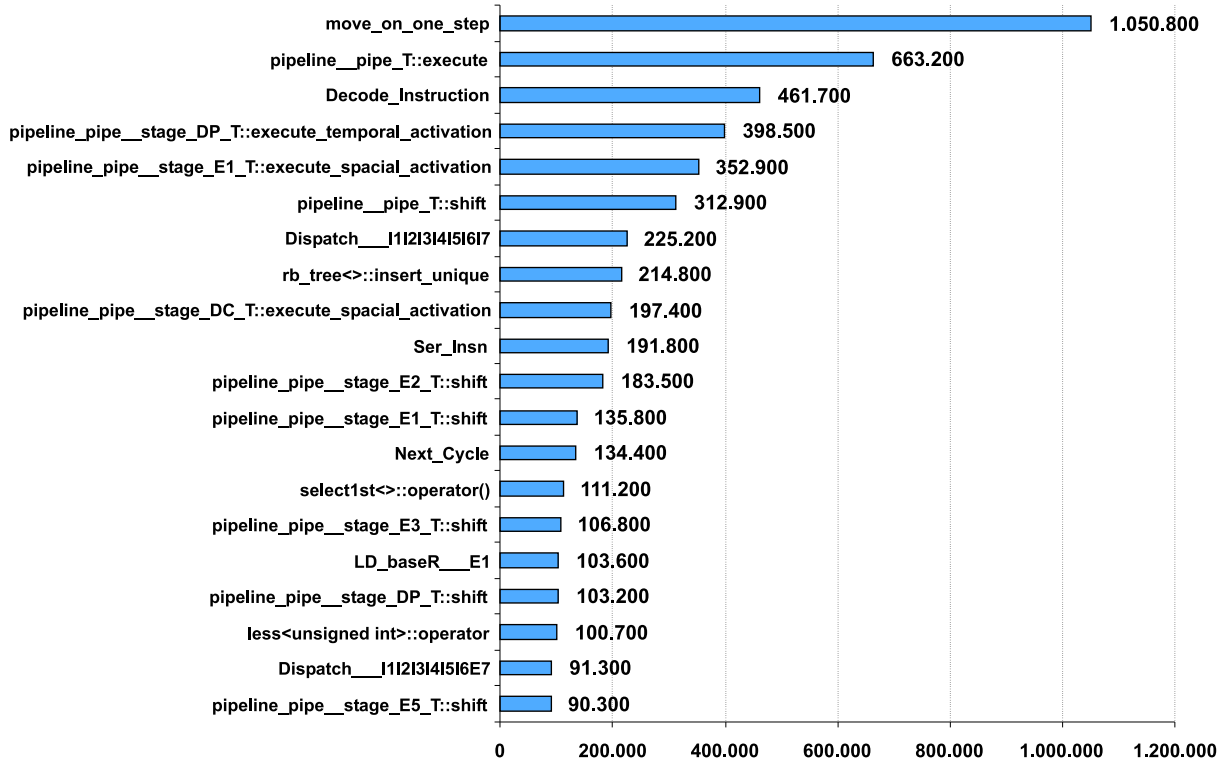

Figure 5.4.1: Execution time histogram of the C62x LISA simulator measured in milliseconds

Due to the high number of functions implemented in the LISA simulator it is useful to group all functions logically and display the shares of the total simulation time in a chart. Figure 5.4.2 displays the shares of the simulator functionality summarized to groups of C/C++ functions:

- Fetch covers the operations of the first four stages of the C6201 pipeline.
- Dispatch describes the operation *Dispatch* and related operations associated to the dispatch stage.
- Decode represents the operation *Decode*.
- E1 through E5 stage summarize all operations associated to the respective pipeline stage. This includes all instructions which are mainly covered in the E1 stage.
- Helper functions - functions implementing memory access and addressing modes, i.e. operations of the external memory interface, data memory controller, and others.
- Cyclic functions are all functions that are activated in each cycle, i.e. the operation *main*, *move_on_one_step*, interrupt processing, etc.
- The other functions represent pipeline functions of the generic processor model.

A remarkable result is that about half of the simulator run time is spent on the management of the pipeline operations. The *cyclic functions* account for a third of the remaining part. The functionality of all instructions – as a part of the E1 stage – accounts for less than 8,3 %.

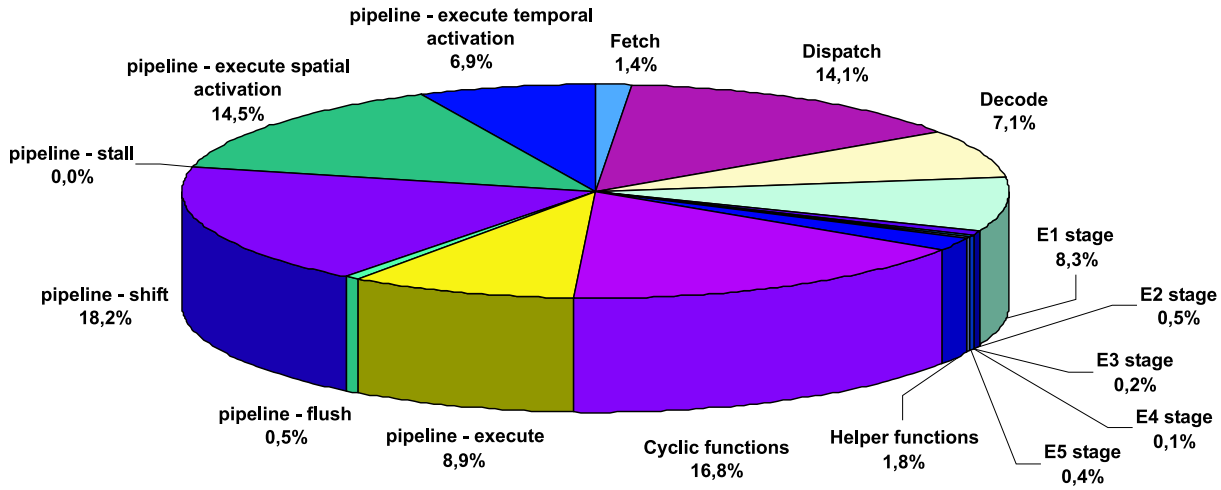

Figure 5.4.2: Shares of the simulator functionality.

It can be concluded that the simulation speed is widely determined by the complexity of the pipeline mechanisms. By comparison, the influence of the actual instruction functionality is marginal.

5.5 Simulation Generation

The first step of generating a simulation is to run the simulation compiler. Since it generates C/C++ source code, simulations have to be compiled with a C++ compiler afterwards. Finally the compiled object must be linked to the simulator. The required time for the simulation compiler, C++ compiler and linker are listed in table 5.9 for our set of application programs based on the three processor models.

Model	Application	Simulation compiler	C compiler	Linker	Total
DLX	FIR (asm)	0.33 s	0.10 s	0.20 s	**0.63 s**
	FIR (C)	0.34 s	0.10 s	0.18 s	**0.62 s**
	ADPCM	0.45 s	0.12 s	0.20 s	**0.77 s**
	GSM	9.01 s	1.14 s	0.54 s	**10.69 s**
AD21xx	FIR (asm)	0.47 s	0.11 s	0.22 s	**0.80 s**
	FIR (C)	0.49 s	0.11 s	0.21 s	**0.89 s**
	ADPCM	0.73 s	0.13 s	0.23 s	**1.09 s**
C62x	FIR (asm)	0.56 s	0.12 s	0.24 s	**0.92 s**
	FIR (C)	0.60 s	0.11 s	0.21 s	**0.92 s**
	ADPCM	0.81 s	0.15 s	0.24 s	**1.20 s**
	GSM	15.72 s	2.20 s	1.00 s	**18.92 s**

Table 5.9: Generation, compilation and linkage time

It can be observed that the generation, compilation and link-times for small applications like the FIR filter and the ADPCM codec are barely measurable for all three processor models. The overall

time for the creation of a compiled simulation does not exceed 1.2 seconds for these applications which can certainly be neglected. Even the time of nearly 19 seconds for the GSM program of the C62x seems acceptable when considering the high simulation speed of the LISA simulator. Already 20 seconds after starting the TI simulator it would be outperformed by the LISA simulator that would have simulated 287k instructions in the 20th second compared to the TI simulator that has simulated 33.8k instructions up to this point.

Nevertheless, it could be explored in future research, how far the translation time can be reduced by integrating the simulation compiler in the debugger front-end and initializing the simulation table directly at run-time instead of producing source code as it is currently done. At least the time required for the compilation and linking will be eliminated by this approach.

5.6 Summary

The methodology of describing processors using the LISA language and generating fast simulators based on these models has been examined in this chapter. Based on two examples it could be shown that modeling efficiency and thus the designer's productivity can be greatly increased when using LISA. The compilation of the processor description into a new set of tools takes only one minute for a processor as complex as the TI C62x. Modeling efficiency and a short retargeting methodology are key components for the exploration of processor architectures.

Furthermore, the compiled simulation technique was successfully implemented and combined with a retargetable simulation environment. The simulators generated in this approach outperform their commercial counterparts by two orders of magnitude while maintaining the same model accuracy. The high simulation speed will further improve the designer's productivity and enable short design iterations due to shortened simulation time. Full application programs can be simulated while processing large test vector sets.

In addition to that, the simulation compilation time could be significantly reduced compared to previous approaches of by using a lower level of compiled simulation.

Chapter 6

Conclusions

Efficient architecture exploration and simulation of embedded processors are key for the development of new technology. The approach based on the LISA language which is presented in this thesis enables such methodologies and furthermore increases the design efficiency and simulation speed by orders of magnitude. Particular requirements of processor modeling in this context are the capabilities to cover a class of processors that includes real DSP and microcontroller architectures and to produce processor models at different abstraction levels in order to trade between simulation speed and accuracy.

The language LISA is introduced that enables the formal description of real embedded processors, their peripherals and interfaces. The development of this new language was necessary because existing languages are focused on either hardware or software properties. LISA uses the form of attributed grammars to combine hardware and software properties of a processor architecture in a joint description. These descriptions allow it to generate software development tools, such as simulators, assemblers and linkers. The behavioral model is completely specified in C which enables the reuse of legacy model components and makes the language intuitive to use. The language is completed by the generic machine model of LISA that is able to produce instruction-accurate, cycle-accurate and phase-accurate models based on control steps that implement a zero-delay model.

Furthermore, the implementation of a retargetable environment is presented that produces compiled simulators from LISA processor descriptions. Using the LISA processor description significantly reduces the efforts of designing software development tools and makes processor specification transparent and understandable to those other than the authors. Complete processor descriptions of real processor architectures are realized to prove the applicability of this approach. The expressive power of LISA is illustrated based on selected sections taken from these descriptions. Compiled simulators are generated from the processor descriptions and the models are successfully verified against our reference models. Measurements show that the simulation speed is up to two orders of magnitude higher than the speed provided by commercial simulators at the same accuracy level. The high simulation speed of the compiled technique helps to shorten the design cycles and improves the productivity of designers.

The approach of formally describing a processor architecture and generating the simulator as part of the development tools is successfully realized. It enables designers of processor architectures and software application code to evaluate their prototypes and reduces significant risks and consis-

tency problems in the development of production-quality simulators.

6.1 Future Research

The presented work shows that fast simulators can be generated from processor descriptions based on the LISA language. Through the joint description of processor hardware and software properties, there is a high potential for the research on further back-end of the LISA compiler. Furthermore, the experience from processor descriptions and the feedback from industry has shown that some aspects of modeling and simulation are topics of further exploration.

6.1.1 Model Verification

While modeling the different processor architectures described in section 5.1, the designers experienced that the verification procedure exceeds by far the design effort of creating the processor description itself. This experience complies with many statements from industry that verification efforts consume typically between 50% - 70% of total design time. Improving this procedure will be an important topic of future research. Especially the development of a systematic test strategy that ensures a certain test coverage will be extremely important.

6.1.2 HW/SW Co-simulation

This work has focused on the generation of fast processor simulators. If one or more processor simulators are integrated in a co-simulation environment with a simulator of hardware components, the overall simulation speed depends on the performance of both type of simulators. Future research should investigate the potential of implementing efficient system-level simulation using abstract hardware models and techniques for efficient simulator coupling [83]. The SystemC initiative is an approach of taking hardware models to higher levels of abstraction and the use of C provides a promising platform for the system-level integration of processor and other hardware components. Suitable interfaces and strategies of synchronizing both parts are open topics that should be explored in the future.

This type of co-simulation requires phase-accurate behavior at the boundary of the processor model. The standard approach currently used in the industry is based on instruction-set simulators that are embedded into bus-interface models (BIM). These BIMs implement complex state machine that emulate the phase-accurate behavior at the outer processor boundary. The development of these BIMs can be avoided by employing phase-accurate processor models that are generated from LISA descriptions.

6.1.3 Memory Model

The language support to describe memories in the presented version of LISA is rather limited. Memories are described as simple arrays that use the direct addressing mode. The initial idea of

LISA was to let the designer implement any desired memory structure and arbitrary addressing modes. However, the vast majority of memory implementations can be categorized into a few different types and the same applies to the addressing modes used. Due to this very limited diversity of memory structures it makes sense to provide support from the LISA language. Having a library of memory structures and addressing modes available would help the designer with the task of exploring a suited memory hierarchy for the processor or system design.

Although caches in real time systems have been not very common in the past, they are increasingly employed to overcome the memory bottleneck [5, 48]. The design methodology for caches in real-time systems has received a lot of attention in recent years [43, 65, 49]. Explicit support for the whole memory hierarchy, addressing modes and cache strategies would be an important improvement of LISA.

A further disadvantage of the current tools implementation is the fact that the model must be recompiled each time the memory configuration in the processor description is changed. This can be avoided by allowing for additional configuration of memory that is external to the processor model. A possible solution for this issue would be an approach of (partially) moving the memory map out of the processor model to build a separate description of the memory interface.

6.1.4 Synthesis

The initial idea of LISA was a machine description language that provides models that are suited to generate processor simulators at different levels of abstraction. The behavioral model of the LISA language is designed to optimally serve the requirements of simulation. In order to avoid redundancy in the description and to optimize simulation speed, the interconnect of components is not specified explicitly. If necessary, the interconnect must be derived from the behavioral model.

Our future work will focus on an efficient verification methodology which compares LISA models against HDL descriptions. The generation of abstract models on the instruction set level (and higher) from a given cycle-based specification is part of our current research.

6.1.5 HLL Compiler

LISA descriptions can cover the programming model consisting of the instruction set, the memory model, register set and other resources. The language is designed to support the automatic generation of simulators, assemblers and linkers. For the architecture exploration of embedded processors like DSPs, compiler design and performance of compiled code is becoming increasingly important [98]. For this reason, the compiler design should also be supported by the formal processor descriptions and generation capabilities. The semantics section was introduced to the LISA language to support the generation of a compiler back-end. Due to the open issues in the design of compilers for DSPs, a generated compiler back-end would probably serve as an implementation prototype that must be optimized manually by compiler designers.

6.1.6 High-level Assembler Languages

In the presented version of LISA, some high-level assembly language constructs are not supported. Before the background of the problems of generating efficient code for DSPs with HLL compilers, some semiconductor vendors use language extensions like DSP-C (NEC, Philips) or intrinsics (Texas Instruments) to lower the level of C to come closer to the assembly level. Other companies approach the problem from the opposite side and rise the level of their assembly language (Infineon). These high-level assembly languages involve loops and blocks which cannot be described with the current version of LISA. Further research could explore and solve these issues.

6.1.7 Power Estimation

The form of LISA based on attributed grammars allows it to extend processor descriptions by further attributes. One example for an extension would be the profiling of power consumption. It has been shown that it is feasible to estimate the power consumption accurately at the instruction-set level by regarding the switching activity of the respective hardware units [55]. Since many embedded systems are battery powered, such estimates would be very beneficial for processor and application software design. The future work on LISA could explore approaches that provide retargetable power estimation on different abstraction levels.

6.1.8 Documentation Generation

The processor descriptions that were created during this project have been developed nearly exclusively based on the written documentation available. The most critical and very difficult to find errors have been caused by inconsistencies between the documentation and the tool implementation. The details of the programming model have been converted by the designers to the respective processor description. It would only be logical to take the reverse approach to generate processor documentation from LISA descriptions using the text section that is included for this purpose. The crucial advantage of the generated documentation would be that it is always up to date and consistent with all other tools generated from the same description.

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Analog Devices. *ADSP-21xx*, 1994.

[3] Analog Devices. *ADSP-TigerSharc001M*, 2000.

[4] G. Araujo, Sudarsanam A., and S. Malik. Instruction set design and optimization for address computation in DSP architectures. In *Proc. of the Int. Symposium on System Synthesis (ISSS)*, 1996.

[5] R. Arnold and F. Mueller *et al.* Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, Dec. 1994.

[6] M. Barbacci. Instruction set processor specifications (ISPS): The notation and its application. *IEEE Transactions on Computers*, C-30(1):24–40, Jan. 1981.

[7] Z. Barzilai and J. Carter *et al.* HSS - A high speed simulator. *IEEE Trans. on CAD*, CAD-6:601–616, July 1987. 1987.

[8] S. Bashford and U. Bieker *et al.* The MIMOLA language. Technical report, University of Dortmund, 1994. version 4.1.

[9] C.G. Bell and A. Newell. *Computer Structures: Readings and Examples*. McGraw Hill, New York, NY, 1971.

[10] P. Bose and T. M. Conte. Performance analysis and its impact on design. *IEEE Computer*, 31(5):41–49, 1998.

[11] D.G. Bradlee, R.E. Henry, and S.J. Eggers. The Marion system for retargetable instruction scheduling. In *Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Canada*, pages 229–240, 1991.

[12] G. Braun. *Examination of the Applicability of Compiled Techniques in Retargetable Processor Simulation*. Integrated Signal Processing Systems (ISS), Aachen University of Technology (RWTH), Aug. 2000. Diploma Thesis D 420.

[13] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[14] Cadence. *Cierto*
http://www.cadence.com/technology/hwsw.

[15] G. De Micheli and M. G. Sami, editors. *Proceedings of NATO Advanced Study Institute on Hardware/Software Co-Design*. Kluwer Academic Publisher, June 19-30 1995.

[16] F. Depuydt. *Register Optimization and Scheduling for Real-Time Digital Signal Processing Architectures*. PhD thesis, PhD thesis, Katholieke Univesiteit Leuven, Oct. 1993.

[17] R. Earnshaw, L. Smith, and K. Welton. Challanges in cross-development. *IEEE Micro*, pages 28–36, July/Aug. 1997.

[18] K. Ebcioglu and J. Fritts *et al.* An eight-issue tree-VLIW processor for dynamic binary translation. In *Proc. of the Int. Conf. on Computer Design (ICCD)*, page 488495, Austin, Oct. 1998.

[19] D. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proc. of the Int. Conf. on Programming Language Design and Implementation (PLDI)*, May 1996.

[20] A. Fauth, M. Freericks, and A. Knoll. Generation of hardware machine models from instruction set descriptions. In *Proc. of the IEEE Workshop on VLSI Signal Processing*, 1993.

[21] A. Fauth and A. Knoll. Automatic generation of DSP program development tools using a machine description formalism. In *Proc. of the ICASSP - Minneapolis, Minn.*, 1993.

[22] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proc. European Design and Test Conf., Paris*, Mar. 1995.

[23] M.S. Flynn. Very High-Speed Computing Systems. *Proceeding of the IEEE*, 54(12):1901–1909, Dec. 1966.

[24] M.S. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9), 1972.

[25] M. Freericks. The nML machine description formalism. Technical Report 1991/15, Technische Universität Berlin, Fachbereich Informatik, Berlin, 1991.

[26] D. Gajski and F. Vahid. Specification and design of embedded hardware-software systems. *IEEE Design & Test of Computers*, spring 1995.

[27] W. Geurts and D. Lanneer *et al.* Design of DSP systems with Chess/Checkers. In *2nd Int. Workshop on Code Generation for Embedded Processors*, Leuven, Mar. 1996.

[28] L. Guerra and J. Fitzner *et al.* Cycle and phase accurate DSP modeling and integration for HW/SW co-verification. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1999.

[29] 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.

[30] A. Halambi and P. Grun *et al.* EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.

[31] M. Hartoog and J. Rowson *et al.* Generation of software tools from processor descriptions for hardware/software codesign. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1997.

[32] J. Hennessy and M. Heinrich. Hardware/software codesign of processors: concepts and examples. In G. De Micheli and M. Sami, editors, *Hardware/Software Co-Design*. Kluwer Academic Publishers, 1995.

[33] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1996. Second Edition.

[34] A. Hoffmann, S. Pees, and H. Meyr. A Retargetable Tool-suite for Exploration of Programmable Architectures in SOC-Design. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, pages 595–599, Orlando, Nov. 1999.

[35] T. Hopes. Hardware/software co-verification, an ip vendors viewpoint. In *Proc. of the Int. Conf. on Computer Design (ICCD)*, 1998.

[36] F.A. Horstmann. *Verhaltensbeschreibung des Digitalen Signalprozessors ADSP2101 von Analog Devices mit der Sprache LISA*. Integrated Signal Processing Systems (ISS), Aachen University of Technology (RWTH), June 1999. Student work.

[37] Infineon. *Carmel DSP Core Technical Overview Handbook*, June 2000.

[38] ITU-CCITT. General aspects of digital transmission systems; technical equipment – blue book. *Recomm. G.700-G.795*, 3(4), 1989.

[39] K. Keutzer. Hardware-Software Co-Design and ESDA. In *Proc. of the Design Automation Conference (DAC)*, pages 435–436, 1999.

[40] P. Klint. Interpretation techniques. *Software – Practice and Experience*, 11(9):963–973, Sep. 1981.

[41] P. Kogge. *The Architecture of Pipelined Computers*. McGraw Hill, New York, NY, 1981.

[42] V. Krishnaswamy and R. Gupta *et al.* A procedure for software sythesis from VHDL models. In *Proc. of the Int. Conf. on Measurement and Modeling of Computer Systems*, Aug. 1999.

[43] M. Lajolo, L. Lavagno, and A. Sangiovanni-Vincentelli. Fast instruction cache simulation strategies in a hardware/software co-design environment. In *Proc. of the Asia South Pacific Design Automation Conference (ASP-DAC)*, 1999.

[44] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, C-36:24–35, Jan. 1987.

[45] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12), December 1998.

[46] R. Leupers and P. Marwedel. Instruction set extraction from programmable structures. In *Proc. EURO-DAC*, 1994.

[47] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1), Jan. 1998.

[48] Y. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. In *Proc. of the Int. Conf. on Computer Aided Design (ICCAD)*, pages 380–387, Nov. 1995.

[49] Y. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, Jan. 1997.

[50] S. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *Conference on Advanced Research in VLSI*, 1995.

[51] F. Löhr, A. Fauth, and M. Freericks. SIGH/SIM - an environment for retargetable instruction set simulation. Technical report, Technische Universität Berlin - Fachbereich 13 - Informatik, 1993.

[52] LSI Logic. *ZSP400 Digital Signal Processor Architecture*, June 2000. Rev. 1.01.

[53] V. Madisetti. *VLSI Digital Signal Processors*. IEEE Press, Butterworth-Heinemann, Newton, MA, 1981.

[54] P. Marwedel. The MIMOLA design system: Tools for the design of digital processors. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1984.

[55] H. Mehta, R.M. Owens, and M.J. Irwin. Instruction-level power profiling. In *Proc. of the Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, June 1996.

[56] Mentor Graphics. *Seamless CVE*
http://www.mentor.com/seamless.

[57] R. Mosig. *Prozessorbeschreibung des ADSP21xx auf Basis der Sprache LISA*. Integrated Signal Processing Systems (ISS), Aachen University of Technology (RWTH), July 1998. Student work.

[58] M. Moudgill. Techniques for implementing fast processor simulators. In *Proceedings of the 31st Annual Simulation Symposium*, Boston, Apr. 1998.

[59] A. Nohl. *Development of a generic integer C++ data type for the use within the machine description language LISA*. Integrated Signal Processing Systems (ISS), Aachen University of Technology (RWTH), Feb. 1999. Student work S 251.

[60] A. Nohl. *Investigation of the retargetability of development tools for embedded processors*. Integrated Signal Processing Systems (ISS), Aachen University of Technology (RWTH), May 2000. Diploma Thesis D 415.

[61] S. Ohr. In dsp development, compilers rule. *EE Times, Issue 1034*, Nov. 1998.

[62] K. Olukotun, M. Heinrich, and D. Ofelt. Digital system simulation: Methodologies and examples. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1998.

[63] P. Paulin, *et al.* FlexWare: A flexible firmware development environment for embedded systems. In P. Marwedel and G. Goosens, editors, *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[64] SystemC Home Page. *http://www.systemc.org*.

[65] P. Panda, N. Dutt, and A. Nicolau. Data cache sizing for embedded processor applications. In *Proc. of the Asia South Pacific Design Automation Conference (ASPDAC)*, 1998.

[66] S. Paul, J. Peffers and D. Thomas. A codesign virtual machine for hierarchical, balanced hardware/software system modeling. In *Proc. of the Design Automation Conference (DAC)*, Jun. 2000.

[67] P. Paulin and C. Liem *et al.* Codesyn: A Retargetable Code Synthesis System. In *Proc. of the 7th Int. Symp. High-Level Synthesis*, Niagra-on-the-Lake, May 1994.

[68] P. Paulin, C. Liem, T. May, and S. Sutarwala. DSP design tool requirements for embedded systems: A telecommunications industrial perspective. *Journal of VLSI Signal Processing*, 1994.

[69] S. Pees. *SuperSimulator – Kompilierte Simulation für Digitale Signalprozessoren*. Integrated Signal Processing Systems (ISS), Aachen University of Technology (RWTH), July 1995. Diploma thesis D 309.

[70] S. Pees and A. Hoffmann. Retargetable simulation and LISA description of the TMS320C6x DSP. Technical Report IB 3/12/1999, Integrated Signal Processing Systems (ISS), Aachen University of Technology (RWTH), Dec. 1999. Final project report.

[71] S. Pees, A. Hoffmann, and H. Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *IEEE Transactions on Design Automation of Electronic Systems*, 5(4), October 2000.

[72] S. Pees, A. Hoffmann, and H. Meyr. Retargeting of Compiled Simulators for Digital Signal Processors Using a Machine Description Language. In *Proceedings of the European Conference on Design, Automation and Test (DATE)*, Paris, March 2000.

[73] S. Pees, A. Hoffmann, and A. Nohl. *LISA User's Manual*. Integrated Signal Processing Systems (ISS), Aachen University of Technology (RWTH), http://www.iss.rwth-aachen.de/lisa, Oct. 2000.

[74] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA – Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *Proceedings of the Design Automation Conference (DAC)*, New Orleans, June 1999.

[75] S. Pees, A. Ropers, and V. Živojnović. Compiled simulation of the TMS320C54x DSP. Technical Report IB 12/5/1997, ISS - Aachen University of Technology, May 1997. Final project report.

[76] S. Pees, V. Živojnović, A. Ropers, and H. Meyr. Fast Simulation of the TI TMS 320C54x DSP. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, pages 995–999, San Diego, Sep. 1997.

[77] The GNU Project. *The Bison Manual*. Free Software Foundation, Boston, MA, November 1999, Edition for Version 1.29. ISBN: 1-882114 44 2.

[78] The GNU Project. *Debugging with DDD*. Free Software Foundation, Boston, MA, January 2000, for DDD Version 3.2. ISBN: 1-882114 44 2.

