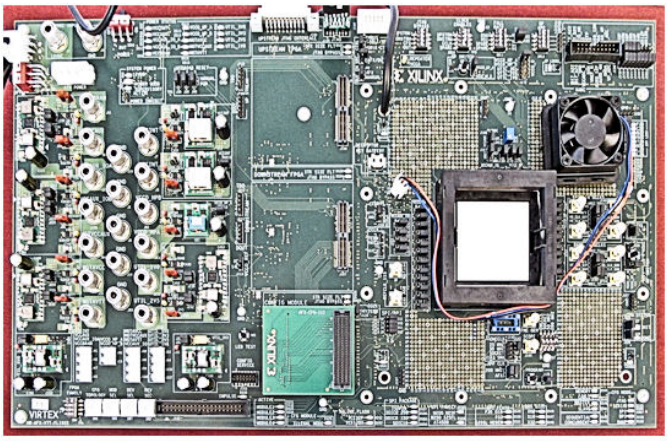
TÉCNICO LISBOA



C Compiler for the VERSAT Reconfigurable Processor

Gonçalo da Conceição Reis dos Santos

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. José João Henriques Teixeira de Sousa

Examination Committee

Chairperson:	Prof. Francisco André Corrêa Alegria
Supervisor:	Prof. José João Henriques Teixeira de Sousa
Member of the Committee:	Prof. Paulo Ferreira Godinho Flores

November 2019

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to express my most sincere gratitude to my advisor, Professor José Teixeira de Sousa, for his guidance during the development of this thesis. In particular for his relentless patience in the early stages of this work and many useful improvements and suggestions. Without his suggestions and support, specially in the hectic days before the completion of my work, this thesis would not have been possible.

I also wish to thank Professor Paulo Flores since his insightful comments allowed me to improve this dissertation.

I am also thankful to my friends and course mates who helped me grow as a person and made it less difficult to be engaged in my master's degree. I am thankful to my friends who were by my side during this phase, for companionship, strength and support in difficult times.

In these acknowledgments, I also cannot forget my parents for their persistent encouragements.

Resumo

Nos últimos anos, a computação reconfigurável tem recebido grande atenção, pois permite mudar a arquitetura dinamicamente. *Versat* é uma dessas arquiteturas reconfiguráveis. O objetivo deste trabalho é fornecer um compilador da linguagem **C** para o *picoVersat*, o controlador do *Versat*. O *picoVersat* possui um conjunto muito reduzido de instruções, realizando cálculos simples e controlando os subsistemas a ele ligados. Foi escolhido o compilador **lcc** porque permite múltiplos processadores alvo, está bem documentado e utiliza uma ferramenta de seleção das instruções, facilitando o processo de geração de código. Cada processador alvo do **lcc** é configurado por uma estrutura que parameteriza o gerador alvo. A reserva de registos pode ajustar-se às caraterísticas de cada processador e uma árvore gramatical permite descrever a maioria das operações, associando-lhes um custo. Consegue-se a otimização da seleção de instruções fornecendo diferentes combinações de árvores gramaticais com custos distintos. As instruções mais complexas podem ser codificadas manualmente, tal como o registo de ativação das funções. Adicionou-se ao **lcc** a geração direta de instruções em *assembly*, pois não é uma função ANSI **C**. Como o *Versat* é configurado através da escrita em posições específicas de memória, o seu controlo realiza-se com simples instruções de atribuição em **C**. Foram realizados testes extensivos ao compilador num ambiente de desenvolvimento integrado. O compilador realizado permite controlar o *Versat* numa linguagem de alto-nível. O número reduzido de instruções disponíveis no *picoVersat* implica que as operações na pilha, como manipulação de argumentos e funções, resulta em longas sequências de instruções.

Palavras-chave: Versat, CGRA, picoVersat, compilador de C, geração de código com lcc.

Abstract

Reconfigurable computing has had a great focus in the past decades as it promises to combine the performance of dedicated hardware with the flexibility of software. *Versat* is a Coarse Grained Reconfigurable Array architecture capable of dynamic and partial reconfiguration. The purpose of this work is to provide a full **C** language compiler for *picoVersat*, the *Versat* hardware controller. *PicoVersat* has a minimalist instruction set, can be programmed in its assembly, and is used to perform simple calculations and reconfiguring Versat.

The **lcc** framework was chosen because it is a retargetable compiler, well documented and uses a code selection tool to help the code generation process. Each **lcc** *back-end* is configured through a structure that parameterizes the *back-end* code generator. Instruction selection optimization is achieved by providing different tree grammar combination of different costs. Complex instructions can be coded manually as well as function activation records. Assembly in-lining through the `asm` routine was added to the compiler *front-end*, since it is not ANSI **C**. The *Versat* configuration is memory mapped, thus each functional unit can be configured by writing to specific memory addresses with an ordinary **C** language assignment instruction. The compiler was integrated into a compilation framework, and extensive testing was performed.

The resulting compiler allows all benefits of the **C** high-level language. Low level assembly instructions are still possible through the `asm` directive. The minimalist *picoVersat* instruction set results in long code sequencies, specially for stack operations like argument or function manipulation.

Keywords: Versat, CGRA, picoVersat, C-compiler, lcc *back-end*.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past decades there has been a great focus in reconfigurable computing. Due to the fact that these systems can change their architecture dynamically to be best suit to the task being executed, they can greatly accelerate the execution of applications mainly in the fields reigned by embedded systems, like telecommunications, multimedia, etc [6, 30, 28]. Many of these embedded applications need to have low power consumption, and reduced silicon area while still having a high number of computations done per second.

Versat is one of these architectures capable of changing component functions and interconnections dynamically [31]. This system uses a Data Engine in order to execute code loops. *Versat*, like architectures of this type, is not designed to run efficiently when the code has a great amount of different instructions, but rather when the code revolves around loops and repeated instructions in general. To do the more eclectic code there is a controller that also manages *Versat*, making it run and taking care of the reconfigurations.

For an architecture like this there is not only the way the components operate themselves but also the coding and execution management of the system. For the coding aspect of these systems the best solution would be to have a compiler for the system that allowed the effective use of the components while being able to be coded in a language easily understood by humans [25]. Unfortunately there is not still an universal, or even close to universal, compiler for these types of architectures. Even so, a compiler for this architecture was made [40] but it still has some flaws that need to be resolved.

So, the point of this work is to replace the existing *Versat* compiler solving the issues that the previous one still has. For this, some different options will be discussed, keeping in mind that this is not a traditional CPU architecture. This means that there is need to study compilers in general as well as the architecture in question.

1.1 Context

Coarse Grain Reconfigurable Arrays (CGRAs) are reconfigurable architectures that have been getting a lot of attention in the past decades. These architectures are usually used in embedded systems

because they are designed to fit very specific needs. This means that even though they are less flexible by nature they also have less delay, area, configuration time and consume less power compared to Field Programmable Gate Arrays (FPGAs), the most commonly used reconfigurable architecture. As such CGRAs are meant to work as accelerators.

These architectures can either be configured a single time per run of an application or configured during the execution of the application. This means that CGRAs can either have static or dynamic configurations, respectively. Static configurations are less flexible while dynamic configurations have to account for the time spent doing the reconfiguration itself.

A compiler for an architecture like this has to take into account these differences between a traditional architecture, and a reconfigurable one. However, the basic structure and behavior of the compiler remains unchanged.

1.2 Motivation

High-level languages, like **C**/**C++** or Java, allow for program development without the knowledge of the inner workings of a given processor. Furthermore, the development is faster and programs are easier to debug or test than using an assembly language for a specific processor. *Versat* is programmable in assembly using the *picoVersat* instruction set. Using the assembler program available for program development is a cumbersome form to port existing algorithms to the *Versat* platform, most of them originally written in **C** or **C++**. A compiler is essential for the test and assessment of the *Versat* platform capabilities. A first prototype compiler, developed by Rui Santiago [40], exists and was used for the development of some examples. However, the compiler is not only not very practical still, but also quite primitive. It supports a limited subset of the **C** language with **C++** alike method invocation, with no variables, functions or structures. All programming uses the *picoVersat* register names to hold values. Furthermore, the only **C++** syntax used is the method invocation from object (object.method), but this being the only difference from **C**, and since there is not the option to create multiple instances of these objects (they act as variables), this syntax could simply be replaced with a **C** compatible syntax. This would allow for the use of a simpler compiler, since in reality none of the extra functionalities of **C++** in relation to **C** are even supposed to be used in this architecture.

1.3 Objectives

The objective of this work is supersede the existing compiler Versat compiler improving the following aspects:

- Partial reconfiguration: prepare the next *Versat* configuration by changing only the configuration fields that differ. Currently all configuration bits are updated during the reconfiguration, which costs time.
- Support a variable number of functional units: the current implementation assumes a fixed number

of functional units, and if the hardware is changed, then the compiler needs to change too. This improvement will lift this restriction.

- Support variable declarations: currently all variables and objects are predefined and user defined variables are not supported. This improvement will greatly improve programmability.
- Support function calls: currently all the code must be written in a single `main()` function and no other functions can be called from this one. This improvement will allow not only for better programming practices, but also to make the code more extensible and easy to understand.

1.4 Challenges

Building a full **C** compiler on such a bare/simple instruction set requires that most instructions need to be decomposed. All **C** language operations that are not directly supported by the architecture have to be mapped into the few existing ones.

Having such reduced register support in *picoVersat* requires constant memory accesses, and not having a *stack pointer*, or a *frame pointer*, makes function calls (in particular recursion) slow.

Detecting, along with regular **C** code, the portions that are meant to be computed into instructions for *Versat* instead of *picoVersat* is difficult, not only because there is a need to identify these ”packs” of instructions, as well as to analyze and to compute these instructions in a different way.

1.5 Document structure

This document is divided into six chapters. The next chapter, State of the Art, introduces the *Versat* architecture in its present form, and surveys the existing **B** and **C** language compiler frameworks that can be adapted into a full *Versat* compiler. Chapter 3, Architecture, provides the general architecture of the compiler for the *Versat* platform. The **lcc** retargetable **C** compiler was selected because it can be easely extended by providing a new *Versat back-end*. The following chapter, Development, explores the intrinsics of the compiler *back-end* development and its integration into the **lcc** framework. Chapter 5, Results, addresses the compiler functionality, efficiency and limitations. The last chapter, Conclusions, sums up the work undertaken.

Chapter 2

State of the Art

In this chapter a description of the current state of compiler frameworks and their application to reconfigurable processors is presented. The chapter begins by explaining the concept of Coarse Grained Reconfigurable Array (CGRA), in order to introduce the Versat reconfigurable processor. Then, several compiler frameworks are studied with the purpose of selecting a tool for the development of the Versat compiler. A section about CGRA compilers and a comparative analysis closes the chapter.

2.1 CGRA

A CGRA is a programmable hardware structure made of coarse grained components such as Arithmetic and Logic Units (ALUs) whose function (add, multiply, shift, *etc.*) and interconnection can be programmed differently for executing different applications [46]. CGRAs can be controlled by one or more host processors. By selecting the appropriate components the performance can optimized, and the total power consumption can be significantly reduced. The wiring between the components can then be programmed prior to data processing [5, 9, 19]. Recent developments allow the configuration of address generators that implement nested loops in a few instructions, reducing the configuration time of CGRAs [11]. Instead of using a single static configuration for each program, CGRAs can now be reconfigured multiple times during program execution, *i.e.* dynamic reconfiguration [21]. However, since reconfiguration consumes execution time, a balance must be met in order to achieve good performance [35].