[79] The GNU Project. *The Flex manual*. Free Software Foundation, Boston, MA, Edition 1.03 for Version 2.3.7. ISBN: 1-882114 21 3.

[80] The GNU Project. *Debugging with GDB*. Free Software Foundation, Boston, MA, Edition for Version 5.0. ISBN: 1-882114 77 9.

[81] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *Int. Conf. on VLSI Design*, Goa, India, Jan. 1999.

[82] A. Ropers and H. Meyr. DSP-Compilers, Desired Link or Developers Nightmare. In *Proc. of DSP Deutschland 99*, München, Sept. 1999.

[83] A. Ropers, S. Pees, and T. Brueggen. Techniques for compiled hardware software cosimulation. In *DSP Germany*, pages 162–171, Munich, Oct. 1998.

[84] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology*, pages 34–43, Winter 1995.

[85] J. Rowson. Hardware/Software co-simulation. In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1994.

[86] C. Schläger. *Modeling of the Pipeline and Peripherals for the Compled Simulation using the Rockwell RISC-Signal-Processor (RSP) as an Example*. Integrated Signal Processing Systems (ISS), Aachen University of Technology (RWTH), July 1996. Diploma thesis D 343.

[87] C. Schneider. Executable specification for multimedia supporting refinement and architecture exploration. In *Proc. of the Euromicro Conference (EUROMICRO)*, Milan, Sep. 1999.

[88] L. Semeria and A. Ghosh. Methodology for Hardware/Software Co-verification in C/C++. In *Proc. of the IEEE Int. High Level Design Validation and Test Workshop (HLDVT)*, pages 67–72, San Diego, Nov. 1999.

[89] C. Siska. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *Proc. of the Int. Symposium on System Synthesis (ISSS)*, Dec. 1998.

[90] R.M. Stallman. *Using and Porting the GNU Compiler Collection (GCC)*. Free Software Foundation, Boston, MA, 2.95 edition, July 1999.

[91] S. Sutarwala, P. Paulin, and Y. Kumar. Insulin: An instruction set simulation environment. In *Proc. of the Conference on Hardware Description Languages*, pages 355–362, Ottawa, Apr. 1993.

[92] Synopsys. *Eagle i*
http://www.synopsys.com/products/hwsw.

[93] Texas Instruments. *TMS320C55x DSP Reference Set: CPU and Peripherals*, Apr. 1999. SPRU131f.

[94] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, Oct. 2000. SPRU189f.

[95] R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, Jan. 1967.

[96] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proc. of the Int. Conf. on Measurement and Modeling of Computer Systems*, May 1996.

[97] V. Zivojnovic. *Der quantitative Ansatz zum gleichzeitigen Entwurf der DSP Architektur und des Compilers*. Shaker Verlag, Aachen, Dissertation an der RWTH-Aachen, 1998. ISBN 3-8265-3919-2.

[98] V. Zivojnovic. *Der quantitative Ansatz zum gleichzeitigen Entwurf der DSP Architektur und des Compilers*. Shaker Verlag, Aachen, Dissertation an der RWTH-Aachen, 1998. ISBN 3-8265-3919-2.

[99] V. Živojnović, J. Martínez, C. Schläger, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proc. of ICSPAT'94 - Dallas*, Oct. 1994.

[100] V. Živojnović, S. Pees, and H. Meyr. LISA – machine description language and generic machine model for HW/SW co-design. In *Proceedings of the IEEE Workshop on VLSI Signal Processing*, San Francisco, Oct. 1996.

[101] V. Živojnović, H. Schraut, M. Willems, and R. Schoenen. DSPs, GPPs, and multimedia applications: An evaluation using DSPstone. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, Boston, Oct. 1995.

[102] V. Živojnović, S. Tjiang, and H. Meyr. Compiled simulation of programmable DSP architectures. In *Proc. of IEEE Workshop on VLSI Signal Processing*, pages 187–196, Sakai, Osaka, Oct. 1995.

Appendix A

Acknowledgement

I would like to thank particularly the whole LISA team at the Integrated Signal Processing Systems Laboratory for the productive collaboration and the joint effort to design the LISA language and to create the experimental tool set based on the LISA language.

Appendix B

LISA Grammar

start ::=
    operation_list

operation_list ::=
    operation_list t_OPERATION ident '{' subop_or_section '}'
  | *operation_list t_OPERATION ident t_IN ident '.' ident '{' subop_or_section '}'*
  | *operation_list t_ALIAS t_OPERATION ident '{' subop_or_section '}'*
  | *operation_list t_ALIAS t_OPERATION ident t_IN ident '.' ident '{' subop_or_section '}'*
  | *operation_list t_INSTRUCTION ident '{' subop_or_section '}'*
  | *operation_list t_INSTRUCTION ident t_IN ident '.' ident '{' subop_or_section '}'*
  | *operation_list t_ALIAS t_INSTRUCTION ident '{' subop_or_section '}'*
  | *operation_list t_ALIAS t_INSTRUCTION ident t_IN ident '.' ident '{' subop_or_section '}'*
  | *operation_list t_RESOURCE '{' resource_list '}'*
  | *operation_list t_ARCHITECTURE ident '{' '}'*
  | *operation_list t_INCLUDE '<' include_file_name '>'*
  | *operation_list t_INCLUDE t_IN_QUOTA*
  | */* empty */*
  | *error*

subop_or_section ::=
    section_list
  | *suboperation_list*

suboperation_list ::=
    suboperation_list t_SUBOPERATION ident '{' section_list '}'
  | *t_SUBOPERATION ident '{' section_list '}'*

section_list ::=
    section_list declare_section
  | *section_list coding_section*
  | *section_list syntax_section*
  | *section_list behavior_section*
  | *section_list expression_section*
  | *section_list activation_section*
  | *section_list if_section_list*
  | *section_list switch_section_list*
  | */*empty*/*

if_section_list ::=
    t_IF '(' if_expression ')' t_THEN '{' section_list '}' else_section_list

else_section_list ::=
    t_ELSE '{' section_list '}'
  | */* empty */*

switch_section_list ::=

t_SWITCH '(' *ident* ')' '{' *case_section_list section_default* '}'

case_section_list ::=
case_section_list t_CASE in_case_selection ':' '{' *section_list* '}'
| *t_CASE in_case_selection* ':' '{' *section_list* '}'

in_case_selection ::=
t_ZEICHEN
| *t_ZAHLEN*

section_default ::=
t_DEFAULT ':' '{' *section_list* '}'
| */* empty */*

contains_operations ::=
ident ';'
| *ident* ',' *contains_operations*

pipeline_register_list ::=
t_PIPE_REG '(' *ident* ',' *ident* '/' *ident* ')' '.' *ident* ',' *pipeline_register_list*
| *t_PIPE_REG* '(' *ident* ',' *ident* '/' *ident* ')' '.' *ident* ';'

resource_list ::=
resource_list resource
| *resource_list t_MEMORY_MAP* '{' *mem_map* '}'
| *resource_list t_PIPELINE pipe_name_list* '=' '{' *pipe_stage_list* '}' ';'
| *resource_list t_PIPE_REG t_IN ident* '{' *resource_list_in_pipe_reg* '}' ';'
| *resource_list t_PROG_MEM resource*
| *resource_list t_DATA_MEM resource*
| *resource_list t_REGISTER resource*
| *resource_list t_CONT_REGISTER resource*
| *resource_list t_PROG_COUNTER resource*
| *resource_list t_PORT resource*
| */* empty */*

resource_list_in_pipe_reg ::=
resource_list_in_pipe_reg resource
| *resource_list_in_pipe_reg t_REGISTER resource*
| *resource_list_in_pipe_reg t_CONT_REGISTER resource*
| */* empty */*

resource ::=
resource_element_list ';'
| *resource_element_with_type t_ALIAS ident* ';'
| *resource_element_with_type t_ALIAS ident array_in_C_style* ';'

resource_element_list ::=
resource_element_list ',' *resource_element*
| *resource_element_with_type*

resource_element_with_type ::=
resource_type ident resource_array_with_upper_and_lower_boundary
| *t_UNSIGNED t_BIT_SIGN array_in_C_style ident resource_array_with_upper_and_lower_boundary*
| *t_SIGNED t_BIT_SIGN array_in_C_style ident resource_array_with_upper_and_lower_boundary*
| *t_BIT_SIGN array_in_C_style ident resource_array_with_upper_and_lower_boundary*

resource_type ::=
resource_type ident
| *ident*

resource_element ::=
ident resource_array_with_upper_and_lower_boundary

pipe_name_list ::=
pipe_name_list ',' *ident*
| *pipe_name_list* ',' *t_IF*
| *ident*

| *t_IF*

pipe_stage_list ::=
pipe_stage_list ';' *ident*
| *pipe_stage_list* ',' *t_IF*
| *ident*
| *t_IF*

mem_map ::=
mem_map in_mem_map t_PTR in_mem_map ',' *t_BYTE* '(' *byte_size* ')' ':' *ident* '[' *array_boundaries*
t_DOUB array_boundaries ']' *struct_or_union* ',' *t_BYTE* '(' *byte_size* ')' ';'
| *mem_map in_mem_map t_PTR in_mem_map* ',' *t_BYTE* '(' *byte_size* ')' ':' ';'
| */* empty */*

in_mem_map ::=
t_HZAHLEN

struct_or_union ::=
struct_or_union '.' *ident*
| *ident*
| */* empty */*

expression_section ::=
t_EXPRESSION '{' *expression_with_type* '}'

expression_with_type ::=
in_expression
| '(' *expression_type* ')' *in_expression*

expression_type ::=
expression_type ident
| *t_UNSIGNED*
| *t_SIGNED*
| *ident*

in_expression ::=
in_expression '[' *in_expression_brackets* ']'
| *in_expression* '(' *in_expression* ')'
| *in_expression t_HZAHLEN*
| *in_expression t_ZAHLEN*
| *in_expression t_ZEICHEN*
| *in_expression* '+'
| *in_expression* '/'
| *in_expression* '-'
| *in_expression t_LEFT_SHIFT*
| *in_expression* '*'
| *in_expression t_RIGHT_SHIFT*
| *in_expression* '.'
| *t_HZAHLEN*
| *t_ZAHLEN*
| *t_ZEICHEN*
| '+'
| '-'
| '*'

in_expression_brackets ::=
in_expression_brackets '[' *in_expression_brackets* ']'
| *in_expression_brackets* '(' *in_expression_brackets* ')'
| *in_expression_brackets all_in_expression*
| */* empty */*

all_in_expression ::=
t_HZAHLEN
| *t_ZAHLEN*
| *t_ZEICHEN*
| '+'
| '/'
| '-'

| *t_LEFT_SHIFT*
| '*'
| *t_RIGHT_SHIFT*
| '.'

declare_section ::=
t_DECLARE '{' *declare_list* '}'

declare_list ::=
declare_list class ';'
| *declare_list group* ';'
| *declare_list t_INSTANCE instance_or_label_identifier_list* ';'
| *declare_list t_LABEL instance_or_label_identifier_list* ';'
| *declare_list t_REFERENCE reference_name_identifier_list* ';'
| *declare_list enum* ';'
| */* empty */*

enum ::=
t_ENUM ident '=' '{' *enum_name_identifier_list* '}'

class ::=
t_CLASS class_name_identifier_list '=' '{' *class_element_list* '}'

group ::=
t_GROUP class_name_identifier_list '=' '{' *class_element_list* '}'

class_element_list ::=
class_element_list t_DOUB_OR ident
| *ident*

coding_section ::=
t_CODING pc_reference '{' *coding_list* '}'

pc_reference ::=
t_AT '(' *in_brackets* ')'
| */* empty */*

coding_list ::=
coding_element
| *compare_list_coding*
| */* empty */*

compare_list_coding ::=
compare_list_par_coding

compare_list_par_coding ::=
compare_list_par_coding operator_AND '(' *comp_eq_coding* ')'
| *compare_list_par_coding operator_AND comp_eq_coding*
| *comp_eq_coding*
| '(' *comp_eq_coding* ')'

comp_eq_coding ::=
ident_chain array_in_C_style t_DOUB_EQ coding_element
| *t_PIPE_REG* '(' *ident* ',' *ident* '/' *ident* ')' '.' *ident t_DOUB_EQ coding_element*
| *ident_chain t_DOUB_EQ coding_element*

coding_element ::=
coding_element bit_field
| *coding_element ident* '=' *coding_syntax_expr*
| *coding_element ident*
| *coding_element ident* '=' *array_with_upper_and_lower_boundary*
| *ident* '=' *coding_syntax_expr*
| *ident* '=' *array_with_upper_and_lower_boundary*
| *bit_field*
| *ident*

coding_syntax_expr ::=

'(' *coding_syntax_expr coding_syntax_operator coding_syntax_expr* ')'
| '(' *sign coding_syntax_expr* ')'
| '(' *coding_syntax_expr* ')'
| '(' *ident* '=' *syntax_type* ')'
| *t_CURRENT_ADDRESS*
| *t_ZAHLEN*
| *bit_field* '!' '(' *bit_field_list* ')'
| *bit_field*
| *ident*

sign ::=
'-'
| '+'

coding_syntax_operator ::=
t_LEFT_SHIFT
| *t_RIGHT_SHIFT*
| '+'
| '-'

syntax_section ::=
t_SYNTAX '{' *syntax_list* '}'

syntax_list ::=
syntax_list syntax_element
| *syntax_list* ' ' *syntax_element*
| */* empty */*

syntax_element ::=
ident
| *ident* '.' *ident*
| *expression_in_quotes*
| *coding_syntax_expr* '=' *syntax_type*
| *t_SYMBOL* '(' *coding_syntax_expr* '=' *syntax_type* ')'
| *t_SYMBOL* '(' *expression_in_quotes coding_syntax_expr* '=' *syntax_type* ')'
| *t_SYMBOL* '(' *coding_syntax_expr* '=' *syntax_type expression_in_quotes* ')'
| *t_SYMBOL* '(' *expression_in_quotes coding_syntax_expr* '=' *syntax_type expression_in_quotes* ')'
| *t_EOI* ':' *t_ZAHLEN*

syntax_type ::=
t_USIGNED
| *t_SIGNED*
| *t_HEX_UN*
| *t_BIN_UN*

expression_in_quotes ::=
t_IN_QUOTA

behavior_section ::=
t_BEHAVIOR pre_section '{' *behavior_rtl_list* '}'
| *t_BEHAVIOR pre_section*

pre_section ::=
t_REQUIRES '(' *require_list* ')' *t_USES* '(' *use_list* ')'
| *t_REQUIRES* '(' *require_list* ')' ',' *t_USES* '(' *use_list* ')'
| *t_USES* '(' *use_list* ')' *t_REQUIRES* '(' *require_list* ')'
| *t_USES* '(' *use_list* ')' ',' *t_REQUIRES* '(' *require_list* ')'
| *t_REQUIRES* '(' *require_list* ')'
| *t_USES* '(' *use_list* ')'
| */* empty */*

require_list ::=
require_list ',' *ident require_array*
| *require_list* ',' '!' *ident require_array*
| *ident require_array*
| '!' *ident require_array*

| /* empty */

use_list ::=
use_list resources_in_USE
| /* empty */

resources_in_USE ::=
t_IN use_identifier_list ';'
| t_OUT use_identifier_list ';'
| t_INOUT use_identifier_list ';'

use_identifier_list ::=
use_identifier_list ',' ident parameters
| use_identifier_list ',' ident
| use_identifier_list ',' ident array_in_C_style
| use_identifier_list ',' ident '[' ']'
| use_identifier_list ',' t_PIPE_REG '(' ident ',' ident '/' ident ')' '.' ident
| use_identifier_list ',' t_PIPE_REG '(' ident ',' ident '/' ident ')' '.' ident array_in_C_style
| use_identifier_list ',' t_PIPE_REG '(' ident ',' ident '/' ident ')' '.' ident '[' ']'
| ident
| ident array_in_C_style
| ident '[' ']'
| ident parameters
| t_PIPE_REG '(' ident ',' ident '/' ident ')' '.' ident
| t_PIPE_REG '(' ident ',' ident '/' ident ')' '.' ident array_in_C_style
| t_PIPE_REG '(' ident ',' ident '/' ident ')' '.' ident '[' ']'

behavior_rtl_list ::=
c_statement

c_statement ::=
c_statement '{' c_statement '}'
| c_statement all_ccode
| /* empty */

activation_section ::=
t_ACTIVATION '{' activation_section_list '}'

activation_section_list ::=
activation_section_list activation_if activation_element_list
| activation_element_list

activation_element_list ::=
activation_element ';' activation_element_list
| activation_element ',' activation_element_list
| activation_element

activation_element ::=
ident parameters
| ident
| pipe_function
| /* empty */

activation_if ::=
t_if '(' in_brackets ')' '{' activation_section_list '}' activation_else

in_brackets ::=
in_brackets '(' in_brackets ')'
| in_brackets all_ccode
| /* empty */

activation_else ::=
t_else '{' activation_section_list '}'
| /* empty */

require_array ::=
require_array '[' in_require_array ']'

| */* empty */*

in_require_array ::=
t_ZEICHEN
| *t_ZAHLEN*

array_in_C_style ::=
array_in_C_style '[' *array_boundaries* ']'
| '[' *array_boundaries* ']'

resource_array_with_upper_and_lower_boundary ::=
array_with_upper_and_lower_boundary in_round_brackets
| *array_with_upper_and_lower_boundary*
| *in_round_brackets*
| */* empty */*

array_with_upper_and_lower_boundary ::=
array_with_upper_and_lower_boundary '[' *array_boundaries* t_DOUB array_boundaries ']'
| '[' *array_boundaries t_DOUB array_boundaries* ']'

in_round_brackets ::=
'(' *array_with_upper_and_lower_boundary* ')'
| '(' *array_with_upper_and_lower_boundary* ')' *array_boundaries*

bit_field_list ::=
bit_field_list t_DOUB_OR bit_field
| *t_BIT*

bit_field ::=
t_BIT

class_name_identifier_list ::=
class_name_identifier_list ',' *ident*
| *ident*

instance_or_label_identifier_list ::=
instance_or_label_identifier_list ',' *ident*
| *ident*

reference_name_identifier_list ::=
reference_name_identifier_list ',' *ident*
| *ident*

enum_name_identifier_list ::=
enum_name_identifier_list ',' *ident*
| *ident*

ident ::=
t_ZEICHEN

parameters ::=
'.' *ident*

operator ::=
t_DOUB_OR
| *t_DOUB_AND*

operator_AND ::=
t_DOUB_AND

if_expression ::=
if_expression_par
| *if_eq*

if_expression_par ::=
if_expression_par operator '(' *if_expression* ')'

| '(' *if_expression* ')'

if_eq ::=
ident_chain t_DOUB_EQ ident_chain
| *ident_chain t_NOT_EQ ident_chain*

ident_chain ::=
ident
| *ident_chain* '.' *ident*
| *t_ZAHLEN*

include_file_name ::=
include_file_name inside_include
| *inside_include*

inside_include ::=
'.'
| '-'
| *t_BIT*
| *t_ZAHLEN*
| *t_ZEICHEN*

all_ccode ::=
t_BEH_DEF

byte_size ::=
t_ZAHLEN

array_boundaries ::=
array_boundaries t_HZAHLEN
| *array_boundaries t_ZAHLEN*
| *array_boundaries t_ZEICHEN*
| *array_boundaries t_PTR*
| *array_boundaries* '+'
| *array_boundaries* '/'
| *array_boundaries* '-'
| *array_boundaries t_LEFT_SHIFT*
| *array_boundaries* '*'
| *array_boundaries t_RIGHT_SHIFT*
| *array_boundaries* '.'
| *t_HZAHLEN*
| *t_ZAHLEN*
| *t_ZEICHEN*

pipe_function ::=
t_PIPE_REG '(' *ident* ',' *ident* '/' *ident* ')' '.' *ident* '(' ')'

Appendix C

Simulator API

The application programming interface (API) of the LISA simulator consists of the following functions:

void reset (char *fileName)
: This function implements the behavior of the operation *reset* as specified in the processor description. Furthermore, the program and data memories are initialized with the target object code. The file name of the object code (*fileName*) is passed as parameter to this API function.

int run_until (int controlSteps)
: This function lets the simulator run for the specified number of control steps. In case of hitting a breakpoint, the execution stops and a one is return. If the count of control steps is run without reaching a breakpoint, a zero is returned.

void move_on_one_step (void)
: This function advances the simulation by executing one control-step.

void move_on_one_insn (void)
: This function advances the simulation until one instruction finishes execution.

void set_breakpoint (unsigned int address)
: This function allows to set a breakpoint at the address that is passed as a parameter.

void clear_breakpoint (unsigned int address)
: This function removes breakpoints at the specified address.

void set_register (char *name, int address)
: This function removes breakpoints at the specified address.

Appendix D

Simulator Hooks

The following hooks are automatically provided by the LISA simulator:

GLOBAL_HOOK can be used to insert global variables into the simulator or to open files for output.

PRE_SIMULATION_HOOK has only effect if the stand alone simulator is run. The hook is inserted at the beginning of the *main*-function.

POST_SIMULATION_HOOK has only effect if the stand alone simulator is run. The hook is inserted at the end of the *main*-function.

RESET_HOOK is inserted at the very beginning of the behavior of the operation *reset*, right before the function loading the memory of the LISA-model with the content of the COFF-file.

PRE_MAIN_HOOK is inserted at the very beginning of operation *main* and is thus passed at every control-step.

POST_MAIN_HOOK is inserted at the very end of operation *main* and is thus passed at every control-step.

PRE_OPERATION_HOOK is placed in the beginning of the behavioral code of each operation. This hook allows to provide operation-specific code. The name of these hooks is composed of the prefix *PRE_OPERATION_HOOK___* followed by the respective operation name. The hook for the operation *example* would cause the lisa compiler to produce a hook named *PRE_OPERATION_HOOK___example*.

POST_OPERATION_HOOK is placed in the end of the operations behavioral code. This hook allows to provide operation-specific code. The name of these hooks is composed of the prefix *POST_OPERATION_HOOK___* followed by the respective operation name. The hook for the operation *example* would cause the lisa compiler to produce a hook named *POST_OPERATION_HOOK___example*.

Appendix E

LISA description of the DLX processor

main.lisa

```
#include "dlx.h"
ARCHITECTURE dlx {}
RESOURCE
{
  MEMORY_MAP
  {
      // Memory is accessed bytewise
      //Syntax:
      // <addr_start> -> <addr_end> : <res_name>, <res_type>, <shift>
      // Program Memory
      0x0000000 -> 0x00FFFFF, BYTES(1) : pmem[0x0..0xFFFFF], BYTES(1);
      // Data Memory
      0x0100000 -> 0x01FFFFF, BYTES(1) : dmem[0x0..0xFFFFF], BYTES(1);
  }
  PROGRAM_COUNTER int pc;
  int cycle;
  int instruction_counter;
  REGISTER int R[0..31];
  REGISTER float F[0..31];
  REGISTER double D[0..15] ALIAS F[0];
  // register(s) used for data memory read operations:
  int lmd, lmd2;
  PROGRAM_MEMORY int pmem[0..0xFFFFF];
  DATA_MEMORY int dmem[0..0xFFFFF];
  PIPELINE pipe = { FE; DC; EX; MEM; WB };
  PIPELINE_REGISTER IN pipe
  {
    int ir;
    REGISTER int npc;
    REGISTER int reg_a, reg_b, imm, alu;
    float reg_a_f, reg_b_f, alu_f;
    double reg_a_d, reg_b_d, alu_d;
    REGISTER bool cond;
  };
  int trap_flag;
  int float_flag;
  int halted_flag;
}
OPERATION main
{
  DECLARE { INSTANCE fetch, decode; }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,EX/MEM).cond; OUT PIPELINE_REGISTER(pipe,FE/DC).npc; )
  {
    PIPELINE(pipe).execute();
    PIPELINE(pipe).shift();
    if (cycle > 0x80000)
      exit(0);
    cycle++;
  }
  ACTIVATION { fetch, decode }
}
OPERATION reset
{
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,EX/MEM).cond; OUT PIPELINE_REGISTER(pipe,FE/DC).npc;)
  {
      pc = 0;
      cycle = 0;
      // R0 is always zero!
      R[0] = 0;
      trap_flag = 0;
      float_flag = 0;
      halted_flag = 0;
      PIPELINE_REGISTER(pipe,EX/MEM).cond = 0;
      PIPELINE_REGISTER(pipe,FE/DC).npc = 0;
      // pipeline init
      PIPELINE(pipe).flush();
      PIPELINE(pipe).execute();
      PIPELINE(pipe).shift();
      PIPELINE(pipe).execute();
      PIPELINE(pipe).shift();
    }
}
```

fetch.lisa

```
OPERATION fetch IN pipe.FE
{
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,EX/MEM).cond; IN PIPELINE_REGISTER(pipe,EX/MEM).alu;
                 IN PIPELINE_REGISTER(pipe,DC/EX).npc; OUT PIPELINE_REGISTER(pipe,FE/DC).npc;)
  {
      ir = pmem[PIPELINE_REGISTER(pipe,FE/DC).npc];
      if (PIPELINE_REGISTER(pipe,EX/MEM).cond)
        PIPELINE_REGISTER(pipe,FE/DC).npc = PIPELINE_REGISTER(pipe,EX/MEM).alu;
      else
        PIPELINE_REGISTER(pipe,FE/DC).npc = PIPELINE_REGISTER(pipe,DC/EX).npc+4;
  }
}
```

decode.lisa

```
OPERATION decode IN pipe.DC
{
  DECLARE { GROUP instruction = {  i_type  ||  r_type  ||  j_type  }; }
  CODING AT(pc) { ir == instruction }
  SYNTAX { instruction  EOI:1 }
  BEHAVIOR { instruction(); }
  ACTIVATION { execute }
}

OPERATION execute IN pipe.EX
{
  BEHAVIOR USES ( IN PIPELINE_REGISTER(pipe,DC/EX).npc; ) { pc = PIPELINE_REGISTER(pipe,DC/EX).npc; }
}

OPERATION i_type
{
  DECLARE { GROUP i_group = { i_arithmetic || i_compare || i_branch || i_load_store ||
                              i_load_store_float || i_load_store_double }; }
  CODING { i_group }
  SYNTAX { i_group }
  BEHAVIOR { i_group(); }
}

OPERATION i_arithmetic IN pipe.DC
{
  DECLARE {
    GROUP opcode = { ADDI || ADDUI || SUBI || SUBUI || ANDI || ORI || XORI || SLLI || SRLI || SRAI || LHI };
    GROUP rs1, rd = { fix_register };
    //INSTANCE i_forwarding, immediate, writeback_register;
    INSTANCE immediate, writeback_register;
  }
  CODING { opcode  rs1  rd  immediate }
  SYNTAX { opcode  rd  ","  rs1  ","  immediate }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a; OUT PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,DC/EX).cond;)
  {
      PIPELINE_REGISTER(pipe,DC/EX).reg_a = rs1;
      PIPELINE_REGISTER(pipe,DC/EX).imm = immediate;
      PIPELINE_REGISTER(pipe,DC/EX).cond = 0;
  }
  ACTIVATION { opcode, writeback_register }
}

OPERATION i_load_store IN pipe.DC
{
  DECLARE {
    GROUP opcode = { LB || LBU || LH || LHU || LW || SB || SH || SW };
    GROUP rs1, rd = { fix_register };
    INSTANCE immediate, gen_address;
  }
  CODING { opcode  rs1  rd  immediate }
  SYNTAX { opcode  rd  ","  rs1  ","  immediate }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a; OUT PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,DC/EX).cond;)
  {
    PIPELINE_REGISTER(pipe,DC/EX).reg_a = rs1;
    PIPELINE_REGISTER(pipe,DC/EX).imm = immediate;
    PIPELINE_REGISTER(pipe,DC/EX).cond = 0;
  }
  ACTIVATION { gen_address, opcode }
}

OPERATION i_load_store_float IN pipe.DC
{
  DECLARE {
    GROUP opcode = { LF || SF };
    GROUP rs1 = { fix_register };
    GROUP rd = { float_register };
    INSTANCE immediate, gen_address;
  }
  CODING { opcode  rs1  rd  immediate }
  SYNTAX { opcode  rd  ","  rs1  ","  immediate }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a; OUT PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,DC/EX).cond; )
  {
    PIPELINE_REGISTER(pipe,DC/EX).reg_a = rs1;
    PIPELINE_REGISTER(pipe,DC/EX).imm = immediate;
    PIPELINE_REGISTER(pipe,DC/EX).cond = 0;
  }
  ACTIVATION { gen_address, opcode }
}

OPERATION i_load_store_double IN pipe.DC
{
  DECLARE {
    GROUP opcode = { LD || SD };
    GROUP rs1 = { fix_register };
    GROUP rd = { double_register };
    INSTANCE immediate, gen_address;
  }
  CODING { opcode  rs1  rd  immediate }
  SYNTAX { opcode  rd  ","  rs1  ","  immediate }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a; OUT PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,DC/EX).cond;)
  {
    PIPELINE_REGISTER(pipe,DC/EX).reg_a = rs1;
    PIPELINE_REGISTER(pipe,DC/EX).imm = immediate;
    PIPELINE_REGISTER(pipe,DC/EX).cond = 0;
  }
```

```
  ACTIVATION { gen_address, opcode }
}

OPERATION i_compare IN pipe.DC
{
  DECLARE {
    GROUP opcode = { SLTI || SGTI || SLEI || SGEI || SEQI || SNEI };
    GROUP rs1, rd = { fix_register };
    //INSTANCE immediate, i_forwarding, writeback_register;
    INSTANCE immediate, writeback_register;
  }
  CODING { opcode  rs1  rd  immediate }
  SYNTAX { opcode  rd  ","  rs1  ","  immediate }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a; OUT PIPELINE_REGISTER(pipe,DC/EX).imm;)
  {
    PIPELINE_REGISTER(pipe,DC/EX).reg_a = rs1;
    PIPELINE_REGISTER(pipe,DC/EX).imm = immediate;
  }
  ACTIVATION { opcode, writeback_register }
}

OPERATION i_branch IN pipe.DC
{
  DECLARE {
    GROUP instruction = { BEQZ || BNEZ || BFPT || BFPF || JR || JALR };
    GROUP rs1 = { fix_register };
    INSTANCE immediate;
  }
  CODING { instruction  rs1  0bx[5]  immediate }
  SYNTAX { instruction  rs1  ","  immediate }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a; OUT PIPELINE_REGISTER(pipe,DC/EX).imm;)
  {
    PIPELINE_REGISTER(pipe,DC/EX).reg_a = rs1;
    PIPELINE_REGISTER(pipe,DC/EX).imm = immediate;
  }
  ACTIVATION { instruction }
}

OPERATION r_type
{
  DECLARE { GROUP type = { r_fixed || r_float || r_double || r_move }; }
  CODING { type }
  SYNTAX { type }
  BEHAVIOR { type(); }
}

OPERATION r_fixed IN pipe.DC
{
  DECLARE {
    GROUP func = { ADD || ADDU || SUB || SUBU ||
                   MULT || MULTU || DIV || DIVU ||
                   AND || OR || XOR ||
                   SLL || SRL || SRA ||
                   SLT || SGT || SLE || SGE || SEQ || SNE };
    GROUP rs1, rs2, rd = { fix_register };
    INSTANCE writeback_register;
  }
  CODING { 0b000000  rs1  rs2  rd  func }
  SYNTAX { func  rd  ","  rs1  ","  rs2 }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a; OUT PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 OUT PIPELINE_REGISTER(pipe,DC/EX).cond;)
  {
    PIPELINE_REGISTER(pipe,DC/EX).reg_a = rs1;
    PIPELINE_REGISTER(pipe,DC/EX).reg_b = rs2;
    PIPELINE_REGISTER(pipe,DC/EX).cond = 0;
  }
  ACTIVATION { func, writeback_register }
}

OPERATION r_float IN pipe.DC
{
  DECLARE {
    GROUP func = { ADDF || SUBF || MULTF || DIVF };
    GROUP rs1, rs2, rd = { float_register };
    INSTANCE writeback_float_register;
  }
  CODING { 0b000001  rs1  rs2  rd  func }
  SYNTAX { func  rd  ","  rs1  ","  rs2 }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a_f; OUT PIPELINE_REGISTER(pipe,DC/EX).reg_b_f;
                 OUT PIPELINE_REGISTER(pipe,DC/EX).cond;)
  {
    PIPELINE_REGISTER(pipe,DC/EX).reg_a_f = rs1;
    PIPELINE_REGISTER(pipe,DC/EX).reg_b_f = rs2;
    PIPELINE_REGISTER(pipe,DC/EX).cond = 0;
  }
  ACTIVATION { func, writeback_float_register }
}

OPERATION r_double IN pipe.DC
{
  DECLARE {
    GROUP func = { ADDD || SUBD || MULTD || DIVD };
    GROUP rs1, rs2, rd = { double_register };
    INSTANCE writeback_double_register;
  }
  CODING { 0b000001  rs1  rs2  rd  func }
  SYNTAX { func  rd  ","  rs1  ","  rs2 }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a_d; OUT PIPELINE_REGISTER(pipe,DC/EX).reg_b_d;
                 OUT PIPELINE_REGISTER(pipe,DC/EX).cond;)
  {
    PIPELINE_REGISTER(pipe,DC/EX).reg_a_d = rs1;
    PIPELINE_REGISTER(pipe,DC/EX).reg_b_d = rs2;
    PIPELINE_REGISTER(pipe,DC/EX).cond = 0;
  }
  ACTIVATION { func, writeback_double_register }
}

OPERATION r_move IN pipe.DC
{
  DECLARE { GROUP instruction = { MOVS2I || MOVI2S || MOVI2FP || MOVFP2I }; }
  CODING { 0b000000  instruction }
  SYNTAX { instruction }
  ACTIVATION { instruction }
}

OPERATION MOVFP2I IN pipe.DC
{
  DECLARE {
    GROUP sr1 = { float_register };
    GROUP rd =  { fix_register };
    INSTANCE writeback_register;
  }
  CODING { sr1  0bx[5]  rd  0b00000110100 }
  SYNTAX { "MOVFP2I"  rd  ","  sr1 }
```

```
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a;) { PIPELINE_REGISTER(pipe,DC/EX).reg_a = (int) sr1; }
  ACTIVATION { move, writeback_register }
}

OPERATION MOVI2FP IN pipe.DC
{
  DECLARE {
    GROUP sr1 = { fix_register };
    GROUP rd =  { float_register };
    INSTANCE writeback_float_register;
  }
  CODING { sr1  0bx[5]  rd  0b00000110101 }
  SYNTAX { "MOVI2FP"  rd  " "  sr1 }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a_f;) { PIPELINE_REGISTER(pipe,DC/EX).reg_a_f = (float) sr1; }
  ACTIVATION { move_float, writeback_float_register }
}

OPERATION MOVI2S IN pipe.DC
{
  DECLARE {
    GROUP sr1 = { fix_register };
    GROUP rd =  { special_register };
    INSTANCE writeback_special_register;
  }
  CODING { sr1  0bx[5]  rd  0b00000110000 }
  SYNTAX { "MOVI2S"  rd  " "  sr1 }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a;) { PIPELINE_REGISTER(pipe,DC/EX).reg_a = sr1; }
  ACTIVATION { move, writeback_special_register }
}

OPERATION MOVS2I IN pipe.DC
{
  DECLARE {
    GROUP sr1 = { special_register };
    GROUP rd =  { fix_register };
    INSTANCE writeback_register;
  }
  CODING { sr1  0bx[5]  rd  0b00000110001 }
  SYNTAX { "MOVS2I"  rd  " "  sr1 }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,DC/EX).reg_a;) { PIPELINE_REGISTER(pipe,DC/EX).reg_a = sr1; }
  ACTIVATION { move, writeback_register }
}

OPERATION j_type IN pipe.DC
{
  DECLARE {
    GROUP instruction = { J || JAL || TRAP || RFE };
    GROUP addr = { offset };
  }
  CODING { instruction  addr }
  SYNTAX { instruction  addr }
  BEHAVIOR { instruction(); }
}
```

operands.lisa

```
OPERATION fix_register
{
  DECLARE { LABEL index; }
  CODING { index=0bx[5] }
  SYNTAX { "R["  ~index=#U  ~"]" }
  EXPRESSION { R[index] }
}

OPERATION float_register
{
  DECLARE { LABEL index; }
  CODING { index=0bx[5] }
  SYNTAX { "F["  ~index=#U  ~"]" }
  EXPRESSION { F[index] }
}

OPERATION double_register
{
  DECLARE { LABEL index; }
  CODING { index=(0bx[5]>>1) }
  SYNTAX { "F["  ~(index<<1)=#U  ~"]" }
  EXPRESSION { D[index] }
}

OPERATION special_register
{
  DECLARE { LABEL index; }
  CODING { index=0bx[5] }
  SYNTAX { "R["  ~index=#U  ~"]" }
  EXPRESSION { R[index] }
}

OPERATION immediate
{
  DECLARE { LABEL index; }
  CODING { index=0bx[16] }
  SYNTAX { SYMBOL (index=#U) }
  EXPRESSION { index }
}

OPERATION offset
{
  DECLARE { LABEL index; }
  CODING { index=0bx[26] }
  SYNTAX { SYMBOL (index=#U) }
  EXPRESSION { index }
}
```

r_type.lisa

```
OPERATION ADD  IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b00000100000 }
  SYNTAX { "ADD" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a + PIPELINE_REGISTER(pipe,DC/EX).reg_b;
    }
}
```

```
OPERATION ADDU IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000100001 }
  SYNTAX { "ADDU" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

30   {
       PIPELINE_REGISTER(pipe,EX/MEM).alu = (unsigned) (PIPELINE_REGISTER(pipe,DC/EX).reg_a + PIPELINE_REGISTER(pipe,DC/EX).reg_b);
     }
}

OPERATION SUB IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000100010 }
  SYNTAX { "SUB" }
40 BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

     {
       PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a - PIPELINE_REGISTER(pipe,DC/EX).reg_b;
     }
}

OPERATION SUBU IN pipe.EX
{
50 DECLARE { REFERENCE rd; }
  CODING { 0b000000100011 }
  SYNTAX { "SUBU" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

     {
       PIPELINE_REGISTER(pipe,EX/MEM).alu = (unsigned) (PIPELINE_REGISTER(pipe,DC/EX).reg_a - PIPELINE_REGISTER(pipe,DC/EX).reg_b);
     }
}
60
OPERATION MULT  IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000001110 }
  SYNTAX { "MULT" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

     {
70     PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a * PIPELINE_REGISTER(pipe,DC/EX).reg_b;
     }
}

OPERATION MULTU IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000010110 }
  SYNTAX { "MULTU" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
80               IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

     {
       PIPELINE_REGISTER(pipe,EX/MEM).alu = ((unsigned) ((unsigned) PIPELINE_REGISTER(pipe,DC/EX).reg_a
                                          * (unsigned) PIPELINE_REGISTER(pipe,DC/EX).reg_b));
     }
}

OPERATION DIV IN pipe.EX
{
90 DECLARE { REFERENCE rd; }
  CODING { 0b000000001111 }
  SYNTAX { "DIV" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

     {
       if ( PIPELINE_REGISTER(pipe,DC/EX).reg_b != 0 )
         PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a / PIPELINE_REGISTER(pipe,DC/EX).reg_b;
     }
100 }

OPERATION DIVU IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000010111 }
  SYNTAX { "DIVU" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

110  {
       if ( PIPELINE_REGISTER(pipe,DC/EX).reg_b != 0 )
         PIPELINE_REGISTER(pipe,EX/MEM).alu = ((unsigned) ((unsigned) PIPELINE_REGISTER(pipe,DC/EX).reg_a
                                            / (unsigned) PIPELINE_REGISTER(pipe,DC/EX).reg_b));
     }
}

OPERATION AND IN pipe.EX
{
  DECLARE { REFERENCE rd; }
120 CODING { 0b000000100100 }
  SYNTAX { "AND" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

     {
       PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a & PIPELINE_REGISTER(pipe,DC/EX).reg_b;
     }
}

130 OPERATION OR  IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000100101 }
  SYNTAX { "OR" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

     {
       PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a | PIPELINE_REGISTER(pipe,DC/EX).reg_b;
140  }
}

OPERATION XOR IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000100110 }
  SYNTAX { "XOR" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)
150
     {
       PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a ^ PIPELINE_REGISTER(pipe,DC/EX).reg_b;
     }
```

```
}
OPERATION SLL IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000000100 }
  SYNTAX { "SLL" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a << PIPELINE_REGISTER(pipe,DC/EX).reg_b;
    }
}
OPERATION SRL IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000000110 }
  SYNTAX { "SRL" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a >> PIPELINE_REGISTER(pipe,DC/EX).reg_b)
                                         & (0xFFFFFFFFU >> PIPELINE_REGISTER(pipe,DC/EX).reg_b);
    }
}
OPERATION SRA IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000000111 }
  SYNTAX { "SRA" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a >> PIPELINE_REGISTER(pipe,DC/EX).reg_b;
    }
}
OPERATION SLT IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000101010 }
  SYNTAX { "SLT" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a < PIPELINE_REGISTER(pipe,DC/EX).reg_b) ? 1 : 0;
    }
}
OPERATION SGT IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000101011 }
  SYNTAX { "SGT" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a > PIPELINE_REGISTER(pipe,DC/EX).reg_b) ? 1 : 0;
    }
}
OPERATION SLE IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000101100 }
  SYNTAX { "SLE" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a <= PIPELINE_REGISTER(pipe,DC/EX).reg_b) ? 1 : 0;
    }
}
OPERATION SGE IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000101101 }
  SYNTAX { "SGE" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a >= PIPELINE_REGISTER(pipe,DC/EX).reg_b) ? 1 : 0;
    }
}
OPERATION SEQ IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000101000 }
  SYNTAX { "SEQ" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a == PIPELINE_REGISTER(pipe,DC/EX).reg_b) ? 1 : 0;
    }
}
OPERATION SNE IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000101001 }
  SYNTAX { "SNE" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu;)

    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a
                                          != PIPELINE_REGISTER(pipe,DC/EX).reg_b) ? 1 : 0;
    }
}
OPERATION move IN pipe.EX
{
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a;
    }
}
OPERATION move_float IN pipe.EX
{
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a_f; OUT PIPELINE_REGISTER(pipe,EX/MEM).alu_f;)
```

```
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu_f = PIPELINE_REGISTER(pipe,DC/EX).reg_a_f;
    }
}

OPERATION ADDF IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000000000 }
  SYNTAX { "ADDF" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a_f; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b_f;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu_f;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu_f = PIPELINE_REGISTER(pipe,DC/EX).reg_a_f + PIPELINE_REGISTER(pipe,DC/EX).reg_b_f;
    }
}

OPERATION ADDD IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000000100 }
  SYNTAX { "ADDD" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a_d; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b_d;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu_d;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu_d = PIPELINE_REGISTER(pipe,DC/EX).reg_a_d + PIPELINE_REGISTER(pipe,DC/EX).reg_b_d;
    }
}

OPERATION SUBF IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000000001 }
  SYNTAX { "SUBF" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a_f; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b_f;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu_f;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu_f = PIPELINE_REGISTER(pipe,DC/EX).reg_a_f - PIPELINE_REGISTER(pipe,DC/EX).reg_b_f;
    }
}

OPERATION SUBD IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000000101 }
  SYNTAX { "SUBD" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a_d; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b_d;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu_d;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu_d = PIPELINE_REGISTER(pipe,DC/EX).reg_a_d - PIPELINE_REGISTER(pipe,DC/EX).reg_b_d;
    }
}

OPERATION MULTF IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000000010 }
  SYNTAX { "MULTF" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a_f; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b_f;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu_f;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu_f = PIPELINE_REGISTER(pipe,DC/EX).reg_a_f * PIPELINE_REGISTER(pipe,DC/EX).reg_b_f;
    }
}

OPERATION MULTD IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000000110 }
  SYNTAX { "MULTD" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a_d; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b_d;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu_d;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu_d = PIPELINE_REGISTER(pipe,DC/EX).reg_a_d * PIPELINE_REGISTER(pipe,DC/EX).reg_b_d;
    }
}

OPERATION DIVF IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000000011 }
  SYNTAX { "DIVF" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a_f; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b_f;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu_f;)
    {
      if (PIPELINE_REGISTER(pipe,DC/EX).reg_b_f != 0)
        PIPELINE_REGISTER(pipe,EX/MEM).alu_f = PIPELINE_REGISTER(pipe,DC/EX).reg_a_f / PIPELINE_REGISTER(pipe,DC/EX).reg_b_f;
    }
}

OPERATION DIVD IN pipe.EX
{
  DECLARE { REFERENCE rd; }
  CODING { 0b000000000111 }
  SYNTAX { "DIVD" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a_d; IN PIPELINE_REGISTER(pipe,DC/EX).reg_b_d;
                 IN PIPELINE_REGISTER(pipe,EX/MEM).alu_d;)
    {
      if (PIPELINE_REGISTER(pipe,DC/EX).reg_b_d != 0)
        PIPELINE_REGISTER(pipe,EX/MEM).alu_d = PIPELINE_REGISTER(pipe,DC/EX).reg_a_d / PIPELINE_REGISTER(pipe,DC/EX).reg_b_d;
    }
}
```

i_type.lisa

```
OPERATION ADDI  IN pipe.EX
{
  CODING { 0b001000 }
  SYNTAX { "ADDI" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a + PIPELINE_REGISTER(pipe,DC/EX).imm;
    }
}

OPERATION ADDUI IN pipe.EX
{
  CODING { 0b001001 }
  SYNTAX { "ADDUI" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = (unsigned) (PIPELINE_REGISTER(pipe,DC/EX).reg_a + PIPELINE_REGISTER(pipe,DC/EX).imm);
```

```
    }
}
OPERATION SUBI  IN pipe.EX
{
  CODING { 0b001010 }
  SYNTAX { "SUBI" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a - PIPELINE_REGISTER(pipe,DC/EX).imm;
    }
}
OPERATION SUBUI IN pipe.EX
{
  CODING { 0b001011 }
  SYNTAX { "SUBUI" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = (unsigned) (PIPELINE_REGISTER(pipe,DC/EX).reg_a - PIPELINE_REGISTER(pipe,DC/EX).imm);
    }
}
OPERATION ANDI IN pipe.EX
{
  CODING { 0b001100 }
  SYNTAX { "ANDI" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a & PIPELINE_REGISTER(pipe,DC/EX).imm;
    }
}
OPERATION ORI  IN pipe.EX
{
  CODING { 0b001101 }
  SYNTAX { "ORI" }
  BEHAVIOR USES(IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a | PIPELINE_REGISTER(pipe,DC/EX).imm;
    }
}
OPERATION XORI IN pipe.EX
{
  CODING { 0b001110 }
  SYNTAX { "XORI" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a ^ PIPELINE_REGISTER(pipe,DC/EX).imm;
    }
}
OPERATION SRAI IN pipe.EX
{
  CODING { 0b010111 }
  SYNTAX { "SRAI" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a >> PIPELINE_REGISTER(pipe,DC/EX).imm;
    }
}
OPERATION SLLI IN pipe.EX
{
  CODING { 0b010100 }
  SYNTAX { "SLLI" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a << PIPELINE_REGISTER(pipe,DC/EX).imm;
    }
}
OPERATION SRLI IN pipe.EX
{
  CODING { 0b010110 }
  SYNTAX { "SRLI" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a >> PIPELINE_REGISTER(pipe,DC/EX).imm)
                                         & (0xFFFFFFFFU >> PIPELINE_REGISTER(pipe,DC/EX).imm);
    }
}
OPERATION gen_address IN pipe.EX
{
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).reg_a + PIPELINE_REGISTER(pipe,DC/EX).imm;
    }
}
OPERATION LHI IN pipe.EX
{
  CODING { 0b001111 }
  SYNTAX { "LHI" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).imm << 16);
    }
}
OPERATION BEQZ IN pipe.EX
{
  CODING { 0b000100 }
  SYNTAX { "BEQZ" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm; IN PIPELINE_REGISTER(pipe,DC/EX).npc;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu; OUT PIPELINE_REGISTER(pipe,EX/MEM).cond;)
    {
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).npc + PIPELINE_REGISTER(pipe,DC/EX).imm;
      PIPELINE_REGISTER(pipe,EX/MEM).cond = (PIPELINE_REGISTER(pipe,DC/EX).reg_a == 0);
    }
}
OPERATION BNEZ IN pipe.EX
{
  CODING { 0b000101 }
  SYNTAX { "BNEZ" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm; IN PIPELINE_REGISTER(pipe,DC/EX).npc;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu; OUT PIPELINE_REGISTER(pipe,EX/MEM).cond;)
    {
```

```
        PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).npc + PIPELINE_REGISTER(pipe,DC/EX).imm;
        PIPELINE_REGISTER(pipe,EX/MEM).cond = (PIPELINE_REGISTER(pipe,DC/EX).reg_a != 0);
    }
}

OPERATION BFPT IN pipe.EX
{
    CODING { 0b000111 }
    SYNTAX { "BFPT" }
    BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).imm; IN PIPELINE_REGISTER(pipe,DC/EX).npc;
                   OUT PIPELINE_REGISTER(pipe,EX/MEM).alu; OUT PIPELINE_REGISTER(pipe,EX/MEM).cond;)
        {
        PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).npc + PIPELINE_REGISTER(pipe,DC/EX).imm;
        PIPELINE_REGISTER(pipe,EX/MEM).cond = float_flag;
    }
}

OPERATION BFPF IN pipe.EX
{
    CODING { 0b000110 }
    SYNTAX { "BFPF" }
    BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).npc; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                   OUT PIPELINE_REGISTER(pipe,EX/MEM).alu; OUT PIPELINE_REGISTER(pipe,EX/MEM).cond;)
        {
        PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).npc + PIPELINE_REGISTER(pipe,DC/EX).imm;
        PIPELINE_REGISTER(pipe,EX/MEM).cond = !float_flag;
    }
}

OPERATION JR IN pipe.EX
{
    CODING { 0b010010 }
    SYNTAX { "JR" }
    BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).npc; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                   OUT PIPELINE_REGISTER(pipe,EX/MEM).alu; OUT PIPELINE_REGISTER(pipe,EX/MEM).cond;)
        {
        PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).npc + PIPELINE_REGISTER(pipe,DC/EX).imm;
        PIPELINE_REGISTER(pipe,EX/MEM).cond = true;
    }
}

OPERATION JALR IN pipe.EX
{
    CODING { 0b010011 }
    SYNTAX { "JALR" }
    BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).npc; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                   OUT PIPELINE_REGISTER(pipe,EX/MEM).alu; OUT PIPELINE_REGISTER(pipe,EX/MEM).cond;)
        {
        PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).npc + PIPELINE_REGISTER(pipe,DC/EX).imm;
        R[31] = PIPELINE_REGISTER(pipe,DC/EX).npc;
        PIPELINE_REGISTER(pipe,EX/MEM).cond = true;
    }
}

OPERATION SLTI IN pipe.EX
{
    DECLARE { REFERENCE rd; }
    CODING { 0b011010 }
    SYNTAX { "SLTI" }
    BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                   OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
        {
        PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a < PIPELINE_REGISTER(pipe,DC/EX).imm) ? 1 : 0;
        }
}

OPERATION SGTI IN pipe.EX
{
    DECLARE { REFERENCE rd; }
    CODING { 0b011011 }
    SYNTAX { "SGTI" }
    BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                   OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
        {
        PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a > PIPELINE_REGISTER(pipe,DC/EX).imm) ? 1 : 0;
        }
}

OPERATION SLEI IN pipe.EX
{
    DECLARE { REFERENCE rd; }
    CODING { 0b011100 }
    SYNTAX { "SLEI" }
    BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                   OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
        {
        PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a <= PIPELINE_REGISTER(pipe,DC/EX).imm) ? 1 : 0;
        }
}

OPERATION SGEI IN pipe.EX
{
    DECLARE { REFERENCE rd; }
    CODING { 0b011101 }
    SYNTAX { "SGEI" }
    BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                   OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
        {
        PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a >= PIPELINE_REGISTER(pipe,DC/EX).imm) ? 1 : 0;
        }
}

OPERATION SEQI IN pipe.EX
{
    DECLARE { REFERENCE rd; }
    CODING { 0b011000 }
    SYNTAX { "SEQI" }
    BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                   OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
        {
        PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a == PIPELINE_REGISTER(pipe,DC/EX).imm) ? 1 : 0;
        }
}

OPERATION SNEI IN pipe.EX
{
    DECLARE { REFERENCE rd; }
    CODING { 0b011001 }
    SYNTAX { "SNEI" }
    BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).reg_a; IN PIPELINE_REGISTER(pipe,DC/EX).imm;
                   OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
        {
        PIPELINE_REGISTER(pipe,EX/MEM).alu = (PIPELINE_REGISTER(pipe,DC/EX).reg_a != PIPELINE_REGISTER(pipe,DC/EX).imm) ? 1 : 0;
        }
}
```

j_type.lisa

```
OPERATION J IN pipe.EX
{
  DECLARE { REFERENCE addr; }
  CODING { 0b000010 }
  SYNTAX { "J" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).npc; OUT PIPELINE_REGISTER(pipe,EX/MEM).cond;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
  {
    PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).npc + addr;
    PIPELINE_REGISTER(pipe,EX/MEM).cond = true;
  }
}

OPERATION JAL IN pipe.EX
{
  DECLARE { REFERENCE addr; }
  CODING { 0b000011 }
  SYNTAX { "JAL" }
  BEHAVIOR USES (IN PIPELINE_REGISTER(pipe,DC/EX).npc; OUT PIPELINE_REGISTER(pipe,EX/MEM).cond;
                 OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
    {
      R[31] = PIPELINE_REGISTER(pipe,DC/EX).npc;
      PIPELINE_REGISTER(pipe,EX/MEM).alu = PIPELINE_REGISTER(pipe,DC/EX).npc + addr;
      PIPELINE_REGISTER(pipe,EX/MEM).cond = true;
    }
}

OPERATION TRAP IN pipe.EX
{
  CODING { 0b010001 }
  SYNTAX { "TRAP" }
  BEHAVIOR { halted_flag = true; }
}

OPERATION RFE IN pipe.EX
{
  CODING { 0b010000 }
  SYNTAX { "RFE" }
  BEHAVIOR USES (OUT PIPELINE_REGISTER(pipe,EX/MEM).cond; OUT PIPELINE_REGISTER(pipe,EX/MEM).alu;)
  {
    PIPELINE_REGISTER(pipe,EX/MEM).alu = R[31];
    PIPELINE_REGISTER(pipe,EX/MEM).cond = true;
  }
}
```