2.2 Versat

Versat is a CGRA architecture where each piece of the program being executed can use a different composition of functional units [10]. It is used as a hardware accelerator for embedded systems. It uses a controller to generate and update the configuration, the *picoVersat*. The controller is programmable and executes programs written in **C** and assembly (specific to the architecture). Contrary to most CGRAs, which can only be fully reconfigured, *Versat* allows for partial reconfiguration (where only few configura-

tion bits are changed), thus optimizing configuration changes. Moreover, in this approach the configurations are self-generated. Consequently, the host does not have to manage the reconfiguration process and is free to perform more useful tasks [32]. *Versat* uses an address generation scheme that is able to support nested loops expressed in a single CGRA configuration. Due to the better silicon area utilization and power efficiency of heterogeneous CGRAs architectures, when comparing to homogeneous ones, an heterogeneous structure was adopted for *Versat*.

2.2.1 Architecture

Versat can be used by host processors in the same chip (allows for procedures to run faster and with less power consumption), and was designed for fixed-point signal processing. Its architecture itself is composed of five different components and is shown in figure 2.1. These components are a Controller, a Control Register File (CRF), the Data Engine (DE), the Configuration Module (CM), and the DMA.

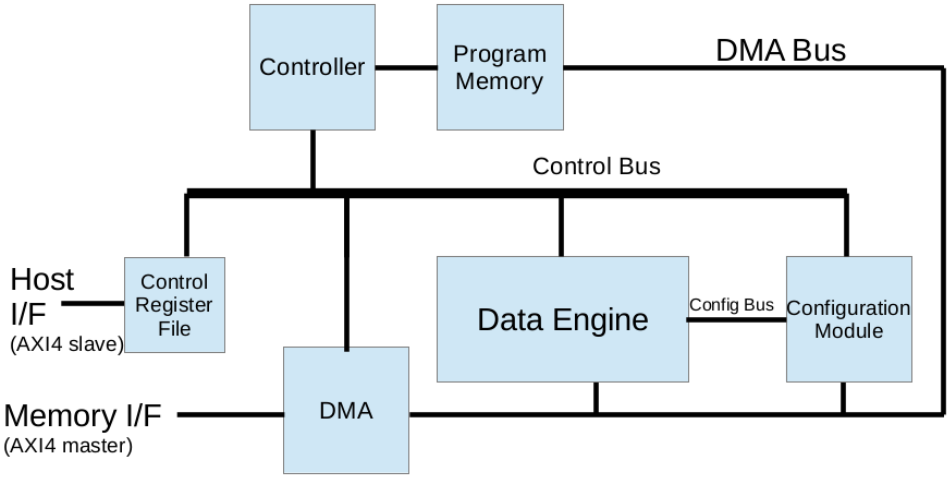


Figure 2.1: *Versat* top-level architecture.

The Controller is a programmable component of the architecture that has a direct dependency with the host that is controlling it, and is in charge of the data flow, algorithm calculation, and control of the self configuration. The control data is passed to the other sectors of *Versat* through a Control Bus like for example the attached Serial Divider. The Controller that is used is called *picoVersat*. The DE is the module that does the computations and is composed of interconnected Functional Units (FUs). The CM has the configurations of the datapaths that are meant to be executed by the DE. It also allows for partial reconfiguration and for the change of configuration in runtime. The DMA is a module that works in parallel with *Versat* and has the function of transferring the configurations, programs and data needed for the execution to and from *Versat*.

Data Engine (DE)

The DE has a fixed topology comprised of 15 Functional Units (FUs) organized in a mesh as shown in figure 2.2.

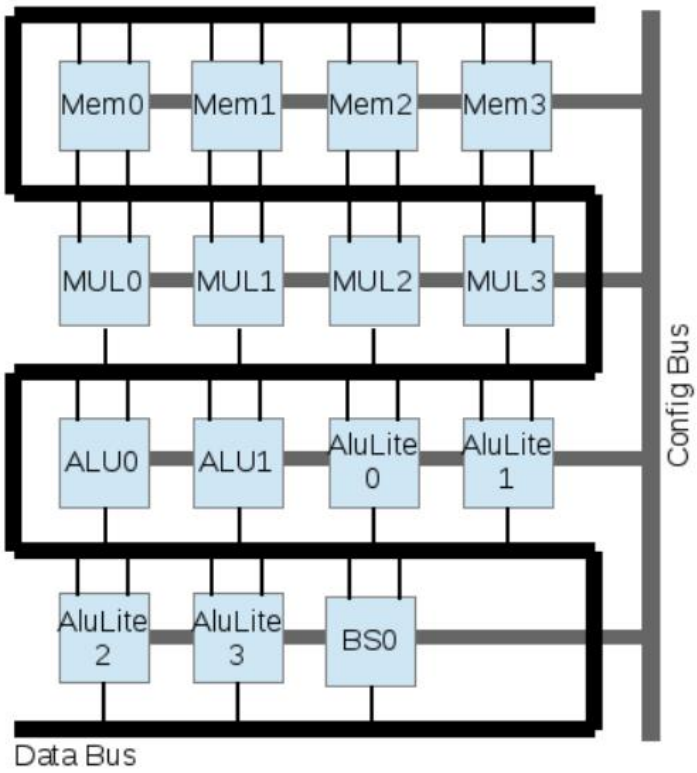


Figure 2.2: Data Engine Structure.

The DE is a 32-bit architecture containing 15 FUs which consist of one barrel shifter, six ALUs, four multipliers and four dual-port 8kB embedded memories. The system's Controller reads and writes from the FUs and to the memories.

Each FU reads it's configuration of an operation and an input selection from the Configuration Bus. All FUs write their 32 bit output to a wide Data Bus of 19x32 bits. The operations are chosen from a fixed number of those that are available to the accelerator.

The DE can be configured using one or more hardware datapaths, which allows for Data Level Parallelism (DLP) or Instruction level Parallelism (ILP) depending on if the execution happens in parallel lanes or if the paths are pipelined, respectively. Datapaths can operate in parallel in this architecture as long as the resources needed to execute each of the datapaths are available, which also gives *Versat* the capability of having Thread Level Parallelism (TLP).

Some of the most used cases of parallelism can be described by the following examples of hardware datapaths. In figure 2.3 there are shown three examples of datapaths describing these cases of parallelism.

Because of the nature of the architecture, pipelined vector addition, the memory read and write operation, and the addition operation, can be done in parallel, for consecutive elements of that vector, as shown in (a). It is possible to do multiple pipelined vector addition operation using data from different memories and saving the results in the same or different memories in parallel, if the data itself is not dependent on each other, like in the case of (b). When there is a need to do the inner product of two vectors, since in this case elements from each vector are multiplied, then the results are added and

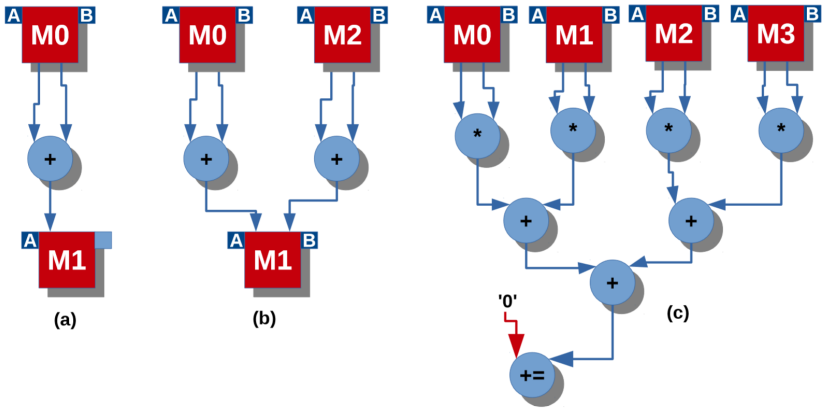


Figure 2.3: Data Engine Structure configuration examples with dual-port (A and B), memories (M0 to M3), and functional units (adders and multipliers).

accumulated, which is shown in datapath (c). All of the ALU operations described can be executed in parallel [32].

There is an independent Address Generation Unit (AGU) for each port of the memories used, which allow for two level nested loops, and execution start with a programmable delay. TLP can be exploited because the AGUs can operate independently, making it possible, for example, to operate through data in different memories using two threads and then save the values obtained in the same goal memory.

Configuration Module

The Configuration Module (CM) is composed of three elements which are the Configuration Register File (CRF), the Configuration Shadow Register (CSR) and the Configuration Memory (CM), as shown in figure 2.4.

The preparation of the next configuration to be executed by the DE is done by the CRF. It is an element composed of 118 configuration fields, and each of these differ on the number of bits they define, totaling a full configuration of 672 bits. By having each of these fields separate, it is possible to take advantage of the fact that, during the execution of most of the application code, there is a high chance that the DE configurations used are very similar to each other. A partial reconfiguration can be done more effectively since they differ in a low number of fields.

During execution, the current DE configuration is present in the CSR. This configuration is updated by copying the configuration in the CRF to the CRS when an *Update Signal* is sent by the Controller. This allows for the reconfiguration of the system using the CRF while the configuration in the CSR is running (hidden reconfiguration in runtime).

Because some DE configurations are very common there is a dedicated memory for the most common configurations. Each configuration has 672 bits this in a dual-port 64 position memory. The configurations saved in this memory can be loaded/stored from/to the CRF in a single clock cycle through one of the memory ports while the other port is 32-bits wide and is used to load/store in an external memory by using the DMA, so that the CM is able to exceed 64 positions.

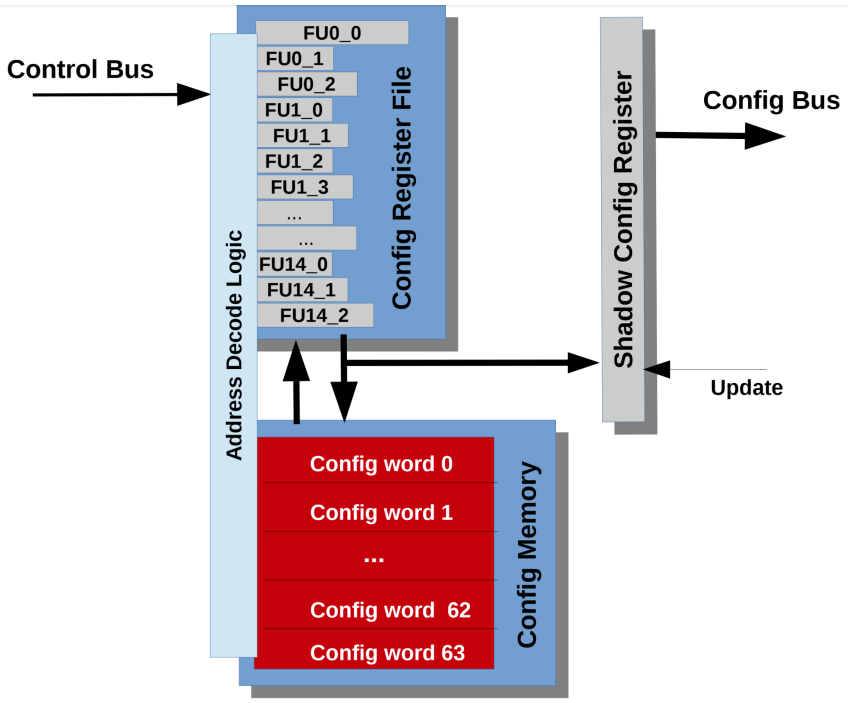


Figure 2.4: Configuration Module Structure.