memory_access.lisa

```
#include "dlx.h"

OPERATION LB IN pipe.MEM
{
  DECLARE {
    REFERENCE rd;
    INSTANCE load_register;
  }
  CODING { 0b100000 }
  SYNTAX { "LB" }
  BEHAVIOR USES ( IN PIPELINE_REGISTER(pipe,EX/MEM).alu; )
    {
      lmd = SIGN_EXTEND(dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu], 8);
    }
  ACTIVATION { load_register }
}

OPERATION LBU IN pipe.MEM
{
  DECLARE {
    REFERENCE rd;
    INSTANCE load_register;
  }
  CODING { 0b100100 }
  SYNTAX { "LBU" }
  BEHAVIOR { lmd = (unsigned) dmem[alu]; }
  ACTIVATION { load_register }
}

OPERATION LH IN pipe.MEM
{
  DECLARE {
    REFERENCE rd;
    INSTANCE load_register;
  }
  CODING { 0b100001 }
  SYNTAX { "LH" }
  BEHAVIOR USES ( IN PIPELINE_REGISTER(pipe,EX/MEM).alu; )
    {
      lmd = SIGN_EXTEND( dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu] | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+1] << 8), 16);
    }
  ACTIVATION { load_register }
}

OPERATION LHU IN pipe.MEM
{
  DECLARE {
    REFERENCE rd;
    INSTANCE load_register;
  }
  CODING { 0b100101 }
  SYNTAX { "LHU" }
  BEHAVIOR USES ( IN PIPELINE_REGISTER(pipe,EX/MEM).alu; )
    {
      lmd = dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu] | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+1] << 8);
    }
  ACTIVATION { load_register }
}

OPERATION LW IN pipe.MEM
{
  DECLARE {
    REFERENCE rd;
    INSTANCE load_register;
  }
  CODING { 0b100011 }
  SYNTAX { "LW" }
  BEHAVIOR USES ( IN PIPELINE_REGISTER(pipe,EX/MEM).alu; )
    {
      lmd = dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu] | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+1] << 8)
          | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+2] << 16) | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+3] << 24);
    }
  ACTIVATION { load_register }
```

```
}
OPERATION LF IN pipe.MEM
{
  DECLARE {
    REFERENCE rd;
    INSTANCE load_float_register;
  }
  CODING { 0b100110 }
  SYNTAX { "LF" }
  BEHAVIOR USES { IN PIPELINE_REGISTER(pipe,EX/MEM).alu; )
    {
      lmd = dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu] | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+1] << 8)
          | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+2] << 16) | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+3] << 24);
    }
  ACTIVATION { load_float_register }
}
OPERATION LD IN pipe.MEM
{
  DECLARE {
    REFERENCE rd;
    INSTANCE load_double_register;
  }
  CODING { 0b100111 }
  SYNTAX { "LD" }
  BEHAVIOR USES { IN PIPELINE_REGISTER(pipe,EX/MEM).alu; )
    {
      lmd = dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu] | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+1] << 8)
        | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+2] << 16) | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+3] << 24);
      lmd2 = dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+4] | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+5] << 8)
        | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+6] << 16) | (dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+7] << 24);
    }
  ACTIVATION { load_double_register }
}
OPERATION SB IN pipe.MEM
{
  DECLARE { REFERENCE rd; }
  CODING { 0b101000 }
  SYNTAX { "SB" }
  BEHAVIOR USES { IN PIPELINE_REGISTER(pipe,EX/MEM).alu; )
    {
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu] = (reg_b & 0xFF);
    }
}
OPERATION SH IN pipe.MEM
{
  DECLARE { REFERENCE rd; }
  CODING { 0b101001 }
  SYNTAX { "SH" }
  BEHAVIOR USES { IN PIPELINE_REGISTER(pipe,EX/MEM).alu; )
    {
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu  ] = ( reg_b        & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+1] = ((reg_b >>  8) & 0xFF);
    }
}
OPERATION SW IN pipe.MEM
{
  DECLARE { REFERENCE rd; }
  CODING { 0b101011 }
  SYNTAX { "SW" }
  BEHAVIOR  USES ( IN PIPELINE_REGISTER(pipe,EX/MEM).alu; )
    {
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu  ] = ( reg_b        & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+1] = ((reg_b >>  8) & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+2] = ((reg_b >> 16) & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+3] = ((reg_b >> 24) & 0xFF);
    }
}
OPERATION SF IN pipe.MEM
{
  DECLARE { REFERENCE rd; }
  CODING { 0b101110 }
  SYNTAX { "SF" }
  BEHAVIOR USES { IN PIPELINE_REGISTER(pipe,EX/MEM).alu; )
    {
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu  ] = ( float2int( reg_b_f )        & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+1] = ((float2int( reg_b_f ) >>  8) & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+2] = ((float2int( reg_b_f ) >> 16) & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+3] = ((float2int( reg_b_f ) >> 24) & 0xFF);
    }
}
OPERATION SD IN pipe.MEM
{
  DECLARE { REFERENCE rd; }
  CODING { 0b101111 }
  SYNTAX { "SD" }
  BEHAVIOR USES { IN PIPELINE_REGISTER(pipe,EX/MEM).alu; )
    {
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu  ] = ( double2int1( reg_b_d )        & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+1] = ((double2int1( reg_b_d ) >>  8) & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+2] = ((double2int1( reg_b_d ) >> 16) & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+3] = ((double2int1( reg_b_d ) >> 24) & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+4] = ( double2int2( reg_b_d )        & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+5] = ((double2int2( reg_b_d ) >>  8) & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+6] = ((double2int2( reg_b_d ) >> 16) & 0xFF);
      dmem[PIPELINE_REGISTER(pipe,EX/MEM).alu+7] = ((double2int2( reg_b_d ) >> 24) & 0xFF);
    }
}
```

write_back.lisa

```
OPERATION writeback_register IN pipe.WB
{
  DECLARE { REFERENCE rd; }
  BEHAVIOR USES ( IN PIPELINE_REGISTER(pipe,MEM/WB).alu; )
    {
      rd = PIPELINE_REGISTER(pipe,MEM/WB).alu;
    }
}
OPERATION writeback_float_register IN pipe.WB
{
  DECLARE { REFERENCE rd; }
  BEHAVIOR USES ( IN PIPELINE_REGISTER(pipe,MEM/WB).alu_f; )
    {
      rd = PIPELINE_REGISTER(pipe,MEM/WB).alu_f;
```

```
    }
}

OPERATION writeback_double_register IN pipe.WB
{
  DECLARE { REFERENCE rd; }
  BEHAVIOR USES ( IN PIPELINE_REGISTER(pipe,MEM/WB).alu_d; )
    {
      rd = PIPELINE_REGISTER(pipe,MEM/WB).alu_d;
    }
}

OPERATION writeback_special_register IN pipe.WB
{
  DECLARE { REFERENCE rd; }
  BEHAVIOR USES ( IN PIPELINE_REGISTER(pipe,MEM/WB).alu; )
    {
      rd = PIPELINE_REGISTER(pipe,MEM/WB).alu;
    }
}

OPERATION load_register IN pipe.WB
{
  DECLARE { REFERENCE rd; }
  BEHAVIOR { rd = lmd; }
}

OPERATION load_float_register IN pipe.WB
{
  DECLARE { REFERENCE rd; }
  BEHAVIOR { rd = int2float( lmd ); }
}

OPERATION load_double_register IN pipe.WB
{
  DECLARE { REFERENCE rd; }
  BEHAVIOR { rd = int2double( lmd, lmd2 ); }
}
```

Appendix F

LISA description of the Texas Instruments C62x

main.lisa

```
#include <c6201.h>
ARCHITECTURE C62x {}
RESOURCE
{
  MEMORY_MAP
    {
      // External Memory Space CE0
      0x00000000 -> 0x01000000, BYTES(1) : ext_mem_ce0[0x0..0x400000].word, BYTES(4);
      // External Memory Space CE1
      0x01000000 -> 0x01400000, BYTES(1) : /* empty */ ;
      // Internal Program Memory
      0x01400000 -> 0x01430000, BYTES(1) : int_prog_mem[0x0..0xC000], BYTES(4);
      // Internal Peripheral Space
      0x01800000 -> 0x01C00000, BYTES(1) : int_periph[0x0..0x4000].word, BYTES(4);
      // External Memory Space CE2
      0x02000000 -> 0x03000000, BYTES(1) : /* empty */;
      // External Memory Space CE3
      0x03000000 -> 0x04000000, BYTES(1) : ext_data_mem[0x0..0x400000].word, BYTES(4);
      // Internal Data Memory
      0x80000000 -> 0x80030000, BYTES(1) : int_data_mem[0x0..0xC000].word, BYTES(4);
    }
  PIPELINE pipe = { PG; PS; PW; PR; DP; DC; E1; E2; E3; E4; E5 };
  insn_t  PG, PS, PW, PR([0..7]), DP([0..7]), DC, E1[0..7], E2[0..7], E3[0..7], E4[0..7], E5[0..7];
  //  functional units
  data_t    L_unit[0..1];
  data_t    S_unit[0..1];
  data_t    M_unit[0..1];
  data_t    D_unit[0..1];
  //  data pipeline of S unit
  data_t    S_data[0..1];
  //  data pipeline of M unit
  data_t    M_data[0..1];
  //  data pipeline of D unit
  data_t    D_data[0..1][1..4];
  //  counter of parallel load instructions
  int       ld_count[0..1];
  //  data pipeline of sat bit
  data_t    sat_bit;
  //  register file
  REGISTER data_t    A[0..15], B[0..15];
  //  registers cross-paths
  data_t    X_path1, X_path2;
  //  program memory is organized in 8 instructions per address
  PROGRAM_MEMORY insn_t  int_prog_mem[0x0..0xC000];
  //  data memory is organized in 4 memory banks
  DATA_MEMORY memory_t  int_data_mem[0x0..0xC000];
  //  internal peripheral space
  DATA_MEMORY memory_t  int_periph[0x0..0x4000];
  // external memory CE0
  PROGRAM_MEMORY memory_t ext_mem_ce0[0x0..0x400000];
  // external data memory
  DATA_MEMORY memory_t ext_data_mem[0x0..0x400000];
  //  data bus
  addr_t    data_read_addr_bus;
  data_t    data_read_bus[0..1][0..3];
  data_t    memory_read[0..1];
  addr_t    data_write_addr_bus;
  data_t    data_write_bus;
  //  program fetch counter
  addr_t  PFC;
  //  program word counter
  addr_t    PWC;
  data_t new_fp;
  //  set of control registers
  amr_t     amr;
  csr_t     csr;
  ifr_t     ifr;
  isr_t     isr;
  icr_t     icr;
  ier_t     ier;
  istp_t    istp;
  addr_t    irp;
  addr_t    nrp;
  PROGRAM_COUNTER addr_t    pce1;
  addr_t    pce1_forward[0..4];
  addr_t    cycle;
  addr_t    cycles2add;
  //  instruction index with value range (0..7) points to the
  //  first instruction of the current execute packet within
  //  the fetch packet
  U32       pipeline_stall;             //  true, if pipeline completely stalled
                                        //  - can be initiated by memory wait
```

```
                                          //  cycles
  U32       dispatch_complete;            //  true, if fetch packet finishes DP
  U32       multicycle_nop;               //  > 0,  if multicycle-NOP executes
                                          //  = -1, if IDLE executes
  U32       new_addr_set;                 //  true, if new program address is set
  U32       branch_active;                //  true, if a branch is active
  U32       offset_addr;                  // Offset for branching to instruction within a fetch-packet
  U32       Branch_in_E5, Branch_in_E4, Branch_in_E3, Branch_in_E2, Branch_in_E1;
  U32       flush_DP, flush_DC;
  // For stand-alone simulation without frontend
  U32 prog_count_real;
  // Instruction-counter
  unsigned bit[32] instruction_counter;
  PIPELINE_REGISTER IN pipe
    {
      int a;
      REGISTER addr_t b;
      REGISTER long ir;
    };
}

OPERATION reset
{
  BEHAVIOR
    {
      PIPELINE(pipe).flush();
      pipeline_stall = 0;
      dispatch_complete = 1;
      multicycle_nop = 0;
      new_addr_set = 1;
      branch_active = 0;
      PFC = LISA_PROGRAM_COUNTER;
      cycle = 0;
      cycles2add = 0;
      data_read_bus[0][0] = 0; data_read_bus[0][1] = 0; data_read_bus[0][2] = 0; data_read_bus[0][3] = 0;
      data_read_bus[1][0] = 0; data_read_bus[1][1] = 0; data_read_bus[1][2] = 0; data_read_bus[1][3] = 0;
      memory_read[0] = 0; memory_read[1] = 0;
      amr.word = 0;
      csr.word = 0X10100; // (CPU_ID << 24) + (REV_ID << 16);
      isr.word = 0;
      icr.word = 0;
      ifr.word = 0;
      istp.word = 0;
      ier.word = 1;
      new_fp = 1;
      Branch_in_E5 = 0;
      Branch_in_E4 = 0;
      Branch_in_E3 = 0;
      Branch_in_E2 = 0;
      Branch_in_E1 = 0;
      flush_DP = 0;
      flush_DC = 0;
      prog_count_real = 0;
      instruction_counter = 0;
      // First cycle
      // Second cycle
      Program_Address_Send(0);
      Program_Address_Generate(0);
      // Third cycle
      Program_Access_Ready_Wait(0);
      Program_Address_Send(0);
      Program_Address_Generate(0);
      // Fourth cycle
      Program_Fetch_Packet_Receive(0);
      Program_Access_Ready_Wait(0);
      Program_Address_Send(0);
      Program_Address_Generate(0);
      cycle = 4;
    }
}

OPERATION main
{
  DECLARE
    {
      INSTANCE Program_Address_Generate, Program_Address_Send, Program_Access_Ready_Wait;
      INSTANCE Program_Fetch_Packet_Receive, Dispatch, Interrupt_Detection, Next_Cycle;
    }
  ACTIVATION
    {
      if ((dispatch_complete && !multicycle_nop) || (Branch_in_E5 && dispatch_complete) || Branch_in_E4)
        {
          Program_Address_Generate,
          Program_Address_Send,
          Program_Access_Ready_Wait,
          Program_Fetch_Packet_Receive,
          Dispatch
        }
      else
        {
          if(Branch_in_E3)
            {
              Program_Address_Generate,
              Program_Address_Send,
              Program_Access_Ready_Wait,
              Program_Fetch_Packet_Receive
            }
          else
            {
              if(Branch_in_E2)
                {
                  Program_Address_Generate,
                  Program_Address_Send,
                  Program_Access_Ready_Wait
                }
              else
                {
                  if(Branch_in_E1)
                    {
                      Program_Address_Generate,
                      Program_Address_Send
                    }
                }
            }
        }
      , // parallel
      if ((ifr.word) && (! branch_active))
        {
          Interrupt_Detection
        }
      , // parallel
      Next_Cycle
    }
  BEHAVIOR
    {
      // Reset signal dispatch_complete
      if(!multicycle_nop || (dispatch_complete && Branch_in_E5))
        dispatch_complete = 0;
```

```
        // Reset branch_active flags for the respective stages
        Branch_in_E1 = Branch_in_E2 = Branch_in_E3 = Branch_in_E4 = Branch_in_E5 = 0;
        // Reset load count signal
        ld_count[0] = 0;
        PIPELINE(pipe).execute();
        if (multicycle_nop)
          {
            if(!flush_DP && !flush_DC)
              PIPE_REGISTER (DP, pipe).stall ();
            if(!flush_DC)
              PIPE_REGISTER (DC, pipe).stall ();
          }
        if(flush_DP)
          {
            if(multicycle_nop)
              {
                PIPE_REGISTER(DP,pipe).flush();
                dispatch_complete = 0;
                flush_DP--;
              }
          }
        if (flush_DC)
          {
            if(multicycle_nop)
              {
                PIPE_REGISTER (DC, pipe).flush ();
              }
            flush_DC--;
          }
        PIPELINE(pipe).shift();
        ld_count[1] = ld_count[0];
        if(flush_DP && !multicycle_nop)
          {
            PIPE_REGISTER(DP,pipe).flush(DELAYED);
            flush_DP--;
          }
      }
  }
  OPERATION Interrupt_Detection
  {
    BEHAVIOR
      {
        if (ier.bit.nmi)
          {
            if (ifr.bit.nmi)
              {
                //  NMIE processing
                Non_Maskable_Interrupt (ier, pce1, PFC, istp, new_addr_set, nrp);
                ifr.bit.nmi = 0;
              }
            else if (csr.bit.gie)
              {
                if ((ifr.bit.int4) && (ier.bit.int4))
                  {
                    Maskable_Interrupt (4, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int4 = 0;
                  }
                else if ((ifr.bit.int5) && (ier.bit.int5))
                  {
                    Maskable_Interrupt (5, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int5 = 0;
                  }
                else if ((ifr.bit.int6) && (ier.bit.int6))
                  {
                    Maskable_Interrupt (6, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int6 = 0;
                  }
                else if ((ifr.bit.int7) && (ier.bit.int7))
                  {
                    Maskable_Interrupt (7, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int7 = 0;
                  }
                else if ((ifr.bit.int8) && (ier.bit.int8))
                  {
                    Maskable_Interrupt (8, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int8 = 0;
                  }
                else if ((ifr.bit.int9) && (ier.bit.int9))
                  {
                    Maskable_Interrupt (9, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int9 = 0;
                  }
                else if ((ifr.bit.int10) && (ier.bit.int10))
                  {
                    Maskable_Interrupt (10, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int10 = 0;
                  }
                else if ((ifr.bit.int11) && (ier.bit.int11))
                  {
                    Maskable_Interrupt (11, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int11 = 0;
                  }
                else if ((ifr.bit.int12) && (ier.bit.int12))
                  {
                    Maskable_Interrupt (12, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int12 = 0;
                  }
                else if ((ifr.bit.int13) && (ier.bit.int13))
                  {
                    Maskable_Interrupt (13, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int13 = 0;
                  }
                else if ((ifr.bit.int14) && (ier.bit.int14))
                  {
                    Maskable_Interrupt (14, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int14 = 0;
                  }
                else if ((ifr.bit.int15) && (ier.bit.int15))
                  {
                    Maskable_Interrupt (15, csr, irp, pce1, PFC, istp, new_addr_set);
                    ifr.bit.int15 = 0;
                  }
              }
          }
      }
  }
  OPERATION Next_Cycle
  {
    BEHAVIOR
      {
        if (cycles2add)
          {
            cycle += cycles2add;
            cycles2add = 0;
          }
        cycle++;
      }
  }
```

fetch.lisa

```
#include <c6201.h>
OPERATION Program_Address_Generate IN pipe.PG
{
  BEHAVIOR REQUIRES ( !pipeline_stall )
     {
      if (!Branch_in_E1)
          PFC = (PFC & PFC_MASK) + 32;
     }
}

OPERATION Program_Address_Send IN pipe.PS
{
  BEHAVIOR USES ( IN PIPELINE_REGISTER(pipe,PG/PS).ir; OUT PIPELINE_REGISTER(pipe,PS/PW).b;
                  INOUT PIPELINE_REGISTER(pipe,PG/PS).a;)
     {
      pce1_forward[0] = PFC;
     }
}

OPERATION Program_Access_Ready_Wait IN pipe.PW
{
  BEHAVIOR REQUIRES ( !pipeline_stall ) { pce1_forward[1] = pce1_forward[0]; }
}

OPERATION Program_Fetch_Packet_Receive IN pipe.PR
{
  BEHAVIOR REQUIRES ( !pipeline_stall )
     {
      offset_addr = ((pce1_forward[2] = pce1_forward[1]) & 0x1F) >> 2;
     }
}
```

dispatch.lisa

```
#include <c6201.h>
#include <protos.h>

OPERATION Dispatch IN pipe.DP
{
  DECLARE
     {
      GROUP Insn1, Insn2, Insn3, Insn4,
            Insn5, Insn6, Insn7, Insn8 = { Ser_Insn || Par_Insn};
      INSTANCE Dispatch_C;
     }
  CODING AT (pce1_forward[2] & PFC_MASK)
     {
         (PR[0] == Insn1) && (PR[1] == Insn2) && (PR[2] == Insn3) && (PR[3] == Insn4)
      && (PR[4] == Insn5) && (PR[5] == Insn6) && (PR[6] == Insn7) && (PR[7] == Insn8)
     }
   SYNTAX
     {
      Insn1  EOI:1  Insn2  EOI:1  Insn3  EOI:1  Insn4  EOI:1  Insn5  EOI:1  Insn6  EOI:1  Insn7  EOI:1  Insn8  EOI:1
     }
  BEHAVIOR REQUIRES (!pipeline_stall)
     {
       DP[0] = PR[0];
       DP[1] = PR[1];
       DP[2] = PR[2];
       DP[3] = PR[3];
       DP[4] = PR[4];
       DP[5] = PR[5];
       DP[6] = PR[6];
       DP[7] = PR[7];
       pce1_forward[3]  = pce1_forward[2];
       new_fp = 1;
     }
  IF (Insn1 == Ser_Insn)
  THEN { ACTIVATION { if (offset_addr < 1) { Insn1; } } }
  ELSE { ACTIVATION { if (offset_addr < 1) { Insn1, } } }
  IF (Insn2 == Ser_Insn)
  THEN { ACTIVATION { if (offset_addr < 2) { Insn2; } } }
  ELSE { ACTIVATION { if (offset_addr < 2) { Insn2, } } }
  IF (Insn3 == Ser_Insn)
  THEN { ACTIVATION { if (offset_addr < 3) { Insn3; } } }
  ELSE { ACTIVATION { if (offset_addr < 3) { Insn3, } } }
  IF (Insn4 == Ser_Insn)
  THEN { ACTIVATION { if (offset_addr < 4) { Insn4; } } }
  ELSE { ACTIVATION { if (offset_addr < 4) { Insn4, } } }
  IF (Insn5 == Ser_Insn)
  THEN { ACTIVATION { if (offset_addr < 5) { Insn5; } } }
  ELSE { ACTIVATION { if (offset_addr < 5) { Insn5, } } }
  IF (Insn6  == Ser_Insn)
  THEN { ACTIVATION { if (offset_addr < 6) { Insn6; } } }
  ELSE { ACTIVATION { if (offset_addr < 6) { Insn6, } } }
  IF (Insn7 == Ser_Insn)
  THEN { ACTIVATION { if (offset_addr < 7) {  Insn7; } } }
  ELSE { ACTIVATION { if (offset_addr < 7) {  Insn7, } } }
  ACTIVATION { Insn8, Dispatch_C }
}

OPERATION Dispatch_C
{
  BEHAVIOR { dispatch_complete = 1; }
}

OPERATION Ser_Insn IN pipe.DP
{
  DECLARE { INSTANCE Decode_Instruction; }
  CODING { Decode_Instruction  0b }
  SYNTAX { Decode_Instruction  }
  ACTIVATION { Decode_Instruction }
}

OPERATION Par_Insn IN pipe.DP
{
  DECLARE { INSTANCE Decode_Instruction; }
  CODING { Decode_Instruction  1b }
  SYNTAX { Decode_Instruction  "||"  }
  ACTIVATION { Decode_Instruction }
}
```

decode.lisa

```
#include <c6201.h>
```

```
OPERATION Decode_Instruction IN pipe.DC
{
  DECLARE
    {
      GROUP Instruction_Type = { Insn_L_unit || Insn_M_unit || Insn_D_unit || LD_15b || LD_baseR || ST_15b || ST_baseR ||
                               Insn_S_unit || ADDK || Field_Op || MV_D || ZERO_D || MVK || MVKH || MVKLH || BRANCH || NOP };
      INSTANCE E1_stage;
    }
  CODING { Instruction_Type }
  SYNTAX { Instruction_Type }
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      pce1_forward[4] = pce1_forward[3];
      if(new_fp)
        {
          pce1 = pce1_forward[3];
          new_fp--;
        }
    }
  ACTIVATION { Instruction_Type, E1_stage }
}

OPERATION Side1 { CODING { 0b } SYNTAX { "1" } EXPRESSION { 0 } }
OPERATION Side2 { CODING { 1b } SYNTAX { "2" } EXPRESSION { 1 } }
OPERATION No_Xpath { CODING { 0b } SYNTAX { " " } }
OPERATION Is_Xpath { CODING { 1b } SYNTAX { "X" } }

OPERATION Reg
{
  SUBOPERATION Int
    {
      DECLARE { REFERENCE Side; LABEL index; }
      CODING { 0bx  index=0bx[4] }
      SWITCH (Side)
        {
          CASE Side1: {
              SYNTAX { "A"  ~index=#U }
              EXPRESSION { A[index] }
            }
          CASE Side2: {
              SYNTAX { "B"  ~index=#U }
              EXPRESSION { B[index] }
            }
        }
    }
  SUBOPERATION Long
    {
      DECLARE { REFERENCE Side; LABEL index; }
      CODING { 0bx  index=(0bx[3]<<1)  0b0 }
      SWITCH (Side)
        {
          CASE Side1: {
              SYNTAX { "A" (index+1)=#U ":A" index=#U }
              EXPRESSION { A[index + 1] }
              BEHAVIOR { int a; }
            }
          CASE Side2: {
              SYNTAX { "B" (index+1)=#U ":B" index=#U }
              EXPRESSION { B[index + 1] }
              BEHAVIOR { int b; }
            }
        }
    }
  SUBOPERATION UCst5
    {
      DECLARE { LABEL value; }
      CODING { value=0bx[5] }
      SYNTAX { SYMBOL(value=#U5) }
      EXPRESSION { value }
    }
  SUBOPERATION SCst5
    {
      DECLARE { LABEL value; }
      CODING { value=0bx[5] }
      SYNTAX { SYMBOL(value=#S5) }
      EXPRESSION { value }
    }
  SUBOPERATION UCst4
    {
      DECLARE { LABEL value; }
      CODING { 0bx  value=0bx[4] }
      SYNTAX { SYMBOL(value=#U4) }
      EXPRESSION { value }
    }
}

OPERATION XReg
{
  SUBOPERATION Int
    {
      DECLARE { REFERENCE Side, Xpath; LABEL index; }
      CODING { 0bx  index=0bx[4] }
      SWITCH (Side)
        {
          CASE Side1: {
              SWITCH (Xpath)
                {
                  CASE No_Xpath:
                    {
                      SYNTAX { "A"  ~index=#U4 }
                      EXPRESSION { A[index] }
                    }
                  CASE Is_Xpath:
                    {
                      SYNTAX { "B"  ~index=#U4 }
                      EXPRESSION { B[index] }
                    }
                }
            }
          CASE Side2: {
              SWITCH (Xpath)
                {
                  CASE No_Xpath:
                    {
                      SYNTAX { "B"  ~index=#U4 }
                      EXPRESSION { B[index] }
                    }

                  CASE Is_Xpath:
                    {
                      SYNTAX { "A"  ~index=#U4 }
                      EXPRESSION { A[index] }
                    }
                }
            }
        }
    }
  SUBOPERATION Long
    {
      DECLARE { GROUP register = { Reg }; }
      CODING { register }
```

```
      SYNTAX { register.Long }
      EXPRESSION { register.Long }
    }
}

OPERATION Minus_one
{
  CODING { 11111b }
  SYNTAX { "-1" }
  EXPRESSION { -1 }
}

OPERATION Addr_Register
{
  SUBOPERATION Reg
    {
      DECLARE { REFERENCE Yside; LABEL index; }
      CODING { 0bx  index=0bx[4] }
      SWITCH (Yside)
        {
          CASE Side1: {
              SYNTAX { "A"  ~index=#U }
              EXPRESSION { A[index] }
            }
          CASE Side2: {
              SYNTAX { "B"  ~index=#U }
              EXPRESSION { B[index] }
            }
        }
    }
  SUBOPERATION Idx
    {
      DECLARE { LABEL index; }
      CODING { 0bx  index=0bx[4] }
      EXPRESSION { index }
    }
}

OPERATION B14 { CODING { 0b0 } SYNTAX { "B14" } EXPRESSION { B[14] } }
OPERATION B15 { CODING { 0b1 } SYNTAX { "B15" } EXPRESSION { B[15] } }
OPERATION AMR { CODING { 00000b } SYNTAX { "AMR" } EXPRESSION { (U32) amr.word } }
OPERATION CSR { CODING { 00001b } SYNTAX { "CSR" } EXPRESSION { (U32) csr.word } }
OPERATION IFR { CODING { 00010b } SYNTAX { "IFR" } EXPRESSION { (U32) ifr.word } }
OPERATION ISR { CODING { 00010b } SYNTAX { "ISR" } EXPRESSION { (U32) isr.word } }
OPERATION ICR { CODING { 00011b } SYNTAX { "ICR" } EXPRESSION { (U32) icr.word } }
OPERATION IER { CODING { 00100b } SYNTAX { "IER" } EXPRESSION { (U32) ier.word } }
OPERATION ISTP { CODING { 00101b } SYNTAX { "ISTP" } EXPRESSION { (U32) istp.word } }
OPERATION IRP { CODING { 00110b } SYNTAX { "IRP" } EXPRESSION { (U32) irp.word } }
OPERATION NRP { CODING { 00111b } SYNTAX { "NRP" } EXPRESSION { (U32) nrp.word } }
OPERATION PCE1 { CODING { 10000b } SYNTAX { "PCE1" } EXPRESSION { pce1 } }

OPERATION Addr_Offset
{
  SUBOPERATION Reg
    {
      DECLARE { REFERENCE Side; LABEL index; }
      CODING { 0bx  index=0bx[4] }
      SWITCH (Side)
        {
          CASE Side1: {
              SYNTAX { "["  ~"A"  ~index=#U  ~"]" }
              EXPRESSION { A[index] }
            }
          CASE Side2: {
              SYNTAX { "["  ~"B"  ~index=#U  ~"]" }
              EXPRESSION { B[index] }
            }
        }
    }
  SUBOPERATION UCst5
    {
      DECLARE { LABEL value; }
      CODING { value=0bx[5] }
      SYNTAX { "["  ~SYMBOL(value=#U)  ~"]" }
      EXPRESSION { value }
    }
}

OPERATION Addr_Offset15b
{
  DECLARE { LABEL value; }
  CODING { value=0bx[15] }
  SYNTAX { "["  ~SYMBOL(value=#U)  ~"]" }
  EXPRESSION { value }
}

OPERATION Pos_Ofs_Reg { CODING { 0101b } }
OPERATION Neg_Ofs_Reg { CODING { 0100b } }
OPERATION Pre_Incr_Reg { CODING { 1101b } }
OPERATION Pre_Decr_Reg { CODING { 1100b } }
OPERATION Post_Incr_Reg { CODING { 1111b } }
OPERATION Post_Decr_Reg { CODING { 1110b } }
OPERATION Pos_Ofs_Const { CODING { 0001b } }
OPERATION Neg_Ofs_Const { CODING { 0000b } }
OPERATION Pre_Incr_Const { CODING { 1001b } }
OPERATION Pre_Decr_Const { CODING { 1000b } }
OPERATION Post_Incr_Const { CODING { 1011b } }
OPERATION Post_Decr_Const { CODING { 1010b } }

OPERATION Is_Unconditional { CODING { 0b000 } }
OPERATION Is_Conditional
{
  DECLARE { INSTANCE Condition_Register; }
  CODING { Condition_Register }
  SYNTAX { Condition_Register }
  EXPRESSION { Condition_Register }
}

OPERATION Is_Zero { CODING { 1b } }
OPERATION Is_Nonzero { CODING { 0b } }

OPERATION Condition_Register
{
  DECLARE { ENUM SELF = { B0,  B1,  B2,  A1,  A2 }; }
  SWITCH (SELF)
    {
      CASE B0: { CODING { 001b }  SYNTAX { "B0" }  EXPRESSION { B[0] }}
      CASE B1: { CODING { 010b }  SYNTAX { "B1" }  EXPRESSION { B[1] }}
      CASE B2: { CODING { 011b }  SYNTAX { "B2" }  EXPRESSION { B[2] }}
      CASE A1: { CODING { 100b }  SYNTAX { "A1" }  EXPRESSION { A[1] }}
      CASE A2: { CODING { 101b }  SYNTAX { "A2" }  EXPRESSION { A[2] }}
    }
}
```

execute.lisa

```
#include <c6201.h>
```

```
OPERATION E1_stage IN pipe.E1
{
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      instruction_counter++;
      pce1 += 4;
    }
}

OPERATION B_E5 IN pipe.E5
{
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      flush_DC = 1;
      Branch_in_E5 = 1;
      branch_active--;
    }
}

OPERATION LD_E2 IN pipe.E2
{
  DECLARE
    {
      INSTANCE EMIF;
      REFERENCE Side;
    }
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      // are there two memory read accesses at the same time?
      if (ld_count[1] > 1)
        {
          register U32 mem1 = (memory_read[0] >> 16);
          register U32 mem2 = (memory_read[1] >> 16);
          //  are both accesses targeting the same internal memory space?
          if (    (   (mem1 == 0x8000) && (mem2 == 0x8000))
              || (   (mem1 == 0x0140) && (mem2 == 0x0140))
              || (   (mem1 >= 0x0180) && (mem1 <  0x01C0) && (mem2 >= 0x0180) && (mem2 <  0x01C0)))
            {
              if (   (data_read_bus[0][0] & data_read_bus[1][0]) || (data_read_bus[0][1] & data_read_bus[1][1])
                  || (data_read_bus[0][2] & data_read_bus[1][2]) || (data_read_bus[0][3] & data_read_bus[1][3]))
                  cycles2add++;
            }
          ld_count[1]--;
        }
      else if(ld_count[1] == 1)
        {
          data_read_bus[0][0] = 0; data_read_bus[0][1] = 0; data_read_bus[0][2] = 0; data_read_bus[0][3] = 0;
          data_read_bus[1][0] = 0; data_read_bus[1][1] = 0; data_read_bus[1][2] = 0; data_read_bus[1][3] = 0;
        }
      EMIF ();
      D_data[Side][2] = D_data[Side][1];
    }
}

OPERATION B_E2 IN pipe.E2
{
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      Branch_in_E2 = 1;
    }
}

OPERATION LD_E3 IN pipe.E3
{
  DECLARE { REFERENCE Side; }
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      D_data[Side][3] = D_data[Side][2];
    }
}

OPERATION B_E3 IN pipe.E3
{
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      Branch_in_E3 = 1;
    }
}

OPERATION LD_E4 IN pipe.E4
{
  DECLARE { REFERENCE Side; }
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      D_data[Side][4] = D_data[Side][3];
    }
}

OPERATION B_E4 IN pipe.E4
{
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      Branch_in_E4 = 1;
      flush_DP = 1;
    }
}

OPERATION LD_E5 IN pipe.E5
{
  DECLARE { REFERENCE Side, Dest; }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT Dest.Int; )
    {
      Dest.Int = D_data[Side][4];
    }
}

OPERATION MPY_E2 IN pipe.E2
{
  DECLARE { REFERENCE Side, Dest; }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT Dest.Int; )
    {
      Dest.Int = M_data[Side];
    }
}

OPERATION MVC_ISR_E2 IN pipe.E2
{
  DECLARE { REFERENCE Side; }
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      isr.word |= LO16 (S_data[Side]);
    }
}

OPERATION MVC_ICR_E2 IN pipe.E2
{
  DECLARE { REFERENCE Side; }
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      icr.word &= (~S_data[Side] | 0xFFFF0000);
    }
```

```
}
OPERATION SAT_E2 IN pipe.E2
{
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      if (sat_bit)
        csr.bit.sat = 1;
    }
}
OPERATION EMIF
{
  DECLARE { REFERENCE Side; }
  BEHAVIOR
    {
      register U32 mem = (memory_read[Side] >> 16);
      register U32 sscrt = (int_periph[0].word & 4);
      /*  CE 0 */
      if (   (mem >= 0x0000) && (mem <  0x0100))
        {
          if (sscrt)
            cycles2add = 13;
          else
            cycles2add = 17;
        }
      /*  CE 1 */
      else if (   (mem >= 0x0100) && (mem <  0x0140))
        {
          if (sscrt)
            cycles2add = 13;
          else
            cycles2add = 17;
        }
      /*  CE 2 */
      else if (   (mem >= 0x0200) && (mem <  0x0300))
        {
          if (sscrt)
            cycles2add = 13;
          else
            cycles2add = 17;
        }
      /*  CE 3 */
      else if (   (mem >= 0x0300) && (mem <  0x0400))
        {
          if (sscrt)
            cycles2add = 13;
          else
            cycles2add = 17;
        }
    }
}
OPERATION End_Multi_Cycle_NOP
{
  BEHAVIOR REQUIRES ( !pipeline_stall )
    {
      multicycle_nop = 0;
      pce1 += 4;
    }
}
```

d_unit.lisa

```
OPERATION Insn_D_unit IN pipe.E1
{
  DECLARE
    {
      GROUP Condition = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Side  = { Side1  ||  Side2 };
      GROUP Dest, Src1, Src2 = { Reg };
      GROUP Instruction = { ADD_D_iii || ADD_D_ici || ADDAB_iii || ADDAH_iii || ADDAW_iii || ADDAB_ici || ADDAH_ici ||
         ADDAW_ici || SUB_D_iii || SUB_D_ici || SUBAB_iii || SUBAH_iii || SUBAW_iii || SUBAB_ici || SUBAH_ici || SUBAW_ici };
    }
  CODING { Condition  z  Dest  Src2  Src1  Instruction  10000b  Side }
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" Instruction }
          ACTIVATION { if (!Condition) { Instruction } }
          BEHAVIOR USES ( OUT D_unit [Side]; )
        }
      ELSE
        {
          SYNTAX { " [" ~Condition ~"]" Instruction }
          ACTIVATION { if (Condition) { Instruction} }
          BEHAVIOR USES ( OUT D_unit [Side]; )
        }
    }
  ELSE
    {
      SYNTAX { "     " Instruction }
      ACTIVATION { Instruction }
      BEHAVIOR USES ( OUT D_unit [Side]; )
    }
}
OPERATION LD_baseR IN pipe.E1
{
  DECLARE
    {
      GROUP Condition    = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Side, Yside  = { Side1  ||  Side2 };
      GROUP Dest    = { Reg };
      GROUP BaseR   = { Addr_Register };
      GROUP OffsetR = { Addr_Offset };
      GROUP Mode = { Pos_Ofs_Reg || Neg_Ofs_Reg || Pre_Incr_Reg || Pre_Decr_Reg || Post_Incr_Reg || Post_Decr_Reg ||
             Pos_Ofs_Const || Neg_Ofs_Const || Pre_Incr_Const || Pre_Decr_Const || Post_Incr_Const || Post_Decr_Const };
      GROUP Instruction = { LDB_baseR  ||  LDH_baseR  ||  LDW_baseR  || LDBU_baseR  ||  LDHU_baseR };
      INSTANCE LD_bR, LD_E2, LD_E3, LD_E4, LD_E5;
    }
  CODING { Condition  z  Dest  BaseR  OffsetR  Mode 0bx  Yside  Instruction  01b  Side }
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" Instruction  ~"," Dest.Int  }
          ACTIVATION { if (!Condition) { Instruction, LD_E2, LD_E3, LD_E4, LD_E5 } }
          BEHAVIOR { if (!Condition) { LD_bR(); } }
        }
```

```
      ELSE
        {
          SYNTAX { " ["  ~Condition  ~"]" Instruction  ~","  Dest.Int  }
          ACTIVATION { if (Condition) { Instruction, LD_E2, LD_E3, LD_E4, LD_E5 } }
          BEHAVIOR { if (Condition) { LD_bR(); } }
        }
    }
  ELSE
    {
      SYNTAX { "     " Instruction  ~","  Dest.Int  }
      ACTIVATION { Instruction, LD_E2, LD_E3, LD_E4, LD_E5 }
      BEHAVIOR { LD_bR(); }
    }
}

OPERATION LD_15b IN pipe.E1
{
  DECLARE
    {
      GROUP Side = { Side1 || Side2 };
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP BReg   = { B14  ||  B15 };
      GROUP Ucst15 = { Addr_Offset15b };
      GROUP Dest   = { Reg };
      GROUP Instruction = { LDB_15b  ||  LDH_15b  ||  LDW_15b  || LDBU_15b  ||  LDHU_15b };
      INSTANCE LD_15, LD_E2, LD_E3, LD_E4, LD_E5;
    }
  CODING { Condition  z  Dest  Ucst15  BReg  Instruction  11b  Side }
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" Instruction  ~".D2" "*+"  ~BReg  ~Ucst15  ~","  Dest.Int }
          ACTIVATION { if (!Condition) { Instruction, LD_E2, LD_E3, LD_E4, LD_E5 } }
          BEHAVIOR { if (!Condition) { LD_15(); } }
        }
      ELSE
        {
          SYNTAX { " ["  ~Condition  ~"]" Instruction ~".D2" "*+"  ~BReg  ~Ucst15  ~","  Dest.Int  }
          ACTIVATION { if (Condition) { Instruction, LD_E2, LD_E3, LD_E4, LD_E5 } }
          BEHAVIOR { if (Condition) { LD_15(); } }
        }
    }
  ELSE
    {
      SYNTAX { "     " Instruction  ~".D2" "*+"  ~BReg  ~Ucst15  ~","  Dest.Int }
      ACTIVATION { Instruction, LD_E2, LD_E3, LD_E4, LD_E5 }
      BEHAVIOR { LD_15(); }
    }
}

OPERATION ST_baseR IN pipe.E1
{
  DECLARE
    {
      GROUP Condition    = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Side, Yside  = { Side1  ||  Side2 };
      GROUP Src     = { Reg };
      GROUP BaseR   = { Addr_Register };
      GROUP OffsetR = { Addr_Offset };
      GROUP Mode = { Pos_Ofs_Reg || Neg_Ofs_Reg || Pre_Incr_Reg || Pre_Decr_Reg || Post_Incr_Reg || Post_Decr_Reg ||
            Pos_Ofs_Const || Neg_Ofs_Const || Pre_Incr_Const || Pre_Decr_Const || Post_Incr_Const || Post_Decr_Const };
      GROUP Instruction = { STB_baseR  ||  STH_baseR  ||  STW_baseR };
    }
  CODING { Condition  z  Src  BaseR  OffsetR  Mode  0bx  Yside  Instruction  01b  Side }
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" Instruction }
          ACTIVATION { if (!Condition) { Instruction } }
        }
      ELSE
        {
          SYNTAX { " ["  ~Condition  ~"]" Instruction }
          ACTIVATION { if (Condition) { Instruction} }
        }
    }
  ELSE
    {
      SYNTAX { "     " Instruction }
      ACTIVATION { Instruction }
    }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT D_unit [Side]; )
}

OPERATION ST_15b IN pipe.E1
{
  DECLARE
    {
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Side   = { Side2 || Side1 };
      GROUP BReg   = { B14  ||  B15 };
      GROUP Ucst15 = { Addr_Offset15b };
      GROUP Src    = { Reg };
      GROUP Instruction = { STB_15b  ||  STH_15b  ||  STW_15b };
    }
  CODING { Condition  z  Src  Ucst15  BReg  Instruction  11b  Side }
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" Instruction  ~".D2"  Src.Int  ~","  "*"  ~BReg  ~Ucst15 }
          ACTIVATION { if (!Condition) { Instruction } }
        }
      ELSE
        {
          SYNTAX { " ["  ~Condition  ~"]" Instruction  ~".D2"  Src.Int  ~","  "*"  ~BReg  ~Ucst15 }
          ACTIVATION { if (Condition) { Instruction} }
        }
    }
  ELSE
    {
      SYNTAX { "     " Instruction  ~".D2"  Src.Int  ~","  "*"  ~BReg  ~Ucst15 }
      ACTIVATION { Instruction }
    }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES (IN  BReg; OUT D_unit [Side2]; OUT Dest.Int; )
}

OPERATION ADD_D_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 010000b }
```

```
  SYNTAX { "ADD"  ~".D"  ~Side  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = Src1.Int + Src2.Int;
    }
}

OPERATION ADD_D_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 0100010b }
  SYNTAX { "ADD"  ~".D"  ~Side  Src1.UCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Src1.UCst5 + Src2.Int;
    }
}

OPERATION ADDAB_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1100000b }
  SYNTAX { "ADDAB"  ~".D"  ~Side  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, Src1.Int, amr);
    }
}

OPERATION ADDAH_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1101000b }
  SYNTAX { "ADDAH"  ~".D"  ~Side  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, Src1.Int << 1, amr);
    }
}

OPERATION ADDAW_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1110000b }
  SYNTAX { "ADDAW"  ~".D"  ~Side  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, Src1.Int << 2, amr);
    }
}

OPERATION ADDAB_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1100010b }
  SYNTAX { "ADDAB"  ~".D"  ~Side  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, Src1.UCst5, amr);
    }
}

OPERATION ADDAH_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1101010b }
  SYNTAX { "ADDAH"  ~".D"  ~Side  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, Src1.UCst5 << 1, amr);
    }
}

OPERATION ADDAW_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1110010b }
  SYNTAX { "ADDAW"  ~".D"  ~Side  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, Src1.UCst5 << 2, amr);
    }
}

OPERATION LDB_baseR IN pipe.E1
{
  DECLARE { REFERENCE Mode, Side, Yside, BaseR, OffsetR; }
  CODING { 010b }
  SWITCH (Mode)
    {
      CASE Pos_Ofs_Reg: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*+"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.Reg, amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (addr, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
            }
        }
      CASE Neg_Ofs_Reg: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*-"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.Reg, amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (addr, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
            }
        }
      CASE Pre_Incr_Reg: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*++"  ~BaseR.Reg  ~OffsetR.Reg }

          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.Reg, amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (BaseR.Reg, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
            }
        }
      CASE Pre_Decr_Reg: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*--"  ~BaseR.Reg  ~OffsetR.Reg }

          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.Reg, amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (BaseR.Reg, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
            }
        }
      CASE Post_Incr_Reg: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.Reg }

          BEHAVIOR USES (OUT BaseR.Reg;)
```

```
            {
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (BaseR.Reg, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.Reg, amr);
            }
        }
      CASE Post_Decr_Reg: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.Reg }

          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (BaseR.Reg, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.Reg, amr);
            }
        }
      CASE Pos_Ofs_Const: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*+"  ~BaseR.Reg  ~OffsetR.UCst5 }

          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.UCst5, amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (addr, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
            }
        }
      CASE Neg_Ofs_Const: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*-"  ~BaseR.Reg  ~OffsetR.UCst5 }

          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.UCst5, amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (addr, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
            }
        }
      CASE Pre_Incr_Const: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*++"  ~BaseR.Reg  ~OffsetR.UCst5 }

          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.UCst5, amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (BaseR.Reg, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
            }
        }
      CASE Pre_Decr_Const: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*--"  ~BaseR.Reg  ~OffsetR.UCst5 }

          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.UCst5, amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (BaseR.Reg, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
            }
        }
      CASE Post_Incr_Const: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.UCst5 }

          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (BaseR.Reg, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.UCst5, amr);
            }
        }
      CASE Post_Decr_Const: {
          SYNTAX { "LDB"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.UCst5 }

          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              D_data[Side][1] = SIGN_EXT (DMC_Read_B (BaseR.Reg, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.UCst5, amr);
            }
        }
    }
}

OPERATION LDH_baseR IN pipe.E1
{
  DECLARE { REFERENCE Mode, Side, Yside, BaseR, OffsetR; }
  CODING { 100b }
  SWITCH (Mode)
    {
      CASE Pos_Ofs_Reg: {
          SYNTAX { "LDH"  ~".D"  ~Yside  "*+"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 1), amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_H (addr, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
            }
        }
      CASE Neg_Ofs_Reg: {
          SYNTAX { "LDH"  ~".D"  ~Yside  "*-"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 1), amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_H (addr, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
            }
        }
      CASE Pre_Incr_Reg: {
          SYNTAX { "LDH"  ~".D"  ~Yside  "*++"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 1), amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
            }
        }
      CASE Pre_Decr_Reg: {
          SYNTAX { "LDH"  ~".D"  ~Yside  "*--"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 1), amr);
              D_data[Side][1] = SIGN_EXT (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
            }
        }
      CASE Post_Incr_Reg: {
          SYNTAX { "LDH"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              D_data[Side][1] = SIGN_EXT (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 1), amr);
```

```
            }
        }
        CASE Post_Decr_Reg: {
            SYNTAX { "LDH"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                D_data[Side][1] = SIGN_EXT (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                        ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 1), amr);
              }
        }
        CASE Pos_Ofs_Const: {
            SYNTAX { "LDH"  ~".D"  ~Yside  "*+"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 1), amr);
                D_data[Side][1] = SIGN_EXT (DMC_Read_H (addr, Side, data_read_bus, memory_read,
                                                        ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
              }
        }
        CASE Neg_Ofs_Const: {
            SYNTAX { "LDH"  ~".D"  ~Yside  "*-"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 1), amr);
                D_data[Side][1] = SIGN_EXT (DMC_Read_H (addr, Side, data_read_bus, memory_read,
                                                        ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
              }
        }
        CASE Pre_Incr_Const: {
            SYNTAX { "LDH"  ~".D"  ~Yside  "*++"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 1), amr);
                D_data[Side][1] = SIGN_EXT (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                        ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
              }
        }
        CASE Pre_Decr_Const: {
            SYNTAX { "LDH"  ~".D"  ~Yside  "*--"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 1), amr);
                D_data[Side][1] = SIGN_EXT (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                        ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
              }
        }
        CASE Post_Incr_Const: {
            SYNTAX { "LDH"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                D_data[Side][1] = SIGN_EXT (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                        ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 1), amr);
              }
        }
        CASE Post_Decr_Const: {
            SYNTAX { "LDH"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                D_data[Side][1] = SIGN_EXT (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                        ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 1), amr);
              }
        }
    }
}

OPERATION LDW_baseR IN pipe.E1
{
  DECLARE { REFERENCE Mode, Side, Yside, BaseR, OffsetR; }
  CODING { 110b }
  SWITCH (Mode)
    {
      CASE Pos_Ofs_Reg: {
          SYNTAX { "LDW"  ~".D"  ~Yside  "*+"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 2), amr);
              D_data[Side][1] = DMC_Read_W (addr, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
            }
        }
      CASE Neg_Ofs_Reg: {
          SYNTAX { "LDW"  ~".D"  ~Yside  "*-"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 2), amr);
              D_data[Side][1] = DMC_Read_W (addr, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
            }
        }
      CASE Pre_Incr_Reg: {
          SYNTAX { "LDW"  ~".D"  ~Yside  "*++"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 2), amr);
              D_data[Side][1] = DMC_Read_W (BaseR.Reg, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
            }
        }
      CASE Pre_Decr_Reg: {
          SYNTAX { "LDW"  ~".D"  ~Yside  "*--"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 2), amr);
              D_data[Side][1] = DMC_Read_W (BaseR.Reg, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
            }
        }
      CASE Post_Incr_Reg: {
          SYNTAX { "LDW"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              D_data[Side][1] = DMC_Read_W (BaseR.Reg, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 2), amr);
            }
        }
      CASE Post_Decr_Reg: {
          SYNTAX { "LDW"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
```

```
            {
              D_data[Side][1] = DMC_Read_W (BaseR.Reg, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 2), amr);
            }
        }
        CASE Pos_Ofs_Const: {
590         SYNTAX { "LDW"  ~".D"  ~Yside  "*+"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 2), amr);
              D_data[Side][1] = DMC_Read_W (addr, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
            }
        }
        CASE Neg_Ofs_Const: {
600         SYNTAX { "LDW"  ~".D"  ~Yside  "*-"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5<< 2), amr);
              D_data[Side][1] = DMC_Read_W (addr, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
            }
        }
        CASE Pre_Incr_Const: {
610         SYNTAX { "LDW"  ~".D"  ~Yside  "*++"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 2), amr);
              D_data[Side][1] = DMC_Read_W (BaseR.Reg, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
            }
        }
        CASE Pre_Decr_Const: {
620         SYNTAX { "LDW"  ~".D"  ~Yside  "*--"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 2), amr);
              D_data[Side][1] = DMC_Read_W (BaseR.Reg, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
            }
        }
        CASE Post_Incr_Const: {
630         SYNTAX { "LDW"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
            {
              D_data[Side][1] = DMC_Read_W (BaseR.Reg, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 2), amr);
            }
        }
        CASE Post_Decr_Const: {
640         SYNTAX { "LDW"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
            {
              D_data[Side][1] = DMC_Read_W (BaseR.Reg, Side, data_read_bus, memory_read,
                                            ext_mem_ce0,int_prog_mem,int_periph,
                                            ext_data_mem,int_data_mem);
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 2), amr);
            }
        }
    }
650 }

    OPERATION LDBU_baseR IN pipe.E1
    {
      DECLARE { REFERENCE Mode, Side, Yside, BaseR, OffsetR; }
      CODING { 001b }
      SWITCH (Mode)
        {
          CASE Pos_Ofs_Reg: {
              SYNTAX { "LDBU"  ~".D"  ~Yside  "*+"  ~BaseR.Reg  ~OffsetR.Reg }
660           BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.Reg, amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (addr, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
              }
          }
          CASE Neg_Ofs_Reg: {
              SYNTAX { "LDBU"  ~".D"  ~Yside  "*-"  ~BaseR.Reg  ~OffsetR.Reg }
670           BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.Reg, amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (addr, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
              }
          }
          CASE Pre_Incr_Reg: {
              SYNTAX { "LDBU"  ~".D"  ~Yside  "*++"  ~BaseR.Reg  ~OffsetR.Reg }
680           BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.Reg, amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (BaseR.Reg, Side, data_read_bus,
                                                            memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
              }
          }
          CASE Pre_Decr_Reg: {
690           SYNTAX { "LDBU"  ~".D"  ~Yside  "*--"  ~BaseR.Reg  ~OffsetR.Reg }
              BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.Reg, amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (BaseR.Reg, Side, data_read_bus,
                                                            memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
              }
          }
          CASE Post_Incr_Reg: {
700           SYNTAX { "LDBU"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.Reg }
              BEHAVIOR USES (OUT BaseR.Reg;)
              {
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (BaseR.Reg, Side,
                                                            data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.Reg, amr);
```

```
            }
        }
        CASE Post_Decr_Reg: {
            SYNTAX { "LDBU"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (BaseR.Reg, Side,
                                                            data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.Reg, amr);
            }
        }
        CASE Pos_Ofs_Const: {
            SYNTAX { "LDBU"  ~".D"  ~Yside  "*+"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.UCst5, amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (addr, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
            }
        }
        CASE Neg_Ofs_Const: {
            SYNTAX { "LDBU"  ~".D"  ~Yside  "*-"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.UCst5, amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (addr, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
            }
        }
        CASE Pre_Incr_Const: {
            SYNTAX { "LDBU"  ~".D"  ~Yside  "*++"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.UCst5, amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (BaseR.Reg, Side,
                                                            data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
            }
        }
        CASE Pre_Decr_Const: {
            SYNTAX { "LDBU"  ~".D"  ~Yside  "*--"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.UCst5, amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (BaseR.Reg, Side,
                                                            data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
            }
        }
        CASE Post_Incr_Const: {
            SYNTAX { "LDBU"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (BaseR.Reg, Side,
                                                            data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.UCst5, amr);
            }
        }
        CASE Post_Decr_Const: {
            SYNTAX { "LDBU"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (BaseR.Reg, Side,
                                                            data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph,
                                                            ext_data_mem,int_data_mem), 8);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.UCst5, amr);
            }
        }
    }
}



OPERATION LDHU_baseR IN pipe.E1
{
  DECLARE { REFERENCE Mode, Side, Yside, BaseR, OffsetR; }
  CODING { 000b }
  SWITCH (Mode)
    {
        CASE Pos_Ofs_Reg: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*+"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 1), amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (addr, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
            }
        }
        CASE Neg_Ofs_Reg: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*-"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 1), amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (addr, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
            }
        }
        CASE Pre_Incr_Reg: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*++"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 1), amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
            }
        }
        CASE Pre_Decr_Reg: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*--"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 1), amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
            }
        }
        CASE Post_Incr_Reg: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.Reg }
```

```
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 1), amr);
              }
          }
        CASE Post_Decr_Reg: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 1), amr);
              }
          }
        CASE Pos_Ofs_Const: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*+"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 1), amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (addr, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
              }
          }
        CASE Neg_Ofs_Const: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*-"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 1), amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (addr, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
              }
          }
        CASE Pre_Incr_Const: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*++"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 1), amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
              }
          }
        CASE Pre_Decr_Const: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*--"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 1), amr);
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
              }
          }
        CASE Post_Incr_Const: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 1), amr);
              }
          }
        CASE Post_Decr_Const: {
            SYNTAX { "LDHU"  ~".D"  ~Yside  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (BaseR.Reg, Side, data_read_bus, memory_read,
                                                            ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 1), amr);
              }
          }
      }
    }

    OPERATION LDB_15b IN pipe.E1
    {
      DECLARE { REFERENCE BReg, Ucst15, Side; }
      CODING { 010b }
      SYNTAX { "LDB" }
      BEHAVIOR
        {
          D_data[Side][1] = SIGN_EXT (DMC_Read_B (BReg + Ucst15, Side, data_read_bus, memory_read,
                                                  ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
        }
    }

    OPERATION LDH_15b IN pipe.E1
    {
      DECLARE { REFERENCE BReg, Ucst15, Side; }
      CODING { 100b }
      SYNTAX { "LDH" }
      BEHAVIOR
        {
          D_data[Side][1] = SIGN_EXT (DMC_Read_H (BReg + (Ucst15 << 1), Side, data_read_bus, memory_read,
                                                  ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
        }
    }

    OPERATION LDW_15b IN pipe.E1
    {
      DECLARE { REFERENCE BReg, Ucst15, Side; }
      CODING { 110b }
      SYNTAX { "LDW" }
      BEHAVIOR
        {
          D_data[Side][1] = DMC_Read_W (BReg + (Ucst15 << 2), Side, data_read_bus, memory_read,
                                        ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
        }
    }

    OPERATION LDBU_15b IN pipe.E1
    {
      DECLARE { REFERENCE BReg, Ucst15, Side; }
      CODING { 001b }
      SYNTAX { "LDBU" }
      BEHAVIOR
        {
          D_data[Side][1] = EXTR_LO_BITS (DMC_Read_B (BReg + Ucst15, Side, data_read_bus, memory_read,
                                                      ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 8);
        }
    }

    OPERATION LDHU_15b IN pipe.E1
    {
      DECLARE { REFERENCE BReg, Ucst15, Side; }
      CODING { 000b }
      SYNTAX { "LDH" }
      BEHAVIOR
        {
          D_data[Side][1] = EXTR_LO_BITS (DMC_Read_H (BReg + (Ucst15 << 1), Side, data_read_bus, memory_read,
```

```
                                                    ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem), 16);
    }
}

ALIAS OPERATION MV_D IN pipe.E1
{
  DECLARE
    {
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Side  = { Side1  ||  Side2 };
      GROUP Dest, Src2 = { Reg };
    }
    CODING { Condition  z  Dest  Src2  0b0[5] 010010b  10000b  Side }
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" "MV"  ~".D"  ~Side  Src2.Int  ~","  Dest.Int }
          BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
            {
              if (!Condition)
                  Dest.Int = Src2.Int;
            }
        }
      ELSE
        {
          SYNTAX { " [" ~Condition ~"]" "MV"  ~".D"  ~Side  Src2.Int  ~","  Dest.Int }
          BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
            {
              if (Condition)
                  Dest.Int = Src2.Int;
            }
        }
    }
  ELSE
    {
      SYNTAX { "     " "MV"  ~".D"  ~Side  Src2.Int  ~","  Dest.Int }
      BEHAVIOR USES (IN Src2.Int; OUT Dest.Int; )
        {
          Dest.Int = Src2.Int;
        }
    }
}

OPERATION STB_baseR IN pipe.E1
{
  DECLARE { REFERENCE Mode, Side, Yside, BaseR, OffsetR, Src; }
  CODING { 011b }
  SWITCH (Mode)
    {
      CASE Pos_Ofs_Reg: {
          SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*+"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.Reg, amr);
              DMC_Write_B (addr, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
            }
        }
      CASE Neg_Ofs_Reg: {
          SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*-"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.Reg, amr);
              DMC_Write_B (addr, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
            }
        }
      CASE Pre_Incr_Reg: {
          SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*++"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.Reg, amr);
              DMC_Write_B (BaseR.Reg, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
            }
        }
      CASE Pre_Decr_Reg: {
          SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*--"  ~BaseR.Reg  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.Reg, amr);
              DMC_Write_B (BaseR.Reg, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
            }
        }
      CASE Post_Incr_Reg: {
          SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              DMC_Write_B (BaseR.Reg, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.Reg, amr);
            }
        }
      CASE Post_Decr_Reg: {
          SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.Reg }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              DMC_Write_B (BaseR.Reg, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.Reg, amr);
            }
        }
      CASE Pos_Ofs_Const: {
          SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*+"  ~BaseR.Reg  ~OffsetR.UCst5 }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.UCst5, amr);
              DMC_Write_B (addr, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
            }
        }
      CASE Neg_Ofs_Const: {
          SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*-"  ~BaseR.Reg  ~OffsetR.UCst5 }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.UCst5, amr);
              DMC_Write_B (addr, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
            }
        }
      CASE Pre_Incr_Const: {
          SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*++"  ~BaseR.Reg  ~OffsetR.UCst5 }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.UCst5, amr);
              DMC_Write_B (BaseR.Reg, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
            }
        }
      CASE Pre_Decr_Const: {
          SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*--"  ~BaseR.Reg  ~OffsetR.UCst5 }
          BEHAVIOR USES (OUT BaseR.Reg;)
            {
              BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.UCst5, amr);
              DMC_Write_B (BaseR.Reg, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
```

```
            }
        }
        CASE Post_Incr_Const: {
            SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                DMC_Write_B (BaseR.Reg, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, OffsetR.UCst5, amr);
              }
        }
        CASE Post_Decr_Const: {
            SYNTAX { "STB"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                DMC_Write_B (BaseR.Reg, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -OffsetR.UCst5, amr);
              }
        }
    }
}

OPERATION STH_baseR IN pipe.E1
{
  DECLARE { REFERENCE Side, Yside, BaseR, OffsetR, Src, Mode; }
  CODING { 101b }
  SWITCH (Mode)
    {
        CASE Pos_Ofs_Reg: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*+"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 1), amr);
                DMC_Write_H (addr, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
        }
        CASE Neg_Ofs_Reg: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*-"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 1), amr);
                DMC_Write_H (addr, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
        }
        CASE Pre_Incr_Reg: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*++"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 1), amr);
                DMC_Write_H (BaseR.Reg, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
        }
        CASE Pre_Decr_Reg: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*--"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 1), amr);
                DMC_Write_H (BaseR.Reg, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
        }
        CASE Post_Incr_Reg: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                DMC_Write_H (BaseR.Reg, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 1), amr);
              }
        }
        CASE Post_Decr_Reg: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                DMC_Write_H (BaseR.Reg, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 1), amr);
              }
        }
        CASE Pos_Ofs_Const: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*+"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 1), amr);
                DMC_Write_H (addr, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
        }
        CASE Neg_Ofs_Const: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*-"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 1), amr);
                DMC_Write_H (addr, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
        }
        CASE Pre_Incr_Const: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*++"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 1), amr);
                DMC_Write_H (BaseR.Reg, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
        }
        CASE Pre_Decr_Const: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*--"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 1), amr);
                DMC_Write_H (BaseR.Reg, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
        }
        CASE Post_Incr_Const: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                DMC_Write_H (BaseR.Reg, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 1), amr);
              }
        }
        CASE Post_Decr_Const: {
            SYNTAX { "STH"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                DMC_Write_H (BaseR.Reg, Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 1), amr);
              }
        }
    }
}

OPERATION STW_baseR IN pipe.E1
{
  DECLARE { REFERENCE Mode, Side, Yside, BaseR, OffsetR, Src; }
  CODING { 111b }
```

```
    SWITCH (Mode)
      {
        CASE Pos_Ofs_Reg: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*+"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 2), amr);
                DMC_Write_W (addr, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
          }
        CASE Neg_Ofs_Reg: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*-"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 2), amr);
                DMC_Write_W (addr, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
          }
        CASE Pre_Incr_Reg: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*++"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 2), amr);
                DMC_Write_W (BaseR.Reg, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
          }
        CASE Pre_Decr_Reg: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*--"  ~BaseR.Reg  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 2), amr);
                DMC_Write_W (BaseR.Reg, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
          }
        CASE Post_Incr_Reg: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                DMC_Write_W (BaseR.Reg, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.Reg << 2), amr);
              }
          }
        CASE Post_Decr_Reg: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.Reg }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                DMC_Write_W (BaseR.Reg, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.Reg << 2), amr);
              }
          }
        CASE Pos_Ofs_Const: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*+"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 2), amr);
                DMC_Write_W (addr, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
          }
        CASE Neg_Ofs_Const: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*-"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                register U32 addr = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 2), amr);
                DMC_Write_W (addr, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
          }
        CASE Pre_Incr_Const: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*++"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 2), amr);
                DMC_Write_W (BaseR.Reg, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
          }
        CASE Pre_Decr_Const: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*--"  ~BaseR.Reg  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 2), amr);
                DMC_Write_W (BaseR.Reg, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
              }
          }
        CASE Post_Incr_Const: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"++"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                DMC_Write_W (BaseR.Reg, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, (OffsetR.UCst5 << 2), amr);
              }
          }
        CASE Post_Decr_Const: {
            SYNTAX { "STW"  ~".D"  ~Yside  Src.Int  ~","  "*"  ~BaseR.Reg  ~"--"  ~OffsetR.UCst5 }
            BEHAVIOR USES (OUT BaseR.Reg;)
              {
                DMC_Write_W (BaseR.Reg, Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
                BaseR.Reg = Circ_Addr (Side, BaseR.Reg, BaseR.Idx, -(OffsetR.UCst5 << 2), amr);
              }
          }
      }
    }

    OPERATION STB_15b IN pipe.E1
    {
      DECLARE { REFERENCE BReg, Ucst15, Src, Side; }
      CODING { 011b }
      SYNTAX { "STB" }
      BEHAVIOR
        {
          DMC_Write_B (BReg + Ucst15, Side, LO8 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
        }
    }

    OPERATION STH_15b IN pipe.E1
    {
      DECLARE { REFERENCE BReg, Ucst15, Src, Side; }
      CODING { 101b }
      SYNTAX { "STH" }
      BEHAVIOR
        {
          DMC_Write_H (BReg + (Ucst15 << 1), Side, LO16 (Src.Int), ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
        }
    }

    OPERATION STW_15b IN pipe.E1
    {
      DECLARE { REFERENCE BReg, Ucst15, Src, Side; }
      CODING { 111b }
      SYNTAX { "STW" }
      BEHAVIOR
        {
```

```
        DMC_Write_W (BReg + (Ucst15 << 2), Side, Src.Int, ext_mem_ce0,int_prog_mem,int_periph, ext_data_mem,int_data_mem);
      }
}

OPERATION SUB_D_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 010001b }
  SYNTAX { "SUB"  ~".D"  ~Side  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Src2.Int - Src1.Int;
    }
}

OPERATION SUB_D_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 010011b }
  SYNTAX { "SUB"  ~".D"  ~Side  Src1.UCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Src2.Int - Src1.UCst5;
    }
}

OPERATION SUBAB_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 110001b }
  SYNTAX { "SUBAB"  ~".D"  ~Side  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, -Src1.Int, amr);
    }
}

OPERATION SUBAH_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 110101b }
  SYNTAX { "SUBAH"  ~".D"  ~Side  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, -Src1.Int << 1, amr);
    }
}

OPERATION SUBAW_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 111001b }
  SYNTAX { "SUBAW"  ~".D"  ~Side  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, -Src1.Int << 2, amr);
    }
}

OPERATION SUBAB_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 110011b }
  SYNTAX { "SUBAB"  ~".D"  ~Side  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, -Src1.UCst5, amr);
    }
}

OPERATION SUBAH_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 110111b }
  SYNTAX { "SUBAH"  ~".D"  ~Side  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, -Src1.UCst5 << 1, amr);
    }
}

OPERATION SUBAW_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 111011b }
  SYNTAX { "SUBAW"  ~".D"  ~Side  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Circ_Addr (Side, Src2.Int, Src2.UCst5, -Src1.UCst5 << 2, amr);
    }
}

ALIAS OPERATION ZERO_D IN pipe.E1
{
  DECLARE
    {
      GROUP Side = { Side1  ||  Side2 };
      GROUP Dest = { Reg };
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
  }
  CODING { Condition  z  Dest  00000b  00000b  010001b  10000b  Side }

  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" "ZERO"  ~".D"  ~Side  Dest.Int  }
          BEHAVIOR USES ( OUT Dest.Int; )
            {
              if (!Condition)
                  Dest.Int = Dest.Int - Dest.Int;
            }
        }
      ELSE
        {
          SYNTAX { " [" ~Condition ~"]" "ZERO"  ~".D"  ~Side  Dest.Int  }
          BEHAVIOR USES ( OUT Dest.Int; )
            {
              if (Condition)
                  Dest.Int = Dest.Int - Dest.Int;
            }
        }
    }
  ELSE
    {
      SYNTAX { "     " "ZERO"  ~".D"  ~Side  Dest.Int  }
      BEHAVIOR USES ( OUT Dest.Int; )
        {
          Dest.Int = Dest.Int - Dest.Int;
        }
    }
}
```

```
OPERATION LD_bR
{
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT D_unit [Side]; ) { ld_count[0]++; }
}
OPERATION LD_15
{
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( IN BReg; OUT D_unit [Side2]; OUT Dest.Int; )
    {
      ld_count[0]++;
    }
}
```