Controller

picoVersat is a minimal hardware controller with a reduced instruction set and few registers, with the purpose of doing simple calculations and giving instructions to its connected systems. Therefore this controller is not designed for high performance computation, even if it is able to effectively implement simple algorithms. It is a programmable solution which mitigates the risk of hardware design mistakes, and helps the design and implementation of more complex control structures. This controller is simple and has a reduced silicon area which allows for a low power consumption.

2.2.2 picoVersat (Controller)

The controller's architecture can be represented by the block diagram shown in figure 2.5.

picoVersat is composed of four main registers:

- Register **RA** is the accumulator and is the most relevant of the registers in the architecture. It can be loaded with a value read from the data interface or with an immediate value from an instruction. It gets the value from the result of the operations, and is used as an operand itself along with an immediate value or an addressed value. The value of **RA** is sent to the data interface.
- Register **RB**, the data pointer, is used to implement indirect loads/stores to/from the accumulator. This register is mapped in memory which means it is accessed by reading/writing as if accessing the data interface.
- Register **RC** is the flag register, storing three operation flags which are the negative, overflow, and carry flags. Much like register **RB**, this register is also memory mapped but this one is mapped to

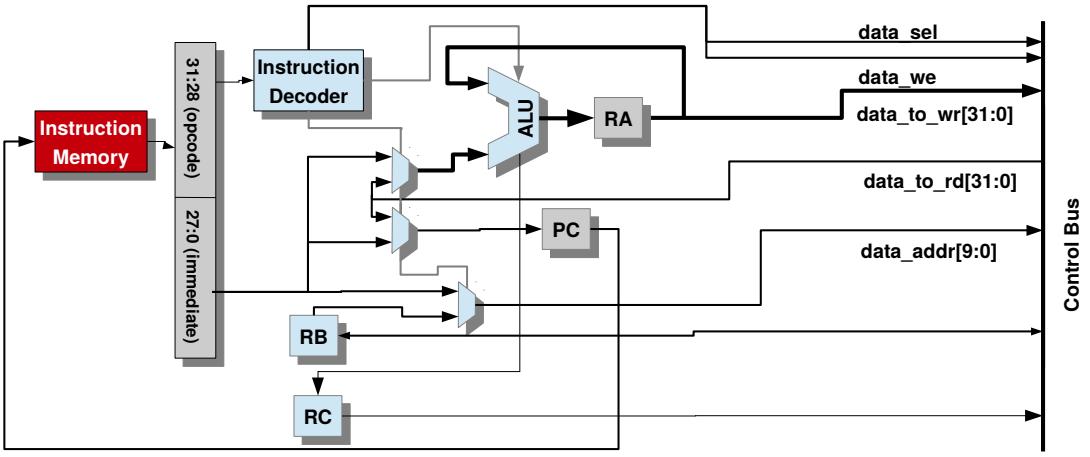


Figure 2.5: picoVersat Block Diagram: with registers (RA, RB, RC and PC), their interconnections and memory access.

a *read only* section, being only set by the ALU. The structure used for this register is depicted in Table 2.1. As it is shown in Table 2.1 some bits are not used and could be used to implement other flags but, as of right now, there has not been a need to add any more.

Bits	Name	Description
31-3	NA	Reserved for future use
2	Negative	Asserted if last ALU operation generated a negative result
1	Overflow	Asserted if last ALU operation generated an arithmetic overflow
0	Carry	Asserted if last ALU operation generated a carry

Table 2.1: Register C: flags

- The Program Counter, **PC** register, contains the next instruction to be fetched from the Program Memory so that it can be executed. This register increments for every instruction, in order to fetch the next instruction except for branch, in which the **PC** register is loaded either with an immediate present in the branch instruction, or with the value present in **RB**. These two types of branches implement direct and indirect branches, respectively.

For increasing the frequency, by reducing the critical path, the controller takes two clock cycles to fetch one instruction, in pipeline. It executes every instruction that is fetched. In other words, it has 2 delay slots in the case of a branch instruction. These delay slots can be filled with a no-operation instruction (NOP), but the compiler/programmer can also use them to execute useful instructions. For instance, in the case of a `for` loop, the delay slots can be used to write the iteration count to some register [32].

Instruction Set

The instruction set that is used to run *picoVersat* is minimalist and has only one type. This type has 32 bits and is divided between an *opcode* and an immediate constant, like it is shown in Table 2.2.

Bits	Description
31-28	Operation code (opcode)
27-0	Immediate constant

Table 2.2: Instruction format.

The operations that the controller can perform are listed in Table 2.3. The notation used for the logic, that represents the operations, is written with **C** language syntax in mind. It is also notable that *Imm* represents an immediate constant and that *M*[*Imm*] represents the contents in address *Imm* in memory.

Mnemonic	Opcode	Description
Arithmetic / Logic		
addi	0x0	RA = RA + Imm; PC=PC+1
add	0x1	RA = RA + M[Imm]; PC=PC+1
sub	0x2	RA = RA - M[Imm]; PC=PC+1
shft	0x3	RA = (Imm $<$ 0) ? RA $<<$ 1: RA $>>$ 1; PC=PC+1
and	0x4	RA = RA & M[Imm]; PC=PC+1
xor	0x5	RA = RA $\oplus$ M[Imm]; PC=PC+1
Load / Store		
ldi	0x6	RA = Imm; PC=PC+1
ldih	0x7	RA[31:16] = Imm; PC=PC+1
rdw	0x8	RA = M[Imm]; PC = PC+1
wrw	0x9	M[Imm] = RA; PC = PC+1
rdwb	0xA	RA = M[RB+Imm]; PC = PC+1
wrwb	0xB	M[RB+Imm] = RA; PC = PC+1
Branch		
beqi	0xC	RA == 0 ? PC = Imm: PC += 1; RA = RA-1
beq	0xD	RA == 0 ? PC = M[RB]: PC += 1; RA = RA-1
bneqi	0xE	RA != 0 ? PC = Imm: PC += 1; RA = RA-1
bneq	0xF	RA != 0 ? PC = M[RB]: PC += 1; RA = RA-1

Table 2.3: Instruction set.

Additionally, the pseudo instruction `nop` can be used in assembly and is converted by the assembler. It means No-OPeration or do nothing and is coded as `addi 0`. The instruction following a branch instruction is always executed due to the instruction pipeline latency (delayed branch or slot). Pad with `nop`s if no useful instructions can be executed.

This controller has the capability to work on its own, that is, the controller can run the functions and operations that do not depend on the *Versat* accelerator itself without it being connected, which allows the execution of simple applications that do not have tight time constrains.

2.2.3 Previous Compiler

The previous compiler [41] was made from scratch and was based around object oriented programming languages like *JAVA* and *C++*. Although, it claims to be a **C++** dialect, it provides no classes, not even **C** structures, variables or declarations, and no object-oriented capabilities like encapsulation or polymorphism. The use of a 'dot' to access object functions (*obj.func()*) of predefined objects extends the use of **C** alike structure addressing to a small set of compiler static dependent objects.

The compiler was developed around the idea that each component of *Versat* is viewed as an object.

Each function that each object can perform is called by referencing the function within the object. The programming is performed by calling operations on those predefined objects. This was a good idea but, since a true object oriented approach is far too complex for the architecture of a CGRA, the result ended up looking like an object based language but not working like one. Instead of using a classic approach where an abstract syntax tree (AST) is built from syntactic description, the compiler only uses components to represent the **Versat** hardware components.

Before developing the previous compiler existing common compilers were investigated, like **gcc** or **llvm**, but it was "concluded that these compilers are good at producing sequences of instructions but not sequences of hardware datapaths" [32, p. 21].

Another drawback is that this compiler does not allow for the coding of some functionalities that the CGRA itself has, and do not have an implemented optimization step in the *back-end* of the compilation process.

The above problems and the aims described in the Objectives section is the reason why the previous compiler needs to be improved, as is the purpose of this work. In order to achieve the objectives there were two options: continue evolving the previous compiler by adding new functionalities, or to create a new compiler through a different method that would be more adapted to the type of architecture.

The first option, even though it seems to be the easiest, suffers from the problem that it took as a base an object oriented architecture but does not work like one. To circumvent the previous approach as well as the other problems described earlier, a research was made in order to see if the other option was better or worst.

The main advantage of the second option of creating a compiler not from scratch is that it allows for the user code to be written exactly in the **C** programming language. This is a very good advantage, not only because of the extra help given to the programmer, but also because it would allow to make the optimization simpler. Another good point to defend this approach is that the controller itself can work on it's own, without the *Versat* accelerator, being able to run any **C** code, even if much slower that with the accelerator connected.

2.3 Compilers

Since using the previous compiler as a starting point is not a good option, existing compilers should be investigated. First, the components of a compiler must be analyzed. Then, taking into account the usage of compiler components by existing compilers, a selection of widespread compilers is analyzed. Finally, a comparative analysis of the most common compilers is performed in order to select candidates suitable for conversion. The converted compiler should generate *picoVersat* assembly, identify and configure the *Versat* CGRA architecture.

Computer programming, and processor programming in particular, is a complex and time consuming task. Computers execute binary code, but writing binary code is an error prone and meticulous task reserved for very small low level programming. Assembly provided a textual approach to computer programming, while keeping full control over the instruction set for a particular processor.

A compiler is a tool that allows the programmer to write abstract high-level instructions and produces assembly code for the processor [1, 8]. The assembler tool can then be used to transform the assembly instructions into binary instructions of the processor. The compiler can also perform semantic error detection and data flow analysis, thus improving the required development cycle and the generated code quality. A good compiler can produce optimized code as good, or sometimes better, than the manually written assembly equivalent.

In order to transform a high-level programming language into assembly code, the compiler tool is composed of two major phases: an analysis phase and a synthesis phase. During the analysis phase the input file is read, byte by byte, and structured into an abstract syntax tree (AST) containing all the relevant information for final code generation. The synthesis phase transforms an AST into machine code assembly instructions [8, 2]. These phases correspond to, what is called in compiler terminology, the *front-end* and the *back-end*, respectively.

2.3.1 Analysis

The analysis phase is decomposed into three analysis stages: lexical, syntactical and semantic. The stages are pipelined in such a way that the output of a lexical analyzer is fed into a syntactic analyzer, and the output of the syntactic analyzer is coupled with the semantic analyzer. At the end of the syntactic analysis, the language in the input file is converted into an AST.

Lexical analysis

The lexical analysis processes the input file, containing the high-level computer program, into identifiable *tokens* [42, 4]. The *tokens* are symbols or words that play a specific role in the input language. For instance, in the **C** programming language, braces and semi-columns, among others, are of particular interest [24]. Other *tokens* may include identifiers, operators and language reserved keywords such as data types or flow control. The lexical analysis also removes information that is irrelevant for code compilation, including comments and white-spaces.