l_unit.lisa

```
#include <c6201.h>
OPERATION Insn_L_unit IN pipe.E1
{
  DECLARE
    {
      GROUP Side  = { Side1  ||  Side2 };
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Instruction_group = { Insn_L_unit_group1  ||  Insn_L_unit_group2 || Insn_L_unit_group3 ||
        MV_L || NEG_L || NOT_L || ZERO_L };
    }
  CODING { Condition  z  Instruction_group  110b  Side }
  IF(Condition == Is_Conditional) THEN
    {
      IF(z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" Instruction_group }
          ACTIVATION { if (!Condition) { Instruction_group } }
        }
      ELSE
        {
          SYNTAX { " [" ~Condition ~"]" Instruction_group }
          ACTIVATION { if (Condition) Instruction_group } }
        }
    }
  ELSE
    {
      SYNTAX { "     " Instruction_group }
      ACTIVATION { Instruction_group }
    }
}
OPERATION Insn_L_unit_group1 IN pipe.E1
{
  DECLARE
    {
      GROUP Dest, Src1 = { Reg };
      GROUP Src2  = { XReg };
      GROUP Xpath = { No_Xpath  ||  Is_Xpath };
      GROUP Instruction = { ABS_int || ABS_long || ADD_L_iii ||  ADD_L_iil || ADDU_L_iil || ADD_L_cii || ADD_L_cll  ||
        AND_L_iii || AND_L_cii || CMPEQ_ii || CMPEQ_ci || CMPGT_ii || CMPGT_ci || CMPGTU_ii || CMPGTU_ci || CMPLT_ii ||
        CMPLT_ci || CMPLTU_ii || CMPLTU_ci || LMBD_reg || LMBD_cst || NORM_int || NORM_long || OR_L_iii || OR_L_cii  ||
        SADD_L_iii || SADD_L_cii || SADD_L_cll || SAT || SSUB_L_iii || SSUB_L_cii || SSUB_L_cll || SUB_L_iii ||
        SUB_L_iil || SUBU_L_iil  || SUB_L_cii || SUB_L_cll  || SUBC  || XOR_L_iii || XOR_L_cii };
      REFERENCE Side;
    }
  CODING { Dest  Src2  Src1  Xpath  Instruction }
  SYNTAX { Instruction }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT L_unit [Side]; )
  IF ((Side == Side1) && (Xpath == Is_Xpath)) THEN
    {
      BEHAVIOR USES (IN X_path1; )
    }
  IF ((Side == Side2) && (Xpath == Is_Xpath)) THEN
    {
      BEHAVIOR USES (IN X_path2; )
    }
  ACTIVATION { Instruction }
}
OPERATION Insn_L_unit_group2 IN pipe.E1
{
  DECLARE
    {
      GROUP Dest, Src2 = { Reg };
      GROUP Src1  = { XReg };
      GROUP Xpath = { No_Xpath || Is_Xpath };
      GROUP Instruction = {
        ADD_L_ill || ADDU_L_ill || CMPEQ_il || CMPGT_il || CMPGTU_il || CMPLT_il || CMPLTU_il || SADD_ill || SSUBrev ||
        SUB_L_iii2 || SUB_L_iil2 || SUBU_L_iil2
      };
      REFERENCE Side;
    }
  CODING { Dest  Src2  Src1  Xpath  Instruction }
  SYNTAX { Instruction }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT L_unit [Side]; )
  IF ((Side == Side1) && (Xpath == Is_Xpath)) THEN
    {
      BEHAVIOR USES (IN X_path1; )
    }
  IF ((Side == Side2) && (Xpath == Is_Xpath)) THEN
    {
      BEHAVIOR USES (IN X_path2; )
    }
  ACTIVATION { Instruction }
}
OPERATION Insn_L_unit_group3 IN pipe.E1
{
  DECLARE
    {
      GROUP Dest, Src1, Src2 = { Reg };
      GROUP Instruction = { CMPEQ_cl || CMPGT_cl || CMPGTU_cl || CMPLT_cl || CMPLTU_cl };
      REFERENCE Side;
    }
  CODING { Dest  Src2  Src1  0bx  Instruction }
  SYNTAX { Instruction }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT L_unit [Side]; )
  ACTIVATION { Instruction }
}
OPERATION ABS_int IN pipe.E1
```

```
{
  DECLARE { REFERENCE Side, Xpath, Src2, Dest; }
  CODING { 0011010b }
  SYNTAX { "ABS"  ~".L"  ~Side  ~Xpath  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int; )
    {
      if ((S32) Src2.Int >= 0)
          Dest.Int = Src2.Int;
      else
          Dest.Int = (Src2.Int != 0x80000000) ? -Src2.Int : 0x7FFFFFFF;
    }
}

OPERATION ABS_long IN pipe.E1
{
  DECLARE { REFERENCE Side, Src2, Dest; }
  CODING { 0111000b }
  SYNTAX { "ABS"  ~".L"  ~Side  Src2.Long  ~","  Dest.Long }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long;)
    {
      if ((Src2.Long & 0x80) == 0)             //  positive values
        {
          Dest.Int = Src2.Int;
          Dest.Long = Src2.Long;
        }
      else                                      //  negative values
        {
          if (   (Src2.Int == 0) && (Src2.Long == 0x80))
            {
              Dest.Int = 0xFFFFFFFF;
              Dest.Long = 0x7F;
            }
          else                                  //  smaller negative values
            {
              register U32 lsb = LO8 (~Src2.Int) + 1;
              register U32 msb = (  (~Src2.Long << 24) + ((- (S32) Src2.Int) >> 8));
              Dest.Int = ((msb << 8) | LO8 (lsb));
              Dest.Long = LO8 (msb >> 24);
            }
        }
    }
}

OPERATION ADD_L_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0000011b }
  SYNTAX { "ADD"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES(IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = Src1.Int + Src2.Int;
    }
}

OPERATION ADD_L_iil IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0100011b }
  SYNTAX { "ADD"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Long }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; OUT Dest.Long; )
    {
     add_int_int_long (Src1.Int, Src2.Int, &Dest.Int, &Dest.Long);
    }
}

OPERATION ADDU_L_iil IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0101011b }
  SYNTAX { "ADDU"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Long }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; OUT Dest.Long; )
    {
     addu_int_int_long (Src1.Int, Src2.Int, &Dest.Int, &Dest.Long);
    }
}

OPERATION ADD_L_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0000010b }
  SYNTAX { "ADD"  ~".L"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES ( IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = SIGN_EXT (Src1.SCst5, 5) + Src2.Int;
    }
}

OPERATION ADD_L_cll IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 0100000b }
  SYNTAX { "ADD"  ~".L"  ~Side  Src1.SCst5  ~","  Src2.Long  ~","  Dest.Long }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long;)
    {
      add_int_long_long (SIGN_EXT (Src1.SCst5, 5), Src2.Int, Src2.Long, &Dest.Int, &Dest.Long);
    }
}

OPERATION ADD_L_ill IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0100001b }
  SYNTAX { "ADD"  ~".L"  ~Side  ~Xpath  Src2.Long  ~","  Src1.Int  ~","  Dest.Long }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long; )
    {
      add_int_long_long (Src1.Int, Src2.Int, Src2.Long, &Dest.Int, &Dest.Long);
    }
}

OPERATION ADDU_L_ill IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0101001b }
  SYNTAX { "ADD"  ~".L"  ~Side  ~Xpath  Src2.Long  ~","  Src1.Int  ~","  Dest.Long }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long; )
    {
      addu_int_long_long (Src1.Int, Src2.Int, Src2.Long, &Dest.Int, &Dest.Long);
    }
}

OPERATION AND_L_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1111011b }
  SYNTAX { "AND"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
 BEHAVIOR USES ( IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = (Src1.Int & Src2.Int);
    }
}

OPERATION AND_L_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
```

```
  CODING { 1111010b }
  SYNTAX { "AND"  ~".L"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES ( IN Src2.Int; OUT Dest.Int; )
   {
    Dest.Int = (SIGN_EXT (Src1.SCst5, 5) & Src2.Int);
   }
}

OPERATION CMPEQ_ii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1010011b }
  SYNTAX { "CMPEQ"  ".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = ((U32) Src1.Int == (U32) Src2.Int);
    }
}

OPERATION CMPEQ_ci IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1010010b }
  SYNTAX { "CMPEQ"  ".L"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = ((U32) SIGN_EXT (Src1.SCst5, 5) == (U32) Src2.Int);
    }
}

OPERATION CMPEQ_il IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1010001b }
  SYNTAX { "CMPEQ"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Long  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; IN Src2.Long; OUT Dest.Int; )
    {
      if (IS_NEG32 (Src1.Int))
        Dest.Int = (   (      0xFF      == (U32) Src2.Long) && ((U32) Src1.Int == (U32) Src2.Int));
      else
        Dest.Int = (   (      0         == (U32) Src2.Long) && ((U32) Src1.Int == (U32) Src2.Int));
    }
}

OPERATION CMPEQ_cl IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1010000b }
  SYNTAX { "CMPEQ"  ".L"  ~Side  Src1.SCst5  ~","  Src2.Long  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int;)
    {
      if (IS_NEG32 (SIGN_EXT (Src1.SCst5, 5)))
        Dest.Int = (   (      0xFF      == (U32) Src2.Long) && ((U32) SIGN_EXT (Src1.SCst5, 5) == (U32) Src2.Int));
      else
        Dest.Int = (   (      0         == (U32) Src2.Long) && ((U32) Src1.SCst5 == (U32) Src2.Int));
    }
}

OPERATION CMPGT_ii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1000111b }
  SYNTAX { "CMPGT"  ".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = ((S32) Src1.Int > (S32) Src2.Int);
    }
}

OPERATION CMPGT_ci IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1000110b }
  SYNTAX { "CMPGT"  ".L"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = ((S32) SIGN_EXT (Src1.SCst5, 5) > (S32) Src2.Int);
    }
}

OPERATION CMPGT_il IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1000101b }
  SYNTAX { "CMPGT"  ".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Long  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; IN Src2.Long; OUT Dest.Int;)
    {
      if (IS_NEG32 (Src1.Int))
        {
          Dest.Int = (-1  >  (S32) SIGN_EXT (Src2.Long, 8));
          if (! Dest.Int)
              Dest.Int = (   (      0xFF      ==       Src2.Long) && ((U32) Src1.Int >  (U32) Src2.Int));
        }
      else
        {
          Dest.Int = (0  >  (S32) SIGN_EXT (Src2.Long, 8));
          if (! Dest.Int)
              Dest.Int = (   (      0x00     ==       Src2.Long) && ((U32) Src1.Int >  (U32) Src2.Int));
        }
    }
}

OPERATION CMPGT_cl IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1000100b }
  SYNTAX { "CMPGT"  ".L"  ~Side  Src1.SCst5  ~","  Src2.Long  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int;)
    {
      register int src1 = SIGN_EXT (Src1.SCst5, 5);
      if (IS_NEG32 (src1))
        {
          Dest.Int = (-1  >  (S32) SIGN_EXT (Src2.Long, 8));
          if (! Dest.Int)
              Dest.Int = (   (      0xFF  ==       Src2.Long) && ((U32) src1  >  (U32) Src2.Int));
        }
      else
        {
          Dest.Int = (0  >  (S32) SIGN_EXT (Src2.Long, 8));
          if (! Dest.Int)
              Dest.Int = (   (      0x00  ==       Src2.Long) && ((U32) src1  >  (U32) Src2.Int));
        }
    }
}

OPERATION CMPGTU_ii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1001111b }
  SYNTAX { "CMPGTU"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = ((U32) Src1.Int > (U32) Src2.Int);
```

```
    }
}

OPERATION CMPGTU_ci IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1001110b }
  SYNTAX { "CMPGTU" "~".L"  ~Side  ~Xpath  Src1.UCst4  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = ((U32) Src1.UCst4 > (U32) Src2.Int);
    }
}

OPERATION CMPGTU_il IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1001101b }
  SYNTAX { "CMPGTU" "~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Long  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; IN Src2.Long; OUT Dest.Int;)
    {
      Dest.Int = (   (      0x00      ==       (Src2.Long & 0xFF)) && ((U32) Src1.Int >  (U32) Src2.Int));
    }
}

OPERATION CMPGTU_cl IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1001100b }
  SYNTAX { "CMPGTU" "~".L"  ~Side  Src1.UCst4  ~","  Src2.Long  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int;)
    {
      Dest.Int = (   (      0x00       ==       Src2.Long) && ((U32) Src1.UCst4 >  (U32) Src2.Int));
    }
}

OPERATION CMPLT_ii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1010111b }
  SYNTAX { "CMPLT"  ".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = ((S32) Src1.Int < (S32) Src2.Int);
    }
}

OPERATION CMPLT_ci IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1010110b }
  SYNTAX { "CMPLT"  ".L"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = ((S32) SIGN_EXT (Src1.SCst5, 5) < (S32) Src2.Int);
    }
}

OPERATION CMPLT_il IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1010101b }
  SYNTAX { "CMPLT"  ".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Long  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; IN Src2.Long; OUT Dest.Int;)
    {
      if (IS_NEG32 (Src1.Int))
        {
          Dest.Int = (-1  <  (S32) SIGN_EXT (Src2.Long, 8));
          if (! Dest.Int)
              Dest.Int = (   (      0xFF      ==       Src2.Long) && ((U32) Src1.Int <  (U32) Src2.Int));
        }
      else
        {
          Dest.Int = (0  <  (S32) SIGN_EXT (Src2.Long, 8));
          if (! Dest.Int)
              Dest.Int = (   (      0x00      ==       Src2.Long) && ((U32) Src1.Int <  (U32) Src2.Int));
        }
    }
}

OPERATION CMPLT_cl IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1010100b }
  SYNTAX { "CMPLT"  ".L"  ~Side  Src1.SCst5  ~","  Src2.Long  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int;)
    {
      register int src1 = SIGN_EXT (Src1.SCst5, 5);
      if (IS_NEG32 (src1))
        {
          Dest.Int = (-1  <  (S32) SIGN_EXT (Src2.Long, 8));
          if (! Dest.Int)
              Dest.Int = (   (      0xFF  ==       Src2.Long) && ((U32) src1  <  (U32) Src2.Int));
        }
      else
        {
          Dest.Int = (0  <  (S32) SIGN_EXT (Src2.Long, 8));
          if (! Dest.Int)
              Dest.Int = (   (      0x00  ==       Src2.Long) && ((U32) src1  <  (U32) Src2.Int));
        }
    }
}

OPERATION CMPLTU_ii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1011111b }
  SYNTAX { "CMPLTU" "~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = ((U32) Src1.Int < (U32) Src2.Int);
    }
}

OPERATION CMPLTU_ci IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1011110b }
  SYNTAX { "CMPLTU" "~".L"  ~Side  ~Xpath  Src1.UCst4  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = ((U32) Src1.UCst4 < (U32) Src2.Int);
    }
}

OPERATION CMPLTU_il IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1011101b }
  SYNTAX { "CMPLTU" "~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Long  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; IN Src2.Long; OUT Dest.Int;)
    {
      Dest.Int = (   (      0x00      ==       Src2.Long) && ((U32) Src1.Int <  (U32) Src2.Int));
```

```
    }
}

OPERATION CMPLTU_cl IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1011100b }
  SYNTAX { "CMPLTU"  ~".L"  ~Side  Src1.UCst4  ~","  Src2.Long  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int;)
    {
      Dest.Int = (   (      0x00       ==       Src2.Long) && ((U32) Src1.UCst4 <  (U32) Src2.Int));
    }
}

OPERATION LMBD_reg IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1101011b }
  SYNTAX { "LMBD"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES ( IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
   {
     register int pos = 0;
     register U32 mask = 0x80000000;
     if (Src1.Int & 1)
       /*  search for a 1  */
       {
         while (((Src2.Int & mask) != mask) && (pos < 32))
           {
             mask = mask >> 1;
             pos++;
           }
       }
     else
       /*  search for a 0  */
       {
         while (((Src2.Int & mask) != 0) && (pos < 32))
           {
             mask = mask >> 1;
             pos++;
           }
       }
     Dest.Int = pos;
   }
}

OPERATION LMBD_cst IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1101010b }
  SYNTAX { "LMBD"  ~".L"  ~Side  ~Xpath  Src1.UCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES ( IN Src2.Int; OUT Dest.Int; )
   {
     register int pos = 0;
     register U32 mask = 0x80000000;
     if (Src1.UCst5 & 1)
       /*  search for a 1  */
       {
         while (((Src2.Int & mask) != mask) && (pos < 32))
           {
             mask = mask >> 1;
             pos++;
          }
       }
     else
       /*  search for a 0  */
       {
         while (((Src2.Int & mask) != 0) && (pos < 32))
           {
             mask = mask >> 1;
             pos++;
           }
       }
     Dest.Int = pos;
   }
}

ALIAS OPERATION MV_L IN pipe.E1
{
  DECLARE
    {
      GROUP Dest = { Reg };
      GROUP Src2 = { XReg };
      GROUP Xpath = { No_Xpath || Is_Xpath };
      REFERENCE Side;
    }
  CODING { Dest  Src2  0b0[5]  Xpath  0000010b }
  SYNTAX { "MV"  ~".L"  ~Side  ~Xpath  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Src2.Int;
    }
}

ALIAS OPERATION NEG_L IN pipe.E1
{
  DECLARE
    {
      GROUP Dest = { Reg };
      GROUP Src2 = { XReg };
      GROUP Xpath = { No_Xpath || Is_Xpath };
      REFERENCE Side;
    }
  CODING { Dest  Src2  0b0[5]  Xpath  0000110b }
  SYNTAX {  "NEG"  ~".L"  ~Side  Src2.Int  ~","  Dest.Int  }
  BEHAVIOR USES ( INOUT Dest.Int; )
    {
      Dest.Int = -Src2.Int;
    }
}

OPERATION NORM_int IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1100011b }
  SYNTAX { "NORM"  ~".L"  ~Side  ~Xpath  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int; )
    {
      register int pos = 0;
      register int bit_found = 0;
      register U32 mask = 0x80000000;
      while ((bit_found == 0) && (pos < 32))
        {
          if (Src2.Int >= 0)
            {
              if ((Src2.Int & mask) == mask)
                bit_found = pos;
            }
          else
            {
              if ((Src2.Int & mask) == 0)
                bit_found = pos;
```

```
            }
          pos++;
          mask = mask >> 1;
        }
      if (bit_found != 0)
        Dest.Int = bit_found - 1;
      else
        Dest.Int = 31;
    }
}

OPERATION NORM_long IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 1100000b }
  SYNTAX { "NORM"  ~".L"  ~Side  Src2.Long  ~","  Dest.Long }
  BEHAVIOR USES ( IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long;)
    {
      register int pos = 0;
      register int bit_found = 0;
      register U32 mask = 0x80000000;
      register U32 mask_msb = 0x80;
      register int sign = Src2.Long & mask_msb;
      while ((bit_found == 0) && (pos < 41))
        {
          if (pos < 8)
            {
              if (sign != 0)
                {
                  if ((Src2.Long & mask_msb) == mask_msb)
                    bit_found = pos;
                }
              else
                {
                  if ((Src2.Long & mask_msb) == 0)
                    bit_found = pos;
                }
              mask_msb = mask_msb >> 1;
            }
          else
            {
              if (sign != 0)
                {
                  if ((Src2.Int & mask) == mask)
                    bit_found = pos;
                }
              else
                {
                  if ((Src2.Int & mask) == 0)
                    bit_found = pos;
                }
              mask = mask >> 1;
            }
          pos++;
        }
      if (bit_found != 0)
        Dest.Int = bit_found - 1;
      else
        Dest.Int = 40;
    }
}

ALIAS OPERATION NOT_L IN pipe.E1
{
  DECLARE
    {
      GROUP Dest = { Reg };
      GROUP Src1 = { Minus_one };
      GROUP Src2 = { XReg };
      GROUP Xpath = { No_Xpath || Is_Xpath };
      REFERENCE Side;
    }
  CODING { Dest  Src2  Src1  Xpath  1101110b }
  SYNTAX { "NOT"  ~".L"  ~Side  ~Xpath  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = (SIGN_EXT (Src1, 5) ^ Src2.Int);
    }
}

OPERATION OR_L_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1111111b }
  SYNTAX { "OR"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES ( IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = (Src1.Int | Src2.Int);
    }
}

OPERATION OR_L_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1111110b }
  SYNTAX { "OR"  ~".L"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES ( IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = (SIGN_EXT (Src1.SCst5, 5) | Src2.Int);
    }
}

OPERATION SADD_L_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest;  INSTANCE SAT_E2; }
  CODING { 0010011b }
  SYNTAX { "SADD"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; OUT Dest.Long; )
    {
      add_int_int_long (Src1.Int, Src2.Int, &Dest.Int, &Dest.Long);
      saturate (Dest.Int, Dest.Long, &Dest.Int, &sat_bit);
    }
  ACTIVATION { SAT_E2 }
}

OPERATION SADD_L_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest;  INSTANCE SAT_E2; }
  CODING { 0010010b }
  SYNTAX { "SADD"  ~".L"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int; OUT Dest.Long;)
    {
      add_int_int_long (SIGN_EXT (Src1.SCst5, 5), Src2.Int, &Dest.Int, &Dest.Long);
      saturate (Dest.Int, Dest.Long, &Dest.Int, &sat_bit);
    }
  ACTIVATION { SAT_E2 }
}

OPERATION SADD_L_cll IN pipe.E1
{
```

```
  DECLARE { REFERENCE Side, Src1, Src2, Dest; INSTANCE SAT_E2; }

  CODING { 0110000b }
  SYNTAX { "SADD"  ~".L"  ~Side  Src1.SCst5  ~","  Src2.Long  ~","  Dest.Long }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long;)
    {
      add_int_long_long (SIGN_EXT (Src1.SCst5, 5), Src2.Int, Src2.Long, &Dest.Int, &Dest.Long);
      saturate_long (Dest.Int, Dest.Long, &Dest.Int, &Dest.Long, &sat_bit);
    }
  ACTIVATION { SAT_E2 }
}

OPERATION SADD_ill IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest;  INSTANCE SAT_E2; }
  CODING { 0110001b }
  SYNTAX { "SADD"  ~".L"  ~Side  ~Xpath  Src2.Long  ~","  Src1.Int  ~","  Dest.Long }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long; )
   {
      add_int_long_long (Src1.Int, Src2.Int, Src2.Long, &Dest.Int, &Dest.Long);
      saturate_long (Dest.Int, Dest.Long, &Dest.Int, &Dest.Long, &sat_bit);
    }
  ACTIVATION { SAT_E2 }
}

OPERATION SAT IN pipe.E1
{
  DECLARE { REFERENCE Side, Src2, Dest;  INSTANCE SAT_E2; }
  CODING { 1000000b }
  SYNTAX { "SAT"  ~".L"  ~Side  Src2.Long  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int; )
    {
      saturate (Src2.Int, Src2.Long, &Dest.Int, &sat_bit);
    }
  ACTIVATION { SAT_E2 }
}

OPERATION SSUB_L_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest;  INSTANCE SAT_E2; }
  CODING { 0001111b }
  SYNTAX { "SSUB"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; OUT Dest.Long; )
    {
      add_int_int_long (Src1.Int, -Src2.Int, &Dest.Int, &Dest.Long);
      saturate (Dest.Int, Dest.Long, &Dest.Int, &sat_bit);
    }
  ACTIVATION { SAT_E2 }
}

OPERATION SSUB_L_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest;  INSTANCE SAT_E2; }
  CODING { 0001110b }
  SYNTAX { "SSUB"  ~".L"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int; OUT Dest.Long;)
    {
      add_int_int_long (SIGN_EXT (Src1.SCst5, 5), -Src2.Int, &Dest.Int, &Dest.Long);
      saturate (Dest.Int, Dest.Long, &Dest.Int, &sat_bit);
    }
  ACTIVATION { SAT_E2 }
}

OPERATION SSUB_L_cll IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; INSTANCE SAT_E2; }
  CODING { 0101100b }
  SYNTAX { "SSUB"  ~".L"  ~Side  Src1.SCst5  ~","  Src2.Long  ~","  Dest.Long }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long;)
    {
      register U32 src2_l = Src2.Int;
      register U32 src2_h = Src2.Long;
      add_int_long_long (1, ~src2_l, ~src2_h, &src2_l, &src2_h);
      add_int_long_long (SIGN_EXT (Src1.SCst5, 5), src2_l, src2_h, &Dest.Int, &Dest.Long);
      saturate_long (Dest.Int, Dest.Long, &Dest.Int, &Dest.Long, &sat_bit);
    }
  ACTIVATION { SAT_E2 }
}

OPERATION SSUBrev IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest;  INSTANCE SAT_E2; }
  CODING { 0011111b }
  SYNTAX { "SSUB"  ~".L"  ~Side  ~Xpath  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; OUT Dest.Long; )
    {
      add_int_int_long (Src1.Int, -Src2.Int, &Dest.Int, &Dest.Long);
      saturate (Dest.Int, Dest.Long, &Dest.Int, &sat_bit);
    }
  ACTIVATION { SAT_E2 }
}

OPERATION SUB_L_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0000111b }
  SYNTAX { "SUB"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES ( IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = Src1.Int - Src2.Int;
    }
}

OPERATION SUB_L_iil IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0100111b }
  SYNTAX { "SUB"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Long }
  BEHAVIOR USES ( IN Src1.Int; IN Src2.Int; OUT Dest.Int, Dest.Long; )
    {
      add_int_int_long (Src1.Int, -Src2.Int, &Dest.Int, &Dest.Long);
    }
}

OPERATION SUBU_L_iil IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0101111b }
  SYNTAX { "SUBU"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Long }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int, Dest.Long;)
    {
      subu_int_int_long (Src1.Int,  Src2.Int, &Dest.Int, &Dest.Long);
    }
}

OPERATION SUB_L_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0000110b }
  SYNTAX { "SUB"  ~".L"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES ( IN Src2.Int; OUT Dest.Int; )
    {
```

```
        Dest.Int = SIGN_EXT (Src1.SCst5, 5) - Src2.Int;
      }
  }
OPERATION SUB_L_cll IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 0100100b }
  SYNTAX { "SUB"  ~".L"  ~Side  Src1.SCst5  ~","  Src2.Long  ~","  Dest.Long }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long; )
    {
      U32 src2_l = Src2.Int;
      U32 src2_h = Src2.Long;
      add_int_long_long (1, ~Src2.Int, ~Src2.Long, &src2_l, &src2_h);
      add_int_long_long (SIGN_EXT (Src1.SCst5, 5), src2_l, src2_h, &Dest.Int, &Dest.Long);
    }
}
OPERATION SUB_L_iii2 IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0010111b }
  SYNTAX { "SUB"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES ( IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = Src1.Int - Src2.Int;
    }
}
OPERATION SUB_L_iil2 IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0110111b }
  SYNTAX { "SUB"  ~".L"  ~Side  ~Xpath  Src2.Long  ~","  Src1.Int  ~","  Dest.Long }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; OUT Dest.Long; )
    {
      add_int_int_long (Src1.Int, -Src2.Int,  &Dest.Int, &Dest.Long);
    }
}
OPERATION SUBU_L_iil2 IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 0111111b }
  SYNTAX { "SUB"  ~".L"  ~Side  ~Xpath  Src2.Long  ~","  Src1.Int  ~","  Dest.Long }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; OUT Dest.Long; )
    {
      subu_int_int_long (Src1.Int, Src2.Int, &Dest.Int, &Dest.Long);
    }
}
OPERATION SUBC IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1001011b }
  SYNTAX { "SUBC"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      register S32 diff = Src1.Int - Src2.Int;
      if (diff >= 0)
          Dest.Int = (diff << 1) + 1;
      else
          Dest.Int = (Src1.Int << 1);
    }
}
OPERATION XOR_L_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1101111b }
  SYNTAX { "XOR"  ~".L"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = (Src1.Int ^ Src2.Int);
    }
}
OPERATION XOR_L_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 1101110b }
  SYNTAX { "XOR"  ~".L"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = (SIGN_EXT (Src1.SCst5, 5) ^ Src2.Int);
    }
}
ALIAS OPERATION ZERO_L IN pipe.E1
{
  DECLARE
    {
      REFERENCE Side;
      GROUP Dest = { Reg };
      GROUP Xpath = { No_Xpath };
    }
  CODING { Dest  00000b  00000b  Xpath  0000111b }
  SYNTAX {  "ZERO"  ~".L"  ~Side  Dest.Int  }
  BEHAVIOR USES ( OUT Dest.Int; )
    {
      Dest.Int = Dest.Int - Dest.Int;
    }
}
```