The lexical analysis can be performed using simple and efficient algorithms based on regular expressions. However, as it is tedious, difficult to amend and error prone work, lexical generator can be used to automate and simplify the lexical generator. The lexical analyzer generator **lex** [29] and its successors, like **flex** or **jflex**, can produce very compact and efficient lexical analyzers from a simple description based on regular expressions and embedded **C** or **C++** code.

Syntactic analysis

The syntactic analysis phase structures a sequence of *tokens* provided by lexical analysis into a tree of tokens [42, 4]. The purpose of the syntactic analysis is to check the order of the *tokens* and identify the language constructs that they aim to describe. After the syntactic analysis the structure of the program is well defined and constructs like expressions, instructions, functions and variable declarations have been identified.

The syntactic analysis, as the lexical analysis, can be greatly simplified using efficient algorithms. However, these algorithms are more complex and time consuming than the ones used in the lexical analysis. These algorithms are grouped into two different groups, **LL** and **LR**, depending whether the constructs are identified **L**eft-to-right or **R**ight-to-left (the second letter in the acronym, since the first reveals the *token* arriving order which is always **L**eft-to-right). The syntactic analysis is described by a context-free grammar describing the constructs of the high-level language being processed. Depending on the specific algorithm used to parse the grammar, a different but equivalent grammar must be found. The **LALR(1)** algorithm, which stands for **L**ook-**A**head **LR** with a single lookahead *token*, allows a wide range of equivalent grammars, making the discovery of a suitable grammar for the high-level language a simple task.

As for the lexical analysis, also the syntactic analyzer can be generated using an automated tool from a grammar description with **C** or **C++** code insert using tools like **yacc** [23] or **bison** [13]. The code insert are used to build the **AST**. Frequently, each grammar rule produces a tree node in the AST. Tools like **antlr**, a **SLL(*)** (a **S**trong **LL** with arbitrary lookahead **(*)** parser generator), provide mechanisms to automate the construction of the AST [36].

Semantic analysis

The semantic analysis phase is responsible for checking if the constructs, identified by the syntactic grammar, are accepted by the high-level language and can be performed by the computer. It must check if a variable was already declared or if the operation request can be performed by the variable. The semantic analysis results from the fact that grammar must be context-free in order to be processed by efficient syntactic algorithms.

The semantic analyzer scans the **AST** for inconsistencies and prevents code from being generated for erroneous programs. Since semantic check is very dependent on the high-level language particulars, it is usually performed by the host language (**C** or **C++** in most cases) after, or even during, the syntactic analysis.

2.3.2 Synthesis

The synthesis phase is no longer dependent on the input high-level language. Many compilers produce the AST from different high-level languages, although some languages may not generate some type of nodes. On the other hand, many compilers produce code for different processors and computer architectures from the same **AST** [20].

The synthesis phase is comprised of several stages: instruction selection, instruction scheduling, register allocation, optimization, and data flow analysis. Code synthesis is an NP-complete problem and a particular order of stages does not necessarily produce the best code, since a good decision in one stage can compromise the quality of subsequent stages. Some compilers repeat some stages or try different approaches in order to find the best among them.

Instruction selection and scheduling

Most compilers begin the synthesis phase by the instruction selection. Instruction selection aims at finding the best processor instructions for a part of an AST [8, 18, 38]. For instance, an **arm** processor offers a free sum (`add`) for each multiplication (`mul`) and free shift for most arithmetic instructions. When these pairs of instructions are found in contiguous tree nodes, a single instruction can be produced, otherwise each node produces a processor instruction. An instruction selector generator is a tool based on a grammar that describes the tree pattern that matches a given processor instruction. In fact, the grammar describes the target processor capabilities in terms of tree patterns. To each tree pattern can be assigned a cost, that measures the time consumed by the instruction execution. The selector generator chooses the best combination of instructions that map all the AST, that is where total cost is minimum. As a result, the instruction selection produces a sequence of target processor instructions. The arguments of the instruction, that is its registers, are not yet assigned. Instruction selection can be performed by dynamic programming where the AST is searched twice, one for determining the instruction costs and a second to generate the best selected instructions. Therefor, the algorithm is linear on the number of instructions (O(N)). The instruction selection tools, known as BURG (**B**ottom **U**p **R**ewrite **G**rammar), like `lburg`, `iburg`, `monoburg` automate the code generation process.

C3E **versus** SSA When representing abstract processor instructions there are two representations: *three address instructions* (C3E) and *static single assignment* (SSA) [2, 37]. In C3E each instruction takes up to three arguments, one of them being the destination of the operation. These instructions arguments are not yet registers, since register allocation was not performed, but names of high-level variables. More recently, since `GCC-4` and in other modern compilers, the SSA has been adopted because it provides a better base for optimization, allocation and scheduling. In SSA each argument is tagged with a version number and, each time the variable is reassigned, the version number is incremented. Therefor, two registers can contain different versions of the same variable, allowing for a better use of common sub-expressions. For instance, in `while (*s++)` both s_0 and s_1 from $s_1 = s_0 + 1$, the old value is kept in a register for the evaluation of the condition while the new value is stored in memory.

Basic blocks A basic block is a sequence of instructions that have no labels between them, allowing transfer of control into the block, nor have jump instructions among them. A jump instruction always ends a basic block. A label always breaks a block in two basic blocks. Within a basic block the sequence of instructions can be changed, as long as the argument dependencies are kept.

Instruction scheduling The scheduling is performed on code basic blocks. The scheduling rearranges the order of the instructions taking into account the number of available functional units in the processor and the latency of their executions. To each instruction is given an available functional unit, and a simulation of the execution is performed to determine when the functional unit is again available for the next instruction. For instance, when a multiplication is being performed, other operations like sums and shifts can be done in parallel as long as there is no dependency on the arguments. The rearrange-

ment of the instructions produces the minimum sum of elapsed time, including latencies. List scheduling is a greedy heuristic approach that performs a simulation of the execution of a basic block instructions once (O(N)), but it is quadratic on the number of operands (O(m^2)), although for most operations m is one or two [8, p.605]. For instance, when an operation requires arguments that result from a sum and a $multiplication$, the $multiplication$ is performed first since it has higher latency, and the sum is executed while the $multiplication$ is being performed if both operations can be paralelized.

Register allocation

Register allocation aims at finding the best use of the available machine registers. On machines with a small number of registers, the process is simple since there are not that many options. Modern RISC machines register allocation, with 16 or more general purpose registers, is both complex and central to the efficiency of the resulting code. Register allocation algorithms are divided in local, within basic blocks, and global, when involving more than one basic block.

When performing local register allocation the available registers can be freely assigned and an optimal algorithm exist, for instance when generating code for expressions. Global register allocation usually sets aside some of the registers and tries to distribute their use among the basic blocks, taking into account their dependencies (see Optimization and data flow analysis). When one basic block uses more registers, another must be sacrificed, and no optimal solutions exist. However, if instructions are limited to **if**, **do**, **while** and **for** without the indiscriminate use of **goto**, a good solution can be obtained.

Register allocation is based on `live` or `dirty` registers, those that presently contain a useful value, and `dead` or `clean` registers, those that can be assigned to new uses. However, `live` registers can contain a copy of memory value, allowing for its use without saving, or computed values that must first be saved before the register can be used for another purpose. It is up to the allocation algorithm to find the best combination of registers in order to reduce the number of memory `loads` and `stores`.

Register allocation algorithms include local algorithms like *Sethi-Ullman* (optimal for expressions), *greedy*, *next-use* and *top-down* or *bottom-up* approaches. Global allocation uses *linear scan* for simple `just-in-time` compilers and more complex *graph coloring* for optimizing compilers.

Optimization and data flow analysis

Once the code is generated it can be rearranged. In fact, optimization begins with instruction selection and is present in each stage of the synthesis phase [39, 34]. Simple optimizations, like *peephole* where a sliding window analyses sequences of adjacent instructions, can be performed after instruction selection when two nodes produce contiguous instructions that can be replaced by a better one. Data flow analysis is an optimization that searches for the best sequencing of basic blocks, reducing the number of inter-block jumps. Complex compilers may provide thousands of optimizations that must be tested in order to find out if an improvement can be achieved.

2.3.3 B language compiler

The **B** programming language is the predecessor of the **C** programming language and provides a very similar syntax [3]. **B** uses only an integer data type, eliminating the need for data structures. It provides access to single byte characters through literals and library manipulation routines (see Figure 2.6). All **C** operators and instructions are present but lacks the **for** construct. Missing is also the preprocessor.

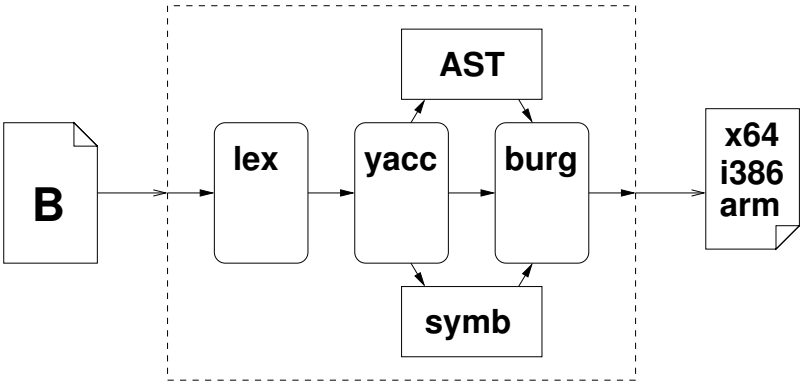


Figure 2.6: **B** language compiler.

The **B** language compiler (**blc**) is very small, under one thousand lines of **C** code with grammars for the **flex**, **bison** and **burg** compiler generators. New constructs can be easily added to the language, but it will no longer be **B**. There is complete access to all parts of the compiler and `asm` support is provided.

2.3.4 gcc

gcc was originally created by Richard Stallman in 1987 (`0.9 beta`, March 22) as free **C** compiler [43]. As of version `1.27` (September 5, 1988) was a stable usable compiler. In 2018 there were 4 major software releases from 3 development branches. It includes, up to now, 494 contributors [44]. The compressed (.tar.gz) source code is 80MB (8.2.0), 535MB uncompressed with over 14 million lines of code[1]. **gcc** includes compilers for **Ada**, **C**, **C++**, **Fortran**, **Java** and **Objective C**. It includes code generation for most 32-bit and 64-bit main stream commercial processors, including **x86**, **x64** and **arm**.

gcc provides three types of intermediate languages: generic, gimple and rtl (see Figure 2.7). Generic is 'language-independent way of representing an entire function in trees', essentially a complex AST. Gimple is a **C3E** representation, where some operations can have more than 3 arguments, in a **C++** inheritance hierarchy and a 29 instruction set. The `rtl` (Register Transfer Language) is a low-level intermediate representation in a **lisp** alike form. Target machine descriptions can be provided through a **burg** alike file of pattern descriptions.

There is a book describing the compiler internals for versions 3.4 and 4.1 [43], although in 2018 distributions are for versions 7, 8 and upcoming 9 (accessible in SVN) [44]. The compressed (.tar.gz) source code is 80MB (8.2.0), 535MB. The document for the upcoming release 9 is essentially a listing of functions with a single line description for each function.

[1] https://gcc.gnu.org/releases.html

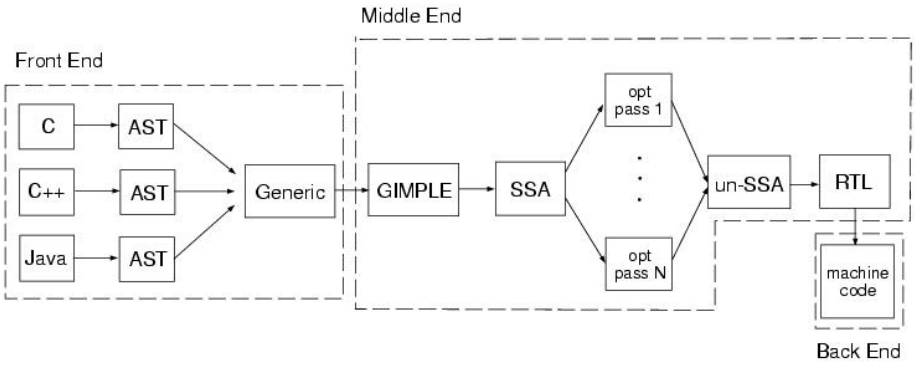


Figure 2.7: **Gcc** compiler.

The best, with thousands of generic and architecture specific optimizations, but it is a monster! Very big, lots of people change it and maintain it, making the code very volatile, since the user must keep up to constant changes in the code. The purpose is not code optimization since *Versat* does not provide alternative instructions for its operations, only *picoVersat* could benefit of **gcc** features. It provides `asm` support and many other **gcc** specific **C** language extensions.

2.3.5 llvm

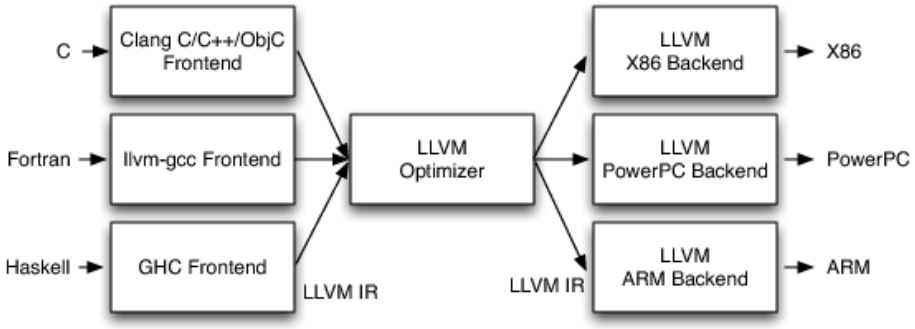


Figure 2.8: **LLVM** compiler.

LLVM (*Low Level Virtual Machine*) was born as an infrastructure for optimization from the masters thesis of Chris Lattner at the University of Illinois at Urbana-Champaign [26, 27]. Although an effort was made to integrate it into **gcc** it evolved into a full compiler with many *front-ends*. The first version was published Oct 24, 2003 with four releases in 2018. Many contributions for the compiler tool-chain were made over the year, resulting in over 250 publications (`http://llvm.org/pubs/`).

Nowadays, the code is extensive. Only the core is 27MB compressed, resulting in almost 7 million lines of code occupying 296 MB (7.0.0)[2]. The **C** language compiler (**clang**) is roughly half the size. Other *front-ends* include **C++**, **Objective C** and **Fortran** among others. It provides *back-ends* for many processors: **X86**, **X86-64**, **PowerPC**, **PowerPC-64**, **ARM**, **Thumb**, **SPARC**, **Alpha**, **CellSPU**, **MIPS**, **MSP430**, **SystemZ**, and **XCore** (see Figure 2.8). A new target description is provided by a declarative

[2]http://releases.llvm.org/

domain-specific language processed by the **tblgen** tool, similar to **burg** grammar descriptions. It features a stable internal instruction set implementation and documentation.

The compiler supports `asm` directive extension and checks the correctness of the embedded assembler itself.

It is still very large and difficult to manipulate. The interface is complex, written in **C++**, but stable. Requires a big effort for a single person for a few months.

2.3.6 lcc

The **lcc** is a **C** compiler developed after the publication of the **BURG** papers by Proebstring, Hanson and Fraser in 1992, as a demonstration of the concept [18, 17, 38]. Up to that time *back-end* compiler writing was an art. The **burg** concept allowed to describe the target architectures in a single file as well as maintaining all of them in a single executable, a retargetable compiler. The target architecture is selectable by a command line option.

Only the **C** *front-end* is supported with a custom made lexical analyzer and a **yacc** parser (see Figure 2.9). The language is described in an AST with a well documented 32 instruction set. Optimizations are performed in the AST, and through common sub-expression analysis, the tree is converted into a **DAG** (*Directed-Acyclic Graph*), although the original tree is still accessible.

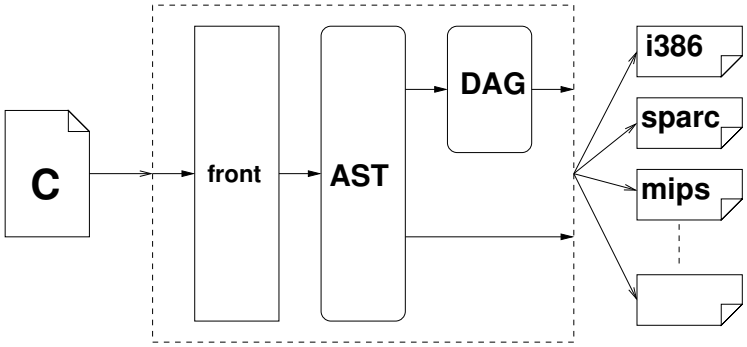


Figure 2.9: **lcc** compiler.

A single **C** *front-end* reduces the complexity and line count. Simple to introduce new *back-end* since it was designed to be retargetable. A detailed and extensive book and well defined *back-end* interface provide a good work basis [20]. It provides no `asm` directive and it's introduction will require changes to core of the *front-end* code. However, the compiler core is composed of 260K lines of code and any changes should be accessible [3].

2.3.7 tcc

The **tcc** (*Tiny C Compiler*) is a small and fast **C** compiler written by Fabrice Bellard that began as the winner of a 2001 contest to provide the smallest self-compiling language (OTCC)[4]. Now it is a full ANSI

[3] https://github.com/drh/lcc
[4] https://bellard.org/otcc/

C compiler [48] in 620KB of compressed code, totaling 105 thousand lines of code (1.7MB) [5]. It also claims to be 9 times faster than **gcc**. It only supports the ANSI **C** *front-end* with *back-ends* for **x86**, **x64** and **arm** architectures. The **tcc** `-run` option allows the code to be compiled directly into memory, and executed without producing output files since it includes an assembler and linker (*self-relying*).

On the other hand, the compiler provides few optimizations and only simple register allocation, but the output is quite efficient for most non production applications. Documentation is limited but the code is manageable, however no `asm` support is provided. Also, the compiler produces binary code directly, not output assembler code, and it might not be easy to add assembler support altogether.

2.3.8 Portable C Compiler

The **pcc** was developed in the 1970s by Stephen Johnson of the Bell Labs [22]. It was one of the first compilers that could target different architectures. **pcc** was the BSD UNIX **C** compiler until it was replaced by **gcc**. A 2007 rewriting ported it to C99 standard. In the present form, **pcc** was released in 2011 and supports **x86** and **x64** architectures. It provides `asm` support, but relies on rather old technology with no instruction selection tool, such as **burg**. On the other hand it comprises only 160K lines of code [6].

2.3.9 Amsterdam Compiler Kit

The Amsterdam Compiler Kit (ACK) [45], developed in early 1980s, was one of the first retargetable compilers with *front-ends* for **C**, **Pascal**, **Modula-2**, **Occam**, and **BASIC** as well as *back-ends* for many processors and micro-processors. The technology used is outdated but simple and no `asm` is provided [7].

2.3.10 Small device C compiler

The **sdcc** is an ANSI **C** retargetable compiler [14], written by Sandeep Dutta. It targets many microprocessors but no major processors, although it can be retargeted for other microprocessors. The list of targets is the following.

Intel MCS51 based microprocessors (8031, 8032, 8051, 8052, etc.), Maxim (formerly Dallas) DS80C390 variants, Freescale (formerly Motorola) HC08 based (hc08, s08), Zilog Z80 based MCUs (z80, z180, gbz80, Rabbit 2000/3000, Rabbit 3000A, TLCS-90), and STMICROELECTRONICS STM8. Work on the Microchip PIC16 and PIC18 is under development.

It does not support `asm` directives and the code, although large (7M lines of code), is essentially dedicated to each specific processor [8].

[5] https://bellard.org/tcc/
[6] http://pcc.ludd.ltu.se/
[7] http://tack.sourceforge.net/
[8] http://sdcc.sourceforge.net/

2.4 CGRA Compilers

Unlike the previous general purpose **C** language compilers, CGRA compilers directly address the problem of configuration and reconfiguration of a CGRA platform.

Some CGRAs, like ADRES, Silicon Hive, and MorphoSys are fully dynamically reconfigurable: exactly one full reconfiguration takes place for every execution cycle [12]. Other CGRAs like the KressArray are fully statically reconfigurable, meaning that the CGRA is configured before a loop is entered, and no reconfiguration takes place during the loop at all [12]. Still other architectures feature a hybrid reconfigurability. The RaPiD architecture features partial dynamic reconfigurability, in which part of the bits are statically reconfigurable and another part is dynamically reconfigurable and controlled by a small sequencer [12]. Most compiler research has been done to generate static schedules for CGRAs [12]. Not enough details are available in the descriptions of the compiling techniques, and few techniques have been tried on a wide range of CGRA architectures [12]. For that reason, it is very difficult to compare the efficiency, effectiveness and user interface of the different techniques, this is, compilation time, quality of the generated code, and how easy it is to use and to port code to the language used, respectively [47].

There are frameworks that try to compile code in usual languages to CGRAs. Most of these frameworks have however some limitations that do not allow them to adapt the code fully to CGRAs' purposes. For example, the IMPACT compiler framework is used in order to parse **C** source code and get the Intermediate Representation (IR) [33]. We can then change what the *back-end* does to this IR, to get the desired result. However, because of the IMPACT *front-end*, most algorithms can only handle inner loops of loop nests [33]. And since changing the *front-end* as well as the *back-end* is the same as creating a new compiler, most of these frameworks have problems when there is a need to really adapt the code not made in the specific assembly language of the CGRA supposed to run the code.

Nowadays most of the algorithms used for CGRAs are adaptations of the ones used for FPGAs but, since the structure is not the same, this approach is not the most efficient and/or effective [7].

2.4.1 Comparative Analysis

Compilers are complex tools. However, if the target processor is simple and the high-level language is limited, a simple compiler can be developed in a few months, using the right tools. For large and complex languages like **C**, with a large population of skilled programmers, a number of freely available compilers exist. The development of a compiler, even with powerful tools, is a very complex and time consuming task, that usually involves tens of persons. For large and complex languages like **C** the use of an existing infrastructure is of prime importance.