s_unit.lisa

```
#include <c6201.h>
OPERATION Insn_S_unit IN pipe.E1
{
  DECLARE
    {
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Instruction_group = { Insn_S_unit_group1 || Insn_S_unit_group2 || Insn_S_unit_group3 || MV_S ||
                                  NEG_S || NOT_S || ZERO_S }; }
  CODING { Condition  z  Instruction_group }
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" Instruction_group }
          ACTIVATION { if (!Condition) { Instruction_group } }
        }
      ELSE
```

```
        {
          SYNTAX { " [" ~Condition ~"]" Instruction_group }
          ACTIVATION { if (Condition) { Instruction_group } }
        }
    }
  ELSE
    {
      SYNTAX { "     " Instruction_group }
      ACTIVATION { Instruction_group }
    }
}

OPERATION Field_Op IN pipe.E1
{
  DECLARE
    {
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Side  = { Side1  ||  Side2 };
      GROUP Dest, Src2, CstA, CstB = { Reg };
      GROUP Instruction = { CLR_const  ||  EXT_const  ||  EXTU_const  ||  SET_const };
    }
  CODING { Condition  z  Dest  Src2  CstA  CstB  Instruction  0010b  Side }
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" Instruction ~".S" ~Side Src2.Int ~"," CstA.UCst5 ~"," CstB.UCst5 ~"," Dest.Int }
          ACTIVATION { if (!Condition) { Instruction } }
        }
      ELSE
        {
          SYNTAX { " [" ~Condition ~"]" Instruction ~".S" ~Side Src2.Int ~"," CstA.UCst5 ~"," CstB.UCst5 ~"," Dest.Int }
          ACTIVATION { if (Condition) { Instruction } }
        }
    }
  ELSE
    {
      SYNTAX { "     " Instruction  ~".S"  ~Side Src2.Int  ~","  CstA.UCst5  ~","  CstB.UCst5  ~","  Dest.Int }
      ACTIVATION { Instruction }
    }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
}

OPERATION Insn_S_unit_group1 IN pipe.E1
{
  DECLARE
    {
      GROUP Side  = { Side1  ||  Side2 };
      GROUP Dest, Src1 = { Reg };
      GROUP Src2  = { XReg };
      GROUP Xpath = { No_Xpath  ||  Is_Xpath };
      GROUP Instruction = { ADD_S_iii || ADD_S_cii || ADD2 || AND_S_iii || AND_S_cii || B_reg || CLR_reg || EXT_reg ||
            EXTU_reg || OR_S_iii || OR_S_cii || SET_reg || SHL_iii || SHL_ici || SHL_iil || SHL_icl || SHR_iii || SHR_ici ||
            SHRU_iii || SHRU_ici || SSHL_iii || SSHL_ici || SUB_S_iii || SUB_S_cii || SUB2 || XOR_S_iii || XOR_S_cii };
    }
  CODING { Dest  Src2  Src1  Xpath  Instruction  1000b  Side }
  SYNTAX { Instruction }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
  IF ((Side == Side1) && (Xpath == Is_Xpath)) THEN
    {
      BEHAVIOR USES (IN X_path1; )
    }
  IF ((Side == Side2) && (Xpath == Is_Xpath)) THEN
    {
      BEHAVIOR USES (IN X_path2; )
    }
  ACTIVATION { Instruction }
}

OPERATION Insn_S_unit_group2 IN pipe.E1
{
  DECLARE
    {
      GROUP Side  = { Side1  ||  Side2 };
      GROUP Dest, Src1, Src2 = { Reg };
      GROUP Instruction = { SHL_lil  ||  SHL_lcl  || SHR_lil  ||  SHR_lcl  || SHRU_lil  ||  SHRU_lcl };
    }
  CODING { Dest  Src2  Src1  0bx  Instruction  1000b  Side }
  SYNTAX { Instruction }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
  ACTIVATION { Instruction }
}

OPERATION Insn_S_unit_group3 IN pipe.E1
{
  DECLARE { GROUP Instruction = { B_IRP  ||  B_NRP  || MVC_C2R  ||  MVC_R2C }; }
  CODING { Instruction }
  SYNTAX { Instruction }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
  ACTIVATION { Instruction }
}

OPERATION ADD_S_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 000111b }
  SYNTAX { "ADD"  ~".S"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Src1.Int + Src2.Int;
    }
}

OPERATION ADD_S_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 000110b }
  SYNTAX { "ADD"  ~".S"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = SIGN_EXT (Src1.SCst5, 5) + Src2.Int;
    }
}

OPERATION ADDK IN pipe.E1
{
  DECLARE
    {
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Side  = { Side1  ||  Side2 };
      GROUP Dest = { Reg };
      LABEL cst;
    }
  CODING { Condition  z  Dest  cst=0bx[16]  10100b  Side }
```

```
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" "ADDK"  ~".S"  ~Side  cst=#S  ~","  Dest.Int }
          BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
            {
              if (!Condition)
                  Dest.Int = (S32) SIGN_EXT (cst, 16) + Dest.Int;
            }
        }
      ELSE
        {
          SYNTAX { " [" ~Condition ~"]" "ADDK"  ~".S"  ~Side  cst=#S  ~","  Dest.Int }
          BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
            {
              if (Condition)
                  Dest.Int = (S32) SIGN_EXT (cst, 16) + Dest.Int;
            }
        }
    }
  ELSE
    {
      SYNTAX { "     " "ADDK"  ~".S"  ~Side  cst=#S  ~","  Dest.Int }
      BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
        {
          Dest.Int = (S32) SIGN_EXT (cst, 16) + Dest.Int;
        }
    }
}

OPERATION ADD2 IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 000001b }
  SYNTAX { "ADD2"  ~".S"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
    Dest.Int = LO16 (Src1.Int + Src2.Int) | ((HI16 (Src1.Int) + HI16 (Src2.Int)) << 16);
  }
}

OPERATION AND_S_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 011111b }
  SYNTAX { "AND"  ~".S"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
    Dest.Int = (Src1.Int & Src2.Int);
  }
}

OPERATION AND_S_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 011110b }
  SYNTAX { "AND"  ~".S"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = (SIGN_EXT (Src1.SCst5, 5) & Src2.Int);
    }
}

OPERATION B_reg IN pipe.E1
{
  DECLARE { REFERENCE Side, Src2; INSTANCE B_E2, B_E3, B_E4, B_E5; }
  CODING { 0b001101 }
  SYNTAX { "B"  ~".S"  ~Side  Src2.Int }
  BEHAVIOR  USES ( OUT PFC; )
    {
      PFC = (U32) Src2.Int;
      branch_active++;
      Branch_in_E1 = 1;
    }
  ACTIVATION { B_E2, B_E3, B_E4, B_E5 }
}

OPERATION B_IRP IN pipe.E1
{
  DECLARE { INSTANCE B_E2, B_E3, B_E4, B_E5; }
  CODING { 0b0[4]  0b0001  0b1000  0b1100  0b0[4]  0b1110  0b001 } // 0x018C0E2
  SYNTAX { "B"  ".S2"  "IRP" }
  BEHAVIOR USES ( OUT S_unit[Side2]; OUT PFC; )
    {
      PFC = (U32) irp;
      //  copy PGIE to GIE
      csr.bit.gie = csr.bit.pgie;
      branch_active++;
      Branch_in_E1 = 1;
      new_addr_set = 1;
    }
  ACTIVATION { B_E2, B_E3, B_E4, B_E5 }
}

OPERATION B_NRP IN pipe.E1
{
  DECLARE { INSTANCE B_E2, B_E3, B_E4, B_E5; }
  CODING { 0b0[4]  0b0001  0b1100  0b1110  0b0[4]  0b1110  0b001 }
  SYNTAX { "B"  ".S2"  "NRP" }
  BEHAVIOR USES ( OUT S_unit[Side2]; OUT PFC; )
    {
      PFC = (U32) nrp;
      //  set NMIE
      ier.bit.nmi = 1;
      branch_active++;
      Branch_in_E1 = 1;
      new_addr_set = 1;
    }
  ACTIVATION { B_E2, B_E3, B_E4, B_E5 }
}

OPERATION CLR_const IN pipe.E1
{
  DECLARE { REFERENCE Dest, CstA, CstB, Src2; }
  CODING { 11b }
  SYNTAX { "CLR" }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = (  EXTR_HI_BITS (Src2.Int, CstB.UCst5) | EXTR_LO_BITS (Src2.Int, CstA.UCst5));
    }
}

OPERATION CLR_reg IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 111111b }
  SYNTAX { "CLR"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;       )
    {
      U32 csta = EXTRACT(Src1.Int, 5, 5);
      U32 cstb = EXTR_LO_BITS (Src1.Int, 5);
      Dest.Int = (  EXTR_HI_BITS (Src2.Int, cstb) | EXTR_LO_BITS (Src2.Int, csta));
```

```
    }
}

OPERATION EXT_const IN pipe.E1
{
  DECLARE { REFERENCE Dest, CstA, CstB, Src2; }
  CODING { 01b }
  SYNTAX { "EXT" }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = SIGN_EXT_FIELD (Src2.Int, CstB.UCst5-CstA.UCst5, 32-CstB.UCst5);
    }
}

OPERATION EXT_reg IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 101111b }
  SYNTAX { "EXT"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
    {
      U32 csta = EXTRACT(Src1.Int, 5, 5);
      U32 cstb = EXTR_LO_BITS (Src1.Int, 5);
      Dest.Int = SIGN_EXT_FIELD (Src2.Int, cstb-csta, 32-cstb);
    }
}

OPERATION EXTU_const IN pipe.E1
{
  DECLARE { REFERENCE Dest, CstA, CstB, Src2; }
  CODING { 00b }
  SYNTAX { "EXTU" }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = EXTRACT (Src2.Int, CstB.UCst5-CstA.UCst5, 32-CstB.UCst5);
    }
}

OPERATION EXTU_reg IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 101011b }
  SYNTAX { "EXTU"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;    )
    {
      U32 csta = EXTRACT(Src1.Int, 5, 5);
      U32 cstb = EXTR_LO_BITS (Src1.Int, 5);
      Dest.Int = EXTRACT (Src2.Int, cstb-csta, 32-cstb);
    }
}

ALIAS OPERATION MV_S IN pipe.E1
{
  DECLARE
    {
      GROUP Dest = { Reg };
      GROUP Src2 = { XReg };
      GROUP Xpath = { No_Xpath  ||  Is_Xpath };
      GROUP Side = { Side1  ||  Side2 };
    }
  CODING { Dest  Src2  0b0[5]  Xpath  000110b  1000b  Side }
  SYNTAX { "MV"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = Src2.Int;
    }
}

OPERATION MVC_C2R IN pipe.E1
{
  DECLARE
    {
      GROUP Side = { Side2 };
      GROUP Dest = { Reg };
      GROUP Src  = { AMR  ||  CSR  ||  IFR  ||  IER  ||  ISTP  || IRP  ||  NRP  ||  PCE1 };
    }
  CODING { Dest  Src  0bx[5]  0bx  001111b  1000b  Side }
  SYNTAX { "MVC"  ~".S2"  Src  ~","  Dest.Int }
  BEHAVIOR USES ( OUT S_unit[Side2]; )
  IF (Src == IER) THEN
    {
      BEHAVIOR { Dest.Int = (Src & 0xFFFF); }
    }
  ELSE
    {
      BEHAVIOR { Dest.Int = Src; }
    }
}

OPERATION MVC_R2C IN pipe.E1
{
  DECLARE
    {
      GROUP Xpath = { No_Xpath  ||  Is_Xpath };
      GROUP Dest = { AMR  ||  CSR  ||  ISR  ||  ICR  ||  IER  ||  ISTP  ||  IRP  ||  NRP };
      GROUP Src2 = { XReg };
      GROUP Side = { Side2 };
      INSTANCE MVC_ICR_E2, MVC_ISR_E2;
    }
  CODING { Dest Src2  0bx[5]  Xpath  001110b  1000b Side }
  SYNTAX { "MVC"  ~".S2"  Xpath  Src2.Int  ~","  Dest }
  BEHAVIOR USES ( OUT S_unit[Side]; )
  SWITCH (Dest)
    {
      CASE AMR:  { BEHAVIOR { Dest   = (Src2.Int & 0x03FFFFF); } }
      CASE CSR:  { BEHAVIOR { Dest   = ((csr.word & 0xFFFF0100) | (Src2.Int & 0xFCFF)); } }
      CASE ISR:
        {
          BEHAVIOR { S_data[Side] = LO16 (Src2.Int); }
          ACTIVATION { MVC_ISR_E2 }
        }
      CASE ICR:
        {
          BEHAVIOR { S_data[Side] = LO16 (Src2.Int); }
          ACTIVATION { MVC_ICR_E2 }
        }
      CASE IER:  { BEHAVIOR { Dest   = (LO16 (Src2.Int) | 1); } }
      CASE ISTP: { BEHAVIOR { Dest   = ((istp.word & 0x3E0) | (Src2.Int & 0xFFFFFC00)); } }
      DEFAULT:   { BEHAVIOR { Dest   = Src2.Int; } }
    }
}

OPERATION MVK IN pipe.E1
{
  DECLARE
    {
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Side  = { Side1  ||  Side2 };
```

```
        GROUP Dest = { Reg };
        LABEL cst;
      }
    CODING { Condition  z  Dest  cst=0bx[16]  01010b  Side }
    IF (Condition == Is_Conditional) THEN
      {
        IF (z == Is_Zero) THEN
          {
            SYNTAX { "[!" ~Condition ~"]" "MVK"  ~".S"  ~Side  SYMBOL("0x" cst=#X)  ~","  Dest.Int }
            BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side], Dest.Int; )
              {
                if (!Condition)
                    Dest.Int = (S32) SIGN_EXT (cst, 16);
              }
          }
        ELSE
          {
            SYNTAX { " [" ~Condition ~"]" "MVK"  ~".S"  ~Side  SYMBOL("0x" cst=#X)  ~","  Dest.Int }
            BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side], Dest.Int; )
              {
                if (Condition)
                    Dest.Int = (S32) SIGN_EXT (cst, 16);
              }
          }
      }
    ELSE
      {
        SYNTAX { "     " "MVK"  ~".S"  ~Side  SYMBOL("0x" cst=#X)  ~","  Dest.Int }
        BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side], Dest.Int; )
          {
            Dest.Int = (S32) SIGN_EXT (cst, 16);
          }
      }
}

ALIAS OPERATION MVKH IN pipe.E1
{
  DECLARE
    {
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Side  = { Side1  ||  Side2 };
      GROUP Dest = { Reg };
      LABEL cst;
    }
  CODING { Condition  z  Dest  cst=0bx[16]  11010b  Side }
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" "MVKH"  ~".S"  ~Side  "0x" ~(cst << 16)=#X  ~","  Dest.Int }
          BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
            {
              if (!Condition)
                  Dest.Int = ((cst << 16) | (LO16 (Dest.Int)));
            }
        }
      ELSE
        {
          SYNTAX { " [" ~Condition ~"]" "MVKH"  ~".S"  ~Side  "0x" ~(cst << 16)=#X  ~","  Dest.Int }
          BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
            {
              if (Condition)
                  Dest.Int = ((cst << 16) | (LO16 (Dest.Int)));
            }
        }
    }
  ELSE
    {
      SYNTAX { "     " "MVKH"  ~".S"  ~Side  "0x" ~(cst << 16)=#X  ~","  Dest.Int }
      BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
        {
          Dest.Int = ((cst << 16) | (LO16 (Dest.Int)));
        }
    }
}

OPERATION MVKLH IN pipe.E1
{
  DECLARE
    {
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Side  = { Side1  ||  Side2 };
      GROUP Dest = { Reg };
      LABEL cst;
    }
  CODING { Condition  z  Dest  cst=0bx[16]  11010b  Side }
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" "MVKLH"  ~".S"  ~Side  cst=#S  ~","  Dest.Int }
          BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
            {
              if (!Condition)
                  Dest.Int = ((cst << 16) | (LO16 (Dest.Int)));
            }
        }
      ELSE
        {
          SYNTAX { " [" ~Condition ~"]" "MVKLH"  ~".S"  ~Side  cst=#S  ~","  Dest.Int }
          BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
            {
              if (!Condition)
                  Dest.Int = ((cst << 16) | (LO16 (Dest.Int)));
            }
        }
    }
  ELSE
    {
      SYNTAX { "     " "MVKLH"  ~".S"  ~Side  cst=#S  ~","  Dest.Int }
      BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT S_unit [Side]; )
        {
          Dest.Int = ((cst << 16) | (LO16 (Dest.Int)));
        }
    }
}

ALIAS OPERATION NEG_S IN pipe.E1
{
  DECLARE
    {
      GROUP Dest = { Reg };
      GROUP Src2 = { XReg };
      GROUP Xpath = { No_Xpath  ||  Is_Xpath };
      GROUP Side = { Side1  ||  Side2 };
    }
```

```
  CODING { Dest  Src2  0b0[5]  Xpath  010110b  1000b  Side }
  SYNTAX { "NEG"  ~".S"  ~Side  Src2.Int  ~","  Dest.Int  }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = - Src2.Int;
    }
}

ALIAS OPERATION NOT_S IN pipe.E1
{
  DECLARE
    {
      GROUP Dest = { Reg };
      GROUP Src1 = { Minus_one };
      GROUP Src2 = { XReg };
      GROUP Side = { Side1  ||  Side2 };
      GROUP Xpath = { No_Xpath  ||  Is_Xpath };
    }
  CODING { Dest  Src2  Src1  Xpath  001010b  1000b  Side }
  SYNTAX { "NOT"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = (SIGN_EXT (Src1, 5) ^ Src2.Int);
    }
}

OPERATION OR_S_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 011011b }
  SYNTAX { "OR"  ~".S"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
  {
    Dest.Int = (Src1.Int | Src2.Int);
  }
}

OPERATION OR_S_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 011010b }
  SYNTAX { "OR"  ~".S"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
  {
    Dest.Int = (SIGN_EXT (Src1.SCst5, 5) | Src2.Int);
  }
}

OPERATION SET_const IN pipe.E1
{
  DECLARE { REFERENCE Dest, CstA, CstB, Src2; }
  CODING { 10b }
  SYNTAX { "SET" }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
  {
    Dest.Int = Src2.Int | (((U32) 0xFFFFFFFF >> (31-(CstB.UCst5-CstA.UCst5))) << CstA.UCst5);
  }
}

OPERATION SET_reg IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 111011b }
  SYNTAX { "SET"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.Int ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
    {
      register U32 csta = EXTRACT(Src1.Int, 5, 5);
      register U32 cstb = EXTR_LO_BITS (Src1.Int, 5);
      Dest.Int = Src2.Int | (((U32) 0xFFFFFFFF >> (32-cstb)) << csta);
    }
}

OPERATION SHL_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 110011b }
  SYNTAX { "SHL"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = (Src2.Int << (Src1.Int & 0x3F));
    }
}

OPERATION SHL_lil IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 110001b }
  SYNTAX { "SHL"  ~".S"  ~Side  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long; )
    {
      shl_long (Src2.Int, Src2.Long, (Src1.Int & 0x3F), &Dest.Int, &Dest.Long);
    }
}

OPERATION SHL_iil IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 010011b }
  SYNTAX { "SHL"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; OUT Dest.Long; )
    {
      shl_long (Src2.Int, 0, (Src1.Int & 0x3F), &Dest.Int, &Dest.Long);
    }
}

OPERATION SHL_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 110010b }
  SYNTAX { "SHL"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = (Src2.Int << Src1.UCst5);
    }
}

OPERATION SHL_lcl IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 110000b }
  SYNTAX { "SHL"  ~".S"  ~Side  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long; )
    {
      shl_long (Src2.Int, Src2.Long, Src1.UCst5, &Dest.Int, &Dest.Long);
    }
}

OPERATION SHL_icl IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 010010b }
```

```
  SYNTAX { "SHL"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int; OUT Dest.Long;  )
    {
      shl_long (Src2.Int, 0, Src1.UCst5, &Dest.Int, &Dest.Long);
    }
}

OPERATION SHR_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 110111b }
  SYNTAX { "SHR"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;    )
    {
      Dest.Int = ((S32) Src2.Int >> (Src1.Int & 0x3F));
    }
}

OPERATION SHR_lil IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 110101b }
  SYNTAX { "SHR"  ~".S"  ~Side  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long; )
    {
      shr_long (Src2.Int, Src2.Long, (Src1.Int & 0x3F), &Dest.Int, &Dest.Long);
    }
}

OPERATION SHR_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 110110b }
  SYNTAX { "SHR"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = ((S32) Src2.Int >> Src1.UCst5);
    }
}

OPERATION SHR_lcl IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 110100b }
  SYNTAX { "SHR"  ~".S"  ~Side  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long; )
    {
      shr_long (Src2.Int, Src2.Long, Src1.UCst5, &Dest.Int, &Dest.Long);
    }
}

OPERATION SHRU_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 100111b }
  SYNTAX { "SHRU"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;    )
    {
      Dest.Int = ((U32) Src2.Int >> (Src1.Int & 0x3F));
    }
}

OPERATION SHRU_lil IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 100101b }
  SYNTAX { "SHRU"  ~".S"  ~Side  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long; )
    {
      shru_long (Src2.Int, Src2.Long, (Src1.Int & 0x3F), &Dest.Int, &Dest.Long);
    }
}

OPERATION SHRU_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 100110b }
  SYNTAX { "SHRU"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = ((U32) Src2.Int >> Src1.UCst5);
    }
}

OPERATION SHRU_lcl IN pipe.E1
{
  DECLARE { REFERENCE Side, Src1, Src2, Dest; }
  CODING { 100100b }
  SYNTAX { "SHRU"  ~".S"  ~Side  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; IN Src2.Long; OUT Dest.Int; OUT Dest.Long; )
    {
      shru_long (Src2.Int, Src2.Long, Src1.UCst5, &Dest.Int, &Dest.Long);
    }
}

OPERATION SSHL_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 100011b }
  SYNTAX { "SSHL"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; OUT Dest.Long;)
    {
      shl_long (Src2.Int, 0, (Src1.Int & 0x3F), &Dest.Int, &Dest.Long);
      saturate (Dest.Int, Dest.Long, &Dest.Int, &sat_bit);
    }
}

OPERATION SSHL_ici IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 100010b }
  SYNTAX { "SSHL"  ~".S"  ~Side  ~Xpath  Src2.Int  ~","  Src1.UCst5  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int; OUT Dest.Long;)
    {
      shl_long (Src2.Int, 0, Src1.UCst5, &Dest.Int, &Dest.Long);
      saturate (Dest.Int, Dest.Long, &Dest.Int, &sat_bit);
    }
}

OPERATION SUB_S_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 010111b }
  SYNTAX { "SUB"  ~".S"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;    )
    {
      Dest.Int = Src1.Int - Src2.Int;
    }
}

OPERATION SUB_S_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
```

```
  CODING { 000110b }
  SYNTAX { "SUB"  ~".S"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = SIGN_EXT (Src1.SCst5, 5) - Src2.Int;
    }
}

OPERATION SUB2 IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 010001b }
  SYNTAX { "SUB2"  ~".S"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = LO16 (Src1.Int - Src2.Int) | ((HI16 (Src1.Int) - HI16 (Src2.Int)) << 16);
    }
}

OPERATION XOR_S_iii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 001011b }
  SYNTAX { "XOR"  ~".S"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
    {
      Dest.Int = (Src1.Int ^ Src2.Int);
    }
}

OPERATION XOR_S_cii IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 001010b }
  SYNTAX { "XOR"  ~".S"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src2.Int; OUT Dest.Int;)
    {
      Dest.Int = (SIGN_EXT (Src1.SCst5, 5) ^ Src2.Int);
    }
}

ALIAS OPERATION ZERO_S IN pipe.E1
{
  DECLARE
    {
      GROUP Side = { Side1  ||  Side2 };
      GROUP Dest = { Reg };
      GROUP Xpath = { No_Xpath };
    }
  CODING { Dest  Dest  Dest  Xpath  010111b  1000b  Side }
  SYNTAX {  "ZERO"  ~".S"  ~Side  Dest.Int  }
  BEHAVIOR USES ( OUT Dest.Int; )
    {
      Dest.Int = Dest.Int - Dest.Int;
    }
}
```