The **B** programming language is a good choice if changes to the *front-end* of the compiler are necessary, since changing the *front-end* as well as the *back-end* is essentially writing a completely new compiler.

More complex languages, like **C++**, are an overkill for a processor with limited memory like *Versat* since with its memory limitations it will not be able to run large object-oriented programs.

Chapter 3

Architecture

The development of a high-level language compiler, with variable and function support, is the main objective of this work. In order to achieve this goal, two approaches can be taken: the existing compiler can be extended, or a new compiler can be developed.

The existing compiler, although it provides valuable information, adopted many design decisions that make the required extensions very tedious and complex. Furthermore, there is no documentation and the only developer is no longer available to explain any doubts that might arise, since only the source code is accessible. The alternative, the development of a new compiler, can be start from scratch or use an existing compiler.

The development of a new compiler for a rather large and complex language like **C** is a daunting task and would require human resources beyond the scope of this thesis. Since several retargetable **C** compilers exist, where the *back-end* can be reconfigured to a new processor, a selection must be carried out. These options also imply that the *front-end* remains unchanged and that the full **C** language is implicitly supported. Consequently, all existing **C** programs that, when compiled, will fit the *Versat* memory can be used. Also, the development of new software can be made without any specific language or architecture common knowledge other than the standard **C** programming language.

3.1 Compiler selection

After analyzing the previous options it was decided that the best approach was to take an existing compiler and add the *picoVersat back-end* to it, making it a *Versat* CGRA compiler. Even using another compiler as a starting point some code from the existing compiler that interacts with *Versat* can be adapted to fit this model.

The work begins with the development of a simple compiler which is able to parse **C** code, and generate the assembly necessary to run the **C** program in the controller (*picoVersat*). Since the controller can technically work on its own without the *Versat* accelerator, the compiler should also be able to reflect this. So even with a reduced instruction set, and limited memory, simple applications have the capability to be run in this component.

From the compiler candidates studied in the previous chapter, the *lcc* compiler is the most promising approach. It offers an instruction selection tool that can simplify the conversion process, but no `asm` directive (not a **C** standard feature) is available. The first stage is to describe all standard **C** operations and functions using *picoVersat's* assembly. This will allow the code to be parsed from **C** code to the controller which will then go through the assembler to generate the machine code needed to run the desired program. The assembler is already working with the instructions described in Section 2.2.1 so no changes will need in order to generate the correct machine code.

3.2 picoVersat back-end

The first stage in the development process consists in constructing a *back-end* that produces *picoVersat* assembly code for the selected compiler. Since we are choosing a retargetable compiler, the introduction of a new *back-end* is not a complex task. At this stage, all *back-end* low-level operations must be mapped into *picoVersat* assembly. The *picoVersat* instruction set is composed of a small number of instructions (see section 2.2.1) when compared with general purpose processors, like the I386 or the ARM processors. Therefore, the mapping of the **C** language constructs into such a small instruction set is a complex task. Most constructs will surely need long sequences of *picoVersat* instructions. Furthermore, some of the *picoVersat* registers must be permanently assigned to compiler management tasks, such as a `stack-pointer` and a `frame-pointer`, while other register may require temporary allocation for tasks such as indirect load or save operations. On general purpose processors, the register set includes such registers (`esp` and `ebp` in I386, for instance) and complex load and save instructions that can combine up to 3 registers (`lea` in I386) allowing the addressing of field in vectors of structures.

A compiler like **lcc** must also be extended to support the `asm` directive, since it does not support it yet. The `asm` instruction allows direct control over *picoVersat* and enables the generation of specific code not accessible through the **C** compiler. Please note that such instruction, although useful for the tasks required of this compiler, is not a part of any **C** language standard. It started as **gcc** extension and has since been adopted by other compilers.This instruction is also useful to provide direct control over the *Versat* data engine.

Figure 3.1 highlights the **lcc** main blocks that need to be modified in order to obtain a workable **lcc** compiler for *picoVersat*.

3.3 Data Engine Incorporation

Even though *picoVersat* (Controller) can work on it's own for very simple problems or auxiliary calculations, it's main purpose is to manage *Versat*, that is, all of its components. In particular, the controller is meant to manage the Data Engine in order to take full advantage of the accelerator. This is the biggest challenge when developing a compiler for this type of architectures.

The code used to incorporate *Versat* support to the compiler needs to be compatible with the code used to compile *picoVersat*. Also, in an ideal scenario, the compiler should not need to be different for

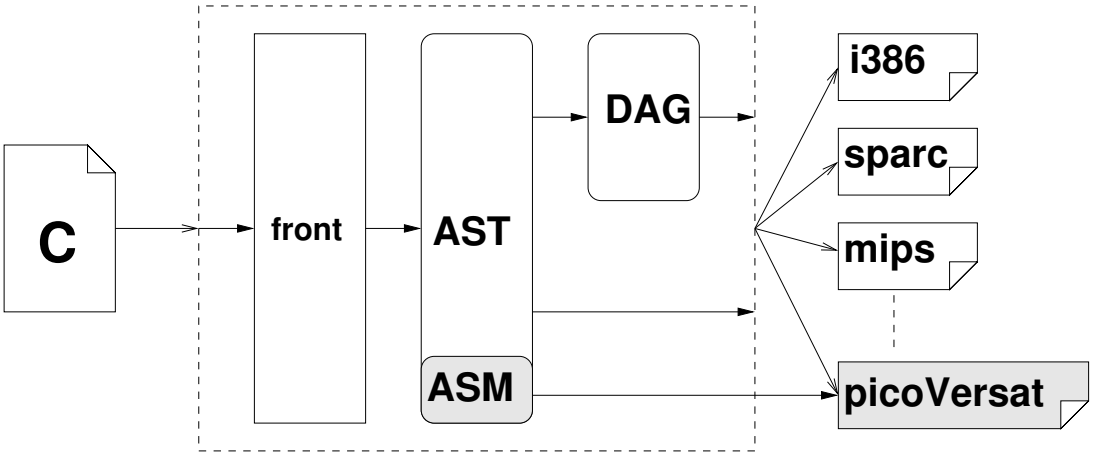


Figure 3.1: Changes in the **lcc** blocks.

compiling code with or without the accelerator's functions, since the controller can operate separately.

In order to make the control code for the *Versat* data engine compatible with a **C** language specification, instructions like `alu0.connectPortA(aluLite0)` must be converted into `connectPortA(alu, 0, aluLite, 0)`. The later notation can then be parsed by any **C** language compatible preprocessor. These instructions can be implemented as simple function call, for instance:

```
#define ALU_CONF_SELB_OFFSET 1
#define connectPortA(int a, in_a, int b, in_b) {
    asm("ldi %d\n", b[in_b]);
    asm("wrc %d, %d\n", a[in_a], ALU_CONF_SELB_OFFSET);
}
```

The top level *Versat* loop can then be invoked through a `versat()` function that drives the above configuration specific routines. This approach replaces the complex `for` instruction of the previous compiler, where some values, as well as the inner structure, must have a specific format instead of general purpose **C** language `for` instruction. The routine parameters correspond to the configuration parameters that define the operations to be performed in each ALU, the port connections, the cycle start value, delay, period, *etc.* Since there are a large number of parameters to be specified, auxiliary structures can be used to group related information. These structures can then be used to determine differences from the previous configuration uploaded to *Versat*, simplifying the partial configuration task.

The `versat()` function uses the parameters to determine the next configuration to be made. After checking the differences between the configuration currently stored in the Configuration Module and the one calculated, a final configuration is determined. This final configuration is made so that only the FUs that need to change their functionality are incrementally updated, decreasing reconfiguration time (partial reconfiguration).

The representation of the `versat()` function has all of the parameters necessary to get a new configuration as arguments. For the current compiler, these parameters are extracted by analyzing a specific `for()` loop model, for instance:

```
for(j=0;j<R6;j++) {
    for(i=0;i<R14;i++) {
        mem0B[R1+j*R13+i] = (mem1A[R1+j*R13+i] * mem2B[1025+j*R2+R10*i]) -
                            (mem0A[R1+j*R13+i] * mem2A[1024+j*R2+R10*i]);
    }
}
```

From this code block all the information needed in order to create a new data engine configuration is present. Therefor, the new formulation of these operations to be done in *Versat* must also have the same core information. For this purpose the function declaration is given by:

```
extern void versat(int N, int M, int* op, int* a, int* b, int* c, int* mem);
```

This is possible since the operations to be done in the loops can be described by a generic formula or combinations of it, for instance:

```
memory_x[a_x*i + b_x*j + c_x] = memory_y[a_y*i + b_y*j + c_y] operation
                                memory_z[a_z*i + b_z*j + c_z];
```

As a result, by having `a`, `b`, `c`, `op` and `mem` as vectors of parameters that complete the formulation of these loop operations, the `versat()` function can be invoked with the necessary specifications, for instance:

```
#include "versat.h"
int main() {
    int *a, *b, *c, *op, *mem;
    int N, M;
    /* ... */
    /* insert values into parameters to get desired operations */
    versat(N, M, a, b, c, op, mem);
    /* ... */
    return 0;
}
```

Alternatively, the structure containing all parameters, defined in `versat.h`, can be used to assign only the desired parameters, or all of the parameters in sequence.

```
#define VERSAT_MEM0A 5120
#define VERSAT_FU 6176

typedef struct versatFU {
    struct mem { int iter, per, duty, sela, start, shift, incr, delay, rvrs; }
        mem0A, mem0B, mem1A, mem1B, mem2A, mem2B, mem3A, mem3B;
    struct alu { int sela, selb, fns; }
```

```
        alu0, alu1, alulite0, alulite1, alulite2, alulite3;
    struct mult { int sela, selb, lonhi, div2; }
        mult0, mult1, mult2, mult3;
    struct bs { int sela, selb, lna, lnr; } bs;
} Versat;

typedef struct versatDE {
    int mem0A, mem0B, mem1A, mem1B, mem2A, mem2B, mem3A, mem3B;
    int alu0, alu1, alulite0, alulite1, alulite2, alulite3;
    int mult0, mult1, mult2, mult3, bs0;
} VersatDE;
```

Now, by including `versat.h`, all parameters are accessible through a single memory mapped structure initialized to the base address of each parameter base.

```
#include "versat.h"
static int base = VERSAT_MEM0A, versatFU = VERSAT_FU;
int main() {
    Versat *versatFu = (Versat*)base;
    VersatDE *versatDE = (VersatDE*)versatFU;
    /* ... */
    return 0;
}
```

Also, a structure can be defined with all required parameters and then copied into the required memory position simply by **C** structure assignment.

```
#include "versat.h"
static int base = VERSAT_MEM0A;
Versat config1 = { { 2000, 1, 1, 0, 0, 1, 1, 0, 0, } /* ... */ };
int main() {
    Versat *versatFu = (Versat*)base;
    /* ... */
    *versatFu = config1;
    /* ... */
    return 0;
}
```

Chapter 4

Development

The **lcc** compiler (see section [15, 16]) is a retargetable compiler with a single *front-end* for the **C** programming language (ANSI-C). As a retargetable compiler, multiple *back-ends* are available and new ones can be easily added. Currently, **lcc** produces **mips**, **sparc** and **intel-x86** assembly, amongst other output formats. Most formats, including the three referred, use an instruction selector (see section 2.3.2) to generate and optimize the output assembly code. The **lcc** compiler uses a specific instruction selection tool (**lburg**), included in the compiler distribution. Ideally, the creation of a new *back-end* corresponds to an **lburg** grammar description, some auxiliary functions, and a controlling structure. A reference to the controlling structure is then inserted into the `src/bind.c` file and the **makefile** update.