m_unit.lisa

```
#include <c6201.h>

OPERATION Insn_M_unit IN pipe.E1
{
  DECLARE
    {
      GROUP Side  = { Side1  ||  Side2 };
      GROUP Condition  = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      GROUP Dest, Src1 = { Reg };
      GROUP Src2  = { XReg };
      GROUP Xpath = { No_Xpath  ||  Is_Xpath };
      GROUP Instruction = { MPY || MPYU || MPYUS || MPYSU || MPYC || MPYSUC || MPYH || MPYHU || MPYHUS || MPYHSU || MPYHL ||
            MPYHLU || MPYHULS || MPYHSLU || MPYLH || MPYLHU || MPYLUHS || MPYLSHU || SMPY || SMPYHL || SMPYLH || SMPYH };
      INSTANCE MPY_E2;
    }
  CODING { Condition  z  Dest  Src2  Src1  Xpath  Instruction  00000b  Side }
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" Instruction }
          ACTIVATION { if (!Condition) { Instruction, MPY_E2 } }
        }
      ELSE
        {
          SYNTAX { " [" ~Condition ~"]" Instruction }
          ACTIVATION { if (Condition) { Instruction, MPY_E2 }
            }
        }
    }
  ELSE
    {
      SYNTAX { "     " Instruction }
      ACTIVATION { Instruction, MPY_E2 }
    }
  BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT M_unit [Side]; )
  IF ((Side == Side1) && (Xpath == Is_Xpath)) THEN
    {
      BEHAVIOR USES (IN X_path1; )
    }
  IF ((Side == Side2) && (Xpath == Is_Xpath)) THEN
    {
      BEHAVIOR USES (IN X_path2; )
    }
}

OPERATION MPY IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 11001b }
  SYNTAX { "MPY"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;  )
    {
      M_data[Side] = ((S32) SIGN_EXT (Src1.Int, 16)) * ((S32) SIGN_EXT (Src2.Int, 16));
    }
}

OPERATION MPYU IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 11111b }
  SYNTAX { "MPYU" ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;  )
```

```
    {
      M_data[Side] = ((U32) LO16 (Src1.Int)) * ((U32) LO16 (Src2.Int));
    }
}

OPERATION MPYUS IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 11101b }
  SYNTAX { "MPYUS"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;  )
    {
      M_data[Side] = ((U32) LO16 (Src1.Int)) * ((S32) SIGN_EXT (Src2.Int, 16));
    }
}

OPERATION MPYSU IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 11011b }
  SYNTAX { "MPYSU"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      M_data[Side] = ((S32) SIGN_EXT (Src1.Int, 16)) * ((U32) LO16 (Src2.Int));
    }
}

OPERATION MPYC IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 11000b }
  SYNTAX { "MPY"  ~".M"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.SCst5, Src2.Int; OUT Dest.Int;)
    {
      M_data[Side] = ((S32) SIGN_EXT (Src1.SCst5, 5)) * ((S32) SIGN_EXT (Src2.Int, 16));
    }
}

OPERATION MPYSUC IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 11110b }
  SYNTAX { "MPYSU"  ~".M"  ~Side  ~Xpath  Src1.SCst5  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.SCst5, Src2.Int; OUT Dest.Int; )
  {
    M_data[Side] = ((S32) SIGN_EXT (Src1.SCst5, 5)) * ((U32) LO16 (Src2.Int));
  }
}

OPERATION MPYH IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 00001b }~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
    M_data[Side] = ((S32) (SIGN_EXT ((Src1.Int >> 16), 16))) * (((S32) (SIGN_EXT ((Src2.Int >> 16), 16))));
  }
}

OPERATION MPYHU IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 00111b }
  SYNTAX { "MPYHU"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
  {
    M_data[Side] = ((U32) HI16 (Src1.Int)) * ((U32) HI16 (Src2.Int));
  }
}

OPERATION MPYHUS IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 00101b }
  SYNTAX { "MPYHUS"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int; )
  {
    M_data[Side] = ((U32) HI16 (Src1.Int)) * (((S32) (SIGN_EXT ((Src2.Int >> 16), 16))));
  }
}

OPERATION MPYHSU IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 00011b }
  SYNTAX { "MPYHSU"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
    M_data[Side] = (((S32) (SIGN_EXT ((Src1.Int >> 16), 16)))) * ((U32) HI16 (Src2.Int));
  }
}

OPERATION MPYLH IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 10001b }
  SYNTAX { "MPYLH"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
    M_data[Side] = ((S32) (SIGN_EXT ((Src2.Int >> 16), 16))) * ((S32) SIGN_EXT (Src1.Int, 16));
  }
}

OPERATION MPYLHU IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 10111b }
  SYNTAX { "MPYLHU"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
    M_data[Side] = ((U32) LO16 (Src1.Int)) * ((U32) HI16 (Src2.Int));
  }
}

OPERATION MPYLUHS IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 10101b }
  SYNTAX { "MPYLUHS"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
    M_data[Side] = ((U32) LO16 (Src1.Int))
      * (((S32) (SIGN_EXT ((Src2.Int >> 16), 16))));
  }
}

OPERATION MPYLSHU IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 10011b }
  SYNTAX { "MPYLSHU"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
```

```
    M_data[Side] = ((S32) SIGN_EXT (Src1.Int, 16)) * ((U32) (Src2.Int));
  }
}

OPERATION MPYHL IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 01001b }
  SYNTAX { "MPYHL" ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
    M_data[Side] = (((S32) (SIGN_EXT ((Src1.Int >> 16), 16))))  * ((S32) SIGN_EXT (Src2.Int, 16));
  }
}

OPERATION MPYHLU IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 01111b }
  SYNTAX { "MPYHLU"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
    M_data[Side] = ((U32) HI16 (Src1.Int))  * ((U32) LO16 (Src2.Int));
  }
}

OPERATION MPYHULS IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 01101b }
  SYNTAX { "MPYHULS"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
    M_data[Side] = ((U32) HI16 (Src1.Int)) * ((S32) SIGN_EXT (Src2.Int, 16));
  }
}

OPERATION MPYHSLU IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 01011b }
  SYNTAX { "MPYHSLU"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
  {
    M_data[Side] = (((S32) (SIGN_EXT ((Src1.Int >> 16), 16)))) * ((U32) LO16 (Src2.Int));
  }
}

OPERATION SMPY IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 11010b }
  SYNTAX { "SMPY" ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
   {
      M_data[Side] = ((  ((S32) SIGN_EXT (Src1.Int, 16)) * ((S32) SIGN_EXT (Src2.Int, 16))) << 1);
      if (M_data[Side] == 0x80000000)
        {
          M_data[Side] = 0x7FFFFFFF;
        }
   }
}

OPERATION SMPYHL IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 01010b }
  SYNTAX { "SMPYHL"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      M_data[Side] = ((  (((S32) (SIGN_EXT ((Src1.Int >> 16), 16)))) * ((S32) SIGN_EXT (Src2.Int, 16))) << 1);
      if (M_data[Side] == 0x80000000)
          M_data[Side] = 0x7FFFFFFF;
    }
}

OPERATION SMPYLH IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 10010b }
  SYNTAX { "SMPYLH"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      M_data[Side] = ((((S32) SIGN_EXT (Src1.Int, 16)) * (((S32) (SIGN_EXT ((Src2.Int >> 16), 16))))) << 1);
      if (M_data[Side] == 0x80000000)
          M_data[Side] = 0x7FFFFFFF;
    }
}

OPERATION SMPYH IN pipe.E1
{
  DECLARE { REFERENCE Side, Xpath, Src1, Src2, Dest; }
  CODING { 00010b }
  SYNTAX { "SMPYH"  ~".M"  ~Side  ~Xpath  Src1.Int  ~","  Src2.Int  ~","  Dest.Int }
  BEHAVIOR USES (IN Src1.Int; IN Src2.Int; OUT Dest.Int;)
    {
      M_data[Side] = (((((S32) (SIGN_EXT ((Src1.Int >> 16), 16)))) * (((S32) (SIGN_EXT ((Src2.Int >> 16), 16))))) << 1);
      if (M_data[Side] == 0x80000000)
          M_data[Side] = 0x7FFFFFFF;
    }
}
```

misc.lisa

```
#include <c6201.h>

OPERATION BRANCH IN pipe.E1
{
  DECLARE
    {
      GROUP Side = { Side1  ||  Side2 };
      GROUP Condition = { Is_Unconditional  ||  Is_Conditional };
      GROUP z = { Is_Zero  ||  Is_Nonzero };
      LABEL cst;
      INSTANCE B_E2, B_E3, B_E4, B_E5;
    }
  CODING { Condition  z  cst=0bx[21]  0b00100  Side}
  IF (Condition == Is_Conditional) THEN
    {
      IF (z == Is_Zero) THEN
        {
          SYNTAX { "[!" ~Condition ~"]" "B" ~".S" ~Side  SYMBOL("0x"((((CURRENT_ADDRESS >> 5) << 5) +((cst=#S21)<<2)))=#X) }
          BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT PFC; )
            {
              if (!Condition)
                {
                  PFC = SIGN_EXT (cst << 2, 23) + (pce1_forward[4] & PFC_MASK);
```

```
                    Branch_in_E1 = 1;
                    branch_active++;
                  }
              }
            ACTIVATION { if (!Condition) { B_E2, B_E3, B_E4, B_E5 } }
          }
        ELSE
          {
            SYNTAX { " [" ~Condition ~"]" "B"  ~".S" ~Side SYMBOL("0x" ((((CURRENT_ADDRESS >> 5) << 5) +((cst=#S21)<<2)))=#X) }
            BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT PFC; )
              {
                if (Condition)
                  {
                    PFC = SIGN_EXT (cst << 2, 23) + (pce1_forward[4] & PFC_MASK);
                    Branch_in_E1 = 1;
                    branch_active++;
                  }
              }
            ACTIVATION { if (Condition) { B_E2, B_E3, B_E4, B_E5 } }
          }
      }
    ELSE
      {
        SYNTAX { "     " "B"  ~".S"  ~Side  SYMBOL ("0x" ((((CURRENT_ADDRESS >> 5) << 5) +((cst=#S21)<<2)))=#X) }
            BEHAVIOR REQUIRES ( !pipeline_stall ) USES ( OUT PFC; )
              {
                PFC = SIGN_EXT (cst << 2, 23) + (pce1_forward[4] & PFC_MASK);
                Branch_in_E1 = 1;
                branch_active++;
              }
            ACTIVATION { B_E2, B_E3, B_E4, B_E5 }
      }
}

OPERATION NOP IN pipe.E1
{
  DECLARE { GROUP nop_cycles = { NOP0 || NOP1 || NOP2 || NOP3 || NOP4 || NOP5 || NOP6 || NOP7 || NOP8 }; }
  CODING { 0bx[14]  0b0  nop_cycles  0b0[12] }
  SYNTAX { "     NOP"  nop_cycles }
  ACTIVATION { nop_cycles }
}

OPERATION NOP0 IN pipe.E1
{
  DECLARE { INSTANCE End_Multi_Cycle_NOP; }
  ACTIVATION {  End_Multi_Cycle_NOP }
  CODING { 0b0000 }
  SYNTAX { "1" }
  BEHAVIOR { pce1 -= 4; }
}

OPERATION NOP1 IN pipe.E1
{
  DECLARE { INSTANCE End_Multi_Cycle_NOP; }
  ACTIVATION { ; End_Multi_Cycle_NOP }
  CODING { 0b0001 }
  SYNTAX { "2" }
  BEHAVIOR { pce1 -= 4; multicycle_nop = 1;}
}

OPERATION NOP2 IN pipe.E1
{
  DECLARE { INSTANCE End_Multi_Cycle_NOP; }
  ACTIVATION { ;; End_Multi_Cycle_NOP }
  CODING { 0b0010 }
  SYNTAX { "3" }
  BEHAVIOR { pce1 -= 4; multicycle_nop = 1;}
}

OPERATION NOP3 IN pipe.E1
{
  DECLARE { INSTANCE End_Multi_Cycle_NOP; }
  ACTIVATION { ;;; End_Multi_Cycle_NOP }
  CODING { 0b0011 }
  SYNTAX {"4" }
  BEHAVIOR { pce1 -= 4; multicycle_nop = 1;}
}

OPERATION NOP4 IN pipe.E1
{
  DECLARE { INSTANCE End_Multi_Cycle_NOP; }
  ACTIVATION { ;;;; End_Multi_Cycle_NOP }
  CODING { 0b0100 }
  SYNTAX { "5" }
  BEHAVIOR { pce1 -= 4; multicycle_nop = 1;}
}

OPERATION NOP5 IN pipe.E1
{
  DECLARE { INSTANCE End_Multi_Cycle_NOP; }
  ACTIVATION { ;;;;; End_Multi_Cycle_NOP }
  SYNTAX { "6" }
  CODING { 0b0101 }
  BEHAVIOR { pce1 -= 4; multicycle_nop = 1;}
}

OPERATION NOP6 IN pipe.E1
{
  DECLARE { INSTANCE End_Multi_Cycle_NOP; }
  ACTIVATION { ;;;;;; End_Multi_Cycle_NOP }
  CODING { 0b0110 }
  SYNTAX { "7" }
  BEHAVIOR { pce1 -= 4; multicycle_nop = 1;}
}

OPERATION NOP7 IN pipe.E1
{
  DECLARE { INSTANCE End_Multi_Cycle_NOP; }
  ACTIVATION { ;;;;;;; End_Multi_Cycle_NOP }
  CODING { 0b0111 }
  SYNTAX { "8" }
  BEHAVIOR { pce1 -= 4; multicycle_nop = 1;}
}

OPERATION NOP8 IN pipe.E1
{
  DECLARE { INSTANCE End_Multi_Cycle_NOP; }
  ACTIVATION { ;;;;;;;; End_Multi_Cycle_NOP }
  CODING { 0b1000 }
  SYNTAX { "9" }
  BEHAVIOR { pce1 -= 4; multicycle_nop = 1;}
}
```

Lebenslauf

Stefan Leo Alexander Pees, geboren am 23. Februar 1968 in Aachen
verheiratet mit Karin Pees, geb. Neumann
Eine Tochter: Annette Pees

1974-1978	Gemeinschaftsgrundschule Laurensberg
1978-1988	Kaiser-Karls-Gymnasium Aachen Abschluss: Abitur
1989-1990	Grundwehrdienst
1989-1995	Studium der Elektrotechnik Fachrichtung Allgemeine Elektrotechnik an der RWTH Aachen Abschluss: Diplom-Ingenieur
Jan. 1996- Feb. 2001	Wissenschaftlicher Angestellter am Lehrstuhl für Integrierte Systeme der Signalverarbeitung der RWTH Aachen
seit Feb. 2001	Mitarbeiter der AXYS GmbH, Aachen als Engineering Manager