The **lburg** grammar description file is composed of three areas separated by a single line containing only the sequence `%%`, like other grammar description files for lexical analysis (see section 2.3.1) or syntactic analysis (see section 2.3.1). The first area contains declarations, the second area contains the grammar, and the last area may contain **C** language functions and variables. The declarations area includes a `%start` declaration, `%term` declarations, and **C** language declarations. The `%start` declaration identifies the grammar non-terminal that each AST tree must produce, based on the grammar tree patterns. If the start non-terminal can not be produced from any combination of the grammar rules, then an error is generated and no output is produced. The *Versat* grammar uses `stmt`, or statement, as the start symbol for the grammar. The `%term` declaration associates a symbolic name used in the patterns of each rule with the number stored in the terminal nodes of the tree (AST) which represents the program. For instance, a tree node containing an integer of metric 1, usually a **C** language `char` type, is labeled with the 1045 value by the **lcc** compiler, and associated with the name `CNSTI1` by the *Versat back-end* with the declaration `%term CNSTI1=1045`.

The values that correspond to each terminal type in the tree can be computed based on the operation, the data type and the metric. The operation is one of 35 types of instructions generated by the compiler for all **C** language constructs. The instruction are numbered from 1 (`CNST`) to 37 (`LABEL`) and 44 (`VREG`). The full list (see [20, p.84] include `CNST` (a literal constant), `ARG` (a function argument), `ASGN` (an assignment), `INDIR` (a pointer dereferencing), `CVF` (a conversion from floating point), `CVI` (a conversion from an integer), `CVU` (a conversion from an unsigned integer), `CVP` (a conversion from a pointer), `NEG`

(the symmetric value), CALL (a function call), LOAD (a register movement), RET (the return from a function), ADDRG (address of a global variable), ADDRF (address of a function argument), ADDRL (address of a local variable), ADD (arithmetic sum), SUB (arithmetic subtraction), LSH (a left shift), MOD (division integer remainder), RSH (a right shift), BAND (a bit-wise and), BCOM (a bit-wise complement), BOR (a bit-wise inclusive or), BXOR (a bit-wise exclusive-or), DIV (arithmetic division), MUL (arithmetic multiplication), EQ (an equal comparison), GE (a greater or equal comparison), GT (a greater than comparison), LE (a less or equal comparison), LT (a less than comparison), NE (a not equal comparison), JUMP (an unconditional jump), LABEL (a global label), VREG (a register variable). The data type can be F (floating point), I (signed integer), U (unsigned integer), P (any pointer type), V (void) and B (structure type). The metric is the size of the variable, where `sizeof(char)` must be 1 (see [20, p.79]) as is required by the **C** programming language. However, not all instructions can be applied for all data types, and are only available for some metric values. For instance, ADDRG only supports P for the processor word size, *i.e.* ADDRP4 for a 32-bit machine and ADDRP8 for a 64-bit machine. On the other hand, addition is supported for almost all data types, except void and structure, and all metrics.

The *Versat back-end* defines only metric 1 terminals since a character has metric 1, although it occupies 32-bits, as all other integer data types.

4.1 lcc compiler interface

As referred, the *back-end* is registered in the compiler by a single data structure, `Interface versatIR` in this case. This structure defines the processor metrics. In *Versat* all data types have a size metric of 1 except the **long long**, **double** and **long double** that should occupy 2 words, although they are currently not implemented. All data types are word aligned, so the align metric is always 1. These metrics correctly map address computations but fail to address literals, since **lcc** assumes that a type `char` is 8-bits and its size is 1 (`sizeof(char)==1`). Although this is true for most common computers of today, old computers and specific processors had different size words. Processors, like *Versat*, that are not used for general computing, do not have different size data types. Instead, all data types are 32-bits long, which is a wast of space for character processing.

In order to circumvent the **lcc** assumption that characters are always 8-bits long, the data types minimum and maximum values were hacked for the *Versat back-end* in the file `types.c`:

```
types.c:42
    int i; for (i = 0; bindings[i].ir != IR; i++);
    int versat = !strcmp(bindings[i].name, "versat");
types.c:51
    if (versat) p->u.limits.max.i = ones(32)>>1;
types.c:56
    if (versat) p->u.limits.max.u = ones(32);
```

However, pointer literals are not computed from data types but from metric sizes, assuming 8-bits per byte. Since the code (`8^ty->size`) is spread out all over the compiler, a major rewriting was needed. Instead, pointer literals should be assigned to integers and then converted into pointers, as referred in

the section 5.3. This is not a common **C** operation, specially in virtual memory machines, but is very useful in the real memory mapped *Versat* architecture.

The *back-end* data structure also defines some architecture requirements. These include the endian number format representation (`little_endian=1` in *Versat*), whether multiplication and division are executed in hardware or by software libraries (`mulops_calls=1` or library in *Versat*), whether the *back-end* can handle a **DAG** (directed acyclic graph), or only a tree (`wants_dag=0` or tree in *Versat*), or if it can handle passing structures to and from functions: `wants_callb=0` and `wants_argb=0`. When the *back-end* does not handle structure passing directly, the compiler does not generate `CALLB` or `ARGB` nodes, and block copies must be handled by a `blkloop` function.

The final part of the *back-end* data structure defines a set of procedures to handle specific parts of the code generation. These include `progbeg` (setup *back-end*), `progend` (finalize *back-end*), `rmap` (define a new register set), `segment` (change segment), `target` (assign register for specific instructions), `clobber` (spill registers before specific operations), `emit2` (emit code that can not handled by string template), `blkfetch` (block based fetch code), `blkstore` (block based store code), `blkloop` (memory block copy), `doarg` (register assignment to arguments), `local` (register assignment to locals), `function` (generate a function activation record), `defsymbol` (define a symbol), `address` (compute an address), `defconst` (define a literal constant), `defaddress` (define an address), `defstring` (define an string), `export` (declare an exported symbol), `import` (declare an external symbol), `global` (declare a global symbol), `space` (reserve uninitialized data) [20, p.79].

4.2 Register assignment

In order to implement the **C** language constructs, the processor must provide an accumulator to handle return values from functions, a stack pointer to save arguments, locals and spills, and a frame pointer to access arguments and locals by a fixed amount. All of these registers can be set in fixed memory positions, but the execution degradation is significant.

The *picoVersat* has 16 registers, but only registers `R1` through `R12` are available as program parameters. The register assignment set `R12` as a stack pointer, and `R11` as a frame pointer. The stack pointer is initially set to the highest memory position `0x1FFF`, and the stack grows downward to lower memory addresses. Also, the stack pointer points to the last used position. Hence, the address `0x1FFF` is used to store the return address `end` (see section 4.8) before the `main` is called.

The accumulator is not a fixed register and requires spilling only when non-void functions are called, in order to hold the return value. The first register `R1` (ACC=0) is assigned as accumulator. The `target` routine instructs the compiler to free **R1** before a `CALL+IUP`: `setreg(p, intreg[ACC])`. The same routine instruct the compiler to place the result of a `RET+IUP` in `R1`: `rtarget(p, 0, intreg[ACC])`.

The assignment of structures `ASGN+B` performs a block copy, and requires two temporary registers. The `clobber` routine spills registers `R1` and `R2` before the instruction is emitted: `spill(ACC|(ACC+1), IREG, p)`.

The remaining registers are of free use by the compiler. However, to make code generation more

efficient, about half of the available 10 registers (from R1 to R10) can be assigned to temporary values, and the others to store program variables. Program variables stored in register speed up significantly the program execution, specially if they require indexing, such as function arguments, locals and vector indices. As R1 and R2 are already used as temporaries by some instructions, registers R1 through R5 are defined as temporaries: tmask=0x1f. Registers R6 through R10 are defined as variables registers: vmask=0x3e0. Registers R12=SP and R11=FP are permanently assigned.

4.3 Code selection

The grammar for the code selection is defined in the second area of the **Iburg** description file. Each grammar rule defines a non-terminal target, a tree pattern, an output string, and a selection cost. For instance, the tree that adds a register with a constant and produces a register is

```
reg:    ADDI1(reg,con)    "\trdw %0\n\taddi %1\n\twrw %c\n" 3
```

The selection cost value represents the latency of the full instruction sequence. In *picoVersat* all instructions take a single clock cycle, so the cost is the number of instructions. For dynamic selection of instructions, the cost value literal is replaced by a function call. This function evaluates the node and returns a cost value. A low cost value, usually 0 or 1 represents a selectable instruction, while a high cost value, usually LBURG_MAX or 32767, is returned when the instruction sequence should not be selected. Since the compiler chooses the lowest cost sequence to generate each stmt, the grammar start symbol, high cost instructions are never used. The output strings have printf alike escape sequences. The terminal values in the tree sequence are represented by %0 for the first argument, %1 for the second argument, *etc.*, and %c for the result. The %a escape is used for the first symbol associated with a node, %b for the second symbol, *etc.* When an output strings starts with a # symbol, the output is redirected to the emit2 routine and more complex computation can be performed, other than simple string variable substitutions.

For debug purposes, this *back-end* strings start with a comment that identifies the selected rule in the output assembly file. However, since the *Versat* assembler also uses the # symbol for comments, a space must be inserted to avoid redirection of comments in rules to the emit2 routine.

The *Versat back-end* uses the non-terminals: stmt for statements (also the grammar start symbol), reg for register values, fpN for function activation frame arguments and locals, addrg for global variables (ADDRGP + MEM_BASE), con5 for small literals (0 to 32) used in shift operations, con1 for loop unrolling of single (1) shift operations, addr is a register that contains and address, and adddrj is a jump or call address (does not use MEM_BASE in *picoVersat-0.0*).

The non-terminal con represents a literal representable in 28 bits, a signed range from -134217728 to 134217727, that can be embedded in other *picoVersat* instructions like addi.

```
con:    CNSTI1    "%a" range(a, -134217728, 134217727)
reg:    con        " # reg: con\n\tldi %0\n" 1
reg:    CNSTI1    "# long constant\n" 2
```

```
reg:    ADDI1(reg,con)    "\trdw %0\n\taddi %1\n\twrw %c\n" 3
reg:    ADDI1(reg,reg)    "\trdw %0\n\tadd %1\n\twrw %c\n" 3
```

The tree pattern `con: CNSTI1` has a cost of 1 if the constant is in the defined `range` and is selected by the first rule, otherwise the third rule is used with cost of 2. The sum of a register with a constant `ADDI1(reg,con)`) has a cost of $1 + 3 = 4$ when the constant is within the range, and a cost of $2 + 3 = 5$ otherwise. Please note that even a constant within the `range` can use the sum of two registers (`ADDI1(reg,reg)`), but the cost is $1 + 1 + 3 = 5$ higher since placing a constant in a register adds 1 to the cost.

The `stmt: reg` rule states that a statement can be an expression, an assignment or function call for example, but has a zero (0) cost since the value can be left the register with not additional instructions.

4.4 Code emitting

The sequence too complex to emit code based only a string template must be handled by the `emit2` routine. This routine is selected by starting the rule output string with the `#` symbol. In *Versat* some instructions require temporary labels like `CALL`, `RSH` or `LSH` and must be handled by `emit2`.

All code in this routine is printed by a `print` routine that uses `%s` and `%s` to output variables, as in a regular `printf`. However, the examples presented in this section have the escape values replaced by `%0` for the first argument, `%1` for the second argument, and `%c` for the result. The `%d` is used for generated label numbers.

The `CALL` instruction does not exist in *picoVersat*. Its emulation must store the return address before jumping to the address of the routine. Since calls from different places require a different return address, the `genlabel(1)` function creates a new label name for each call. Note that the `nop` is executed twice, as a delay slot during the call and upon return from the function call. The resulting code is

```
    rdw R12
    addi -1
    wrw R12
    wrw RB
    ldi L%d
    wrwb
    ldi 0
    beqi %s
L%d nop
```

If the function was called with arguments, their size is stored in the first symbol (`%a`). After returning from the call, all arguments must be removed. This is achieved by adding their size to the stack pointer: `SP+=size`.

```
    rdw R12
    addi %a
    wrw R12
```

The CALL+IUP (for integers, unsigned and pointer return values) must define the register that holds the function return value by calling

```
setreg(p, intreg[ACC]);
```

in the target routine (intreg[ACC] = R1). Note that CALL+V does not return a value and does not need to assign a target register for the instruction.

In *picoVersat* the shift operations only shift one bit (left or right). In order to support multiple shifts in a single instruction a loop must be implemented. Shift operations are performed by a cycle that decrements the counter and shifts the destination register by one. Right shift replaces shft -1 by shft 1.

```
    rdw R%0
    wrw R%c
    rdw R%1
    wrw RB
    beqi L%d1
    nop
L%d rdw R%c
    shft -1
    wrw R%c
    rdw RB
    addi -1
    wrw RB
    bneqi L%d
L%d1 nop
```

The code generation requires that a comparison is coded as a branch when the condition hold *true*. If the condition hold *false*, the instruction should not branch to the given label. The comparison *greater-than-unsigned* (GTU1) requires that the result is *not-zero* an *no-carry*, a label was required to implement the *logical-or* using branches. Unlike all other comparison instructions, *greater-than-unsigned* had to be specifically coded

```
    rdw R%0
    sub R%1
    beqi L%d
    ldi 1
    and RC
    bneqi L%d
    ldi 0
    beqi %a
L%d nop
```

picoVersat does not have instructions for multiplication or division, like many low budget processors. For these processors, the operations are performed by library functions. The mulops_calls=1 flag in the *back-end* configuration data structure makes the compiler generate regular functions calls for these operations. However, the coding of these operations can be optimized, since SP=R12 manipulation is simpler because 3 pushes are performed in sequence, with register saves.

```
    rdw R12
    addi -1
    wrw RB
    rdw R%1
    wrwb
    rdw R12
    addi -2
    wrw RB
    rdw R%0
    wrwb
    rdw R12
    addi -3
    wrw R12
    wrw RB
    ldi L%d
    wrwb
    ldi 0
    beqi %s
L%d rdw R12
    addi 2
    wrw R12
    rdw R1
    wrw R%c
```

The `range` cost routine selects those literals that can handled inline with some instructions. For large integer literals two instructions must be emitted, one for the low bits and another for the high bits. Since the `ldi` instruction can handle upto 28 bit constants a `0xFFFFFFF` mask is used, while the `ldih` instruction sets the literal high nibble by shifting right 28 bits the constant.

```
    ldi 0x%x
    ldih 0x%x
    wrw R%c
```

The `LOAD` and conversion (`CVI`, `CVU`, `CVP`) instructions emit no code if the origin and destination registers are the same. Otherwise a move is emitted

```
    rdw R%0
    wrw R%c
```

The `ASGNB` instruction copies one data structure into another data structure. The instruction is implemented by block copy from `a` to `b`, with offsets `0` for both, with the given `p->syms[0]->u.c.v.i` words.

```
blkloop(b, 0, a, 0, p->syms[0]->u.c.v.i, blkregs);
```

Since two temporary registers are needed, aside the source and destination already assigned, one for the counter and another to hold the value being copied, a set of two registers is defined.

```
static int blkregs[] = { ACC+1, ACC+2 };
```

This set, composed of `R1` and `R2`, is used in the `ASGNB` instruction and is used by `clobber` routine to `spill` the registers prior to the execution of the `ASGNB` instruction.

```
case ASGN+B: spill(ACC|(ACC+1), IREG, p); break;
```

The return instruction (RET+IUP, not RET+V) must inform the compiler through the target routine that the return value is R1=ACC.

```
case RET+I: case RET+U: case RET+P: rtarget(p, 0, intreg[ACC]); break;
```

4.5 Function handling

The function() routine must handle all the specifics of defining a function, including its activation frame. The activation frame is the organization of arguments, return pointer, saved registers, frame pointer, and local variables of a routine. Its memory mapping depends on whether the arguments are passed in registers or on the stack, *etc.*

The gencode routine performs a simulation of the code generation for the routine. No code is produced, but it is determined the number of registers necessary for its implementation, the offset of its arguments and locals (local variables).

Before the code is actually generated by the emitcode routine, all clobbered registers must be saved by issuing push instructions for each one. The push of register-i is [--R12] = Ri, where R12=SP.

```
rdw R12
addi -1
wrw R12
wrw RB
rdw R%d
wrwb
```

When a routine is executing, registers can be spilled into the stack whenever the instruction requests the use of a specific register, or when no more registers are available. The arguments of a routine are stored above the return pointer, while the local variables are stored below the return pointer. Furthermore, their relative offsets to the return pointer remain fixed, even when registers are spilled to the stack. If we can keep track of the return pointer, all arguments and local are in a fixed position. Most compilers use a frame pointer FP=R11 to save this position, and moderns processors like **mips**, **sparc** or **arm** have a frame pointer register. If no registers are saved, the first argument is at FP+2, the second at FP+3, *etc.*, while FP+1 is the return pointer, and FP is the frame pointer of the old routine.

After spilling all register that the routine will use, the old frame pointer for the previous routine must also be saved, and the new frame pointer points to the current stack pointer; PUSH fp; MOV fp, sp

```
rdw R12
addi -1
wrw R12
wrw RB
rdw R11
wrwb
rdw R12
wrw R11
```

Before the routine code can be emitted, the space for all local variables must be reserved. Since `gencode` already determined the space required by the routine locals, the stack pointer is decremented by that amount, leaving a chunk of stack unused for those locals: `SP-=size`

```
rdw R12
addi -%d
wrw R12
```

After the routine is emitted (`emitcode`), all locals must be freed. Since all locals are below the frame pointer, moving the stack pointer to the frame pointer effectively ignore all locals below the stack frame. Then the old frame pointer must be restored: `MOV sp, fp; POP fp`

```
rdw R11
wrw R12
wrw RB
rdwb
rdwb
wrw R11
rdw R12
addi 1
wrw R12
```

At the end of the code generation for the routine, all saved registers must now be restored in reverse order. The pop of register-`i` is `Ri = [R12++]`.

```
rdw R12
wrw RB
rdwb
rdwb
wrw R%d
rdw R12
addi 1
wrw R12
```

Now the stack only contains the return pointer and the routine arguments. Since in **C** the caller must push and pop the arguments, in order to support variadic function arguments like `printf`, the routine is only required to remove the return pointer and jump to its location.

```
rdw R12
wrw RB
rdwb
rdwb
wrw RB
rdw R12
addi 1
wrw R12
ldi 0
beq RB
nop
```

The *picoVersat* assembler is a very simple tool. If the user names a function or a global variable after a *picoVersat* instruction, the assembler silently ignores and emits the stored value. Since some names are pretty common, like `and`, `add` or `sub`, the code checks whether the user defines any of those names, and issues an error.

```
static char *invalid[] = { "and", "bneq", "IMM_W", "rdwb", "ldi",
"DELAY_SLOTS", "MEM_BASE", "SEL_ADDR_W", "beqi", "R14", "R15", "xor",
"sub", "R10", "R11", "wrw", "add", "EXT_BASE", "RB", "ldih", "addi",
"R4", "R5", "R6", "R7", "R0", "R1", "R2", "R3", "RC", "R8", "REGF_BASE",
"R9", "CPRT_BASE", "DATA_W", "R12", "OPCODESZ", "R13", "rdw",
"beq", "ADDR_W", "shft", "wrwb", "INSTR_W", "bneqi", "REGF_ADDR_W",
"nop", 0 };
```

This behavior was detected many hours after an `add` routine was defined and the output made no sense.

```
for (i = 0; invalid[i]; i++)
    if (!strcmp(invalid[i], f->x.name))
        error("'%S' can not be used\n", f->x.name);
```

In `defconst` the literal is emitted unsigned since `.memset` does properly sign extend negative numbers. For instance, -12 is coded as `0x0FFFFFF4` and not as `0xFFFFFFF4`.

4.6 Code optimization

A code selection tool like **lburg** allows the insertion of alternative tree selection patterns that can be used in specific trees with an inferior cost than the generic rules required for all instructions.

As referred above, the add instruction (`ADD+IUP`) can be performed with a register `add` or with an immediate value `addi`. Although the cost is `3` in both cases, the use of an immediate value saves a register and the respective store instruction that was previously required. Note that the operation between two registers is required to perform sums, while the sum with a constant is an optimization when one of the values is not already stored in a register. The compiler can always choose to save the immediate in a register, and then perform the sum between registers. The costs should be defined in such a way that it is more costly to load the immediate to a register. The compiler also detects that the instruction is commutative, so there is no need for a `(reg, con)` and `(con,reg)` rules.

```
reg:    ADDI1(reg,con)    "\trdw %0\n\taddi %1\n\twrw %c\n" 3
reg:    ADDI1(reg,reg)    "\trdw %0\n\tadd %1\n\twrw %c\n" 3
```

The instructions `BAND+IU`, `BXOR+IU`, `ASGN+IUP`, `ARG+IUP` can not handle immediates. However, by replacing a `rdw` by a `ldi` instruction a register is also saved.

```
reg:    BANDI1(reg,con)    "\tldi %1\n\tand %0\n\twrw %c\n" 3
reg:    BANDI1(reg,reg)    "\trdw %0\n\tand %1\n\twrw %c\n" 3
```

The shift instructions are very costly since they must be implement through loop. In order to unroll the loop, optimizations can be defined for each shift value, as long as it is a literal. When both values

reside in registers, no optimizations are possible. For the single shift case, a constant of value 1 (in the range from 1 to 1) is defined (con1). The shift operations that use con1 must only perform a single shift. The same concept can be extended for 2, 3, *etc.*

```
con1:   CNSTI1     "%a"    range(a, 1, 1)
reg:    LSHI1(reg,con1)    "\trdw %0\n\tshft -1\n\twrw %c\n" 3
reg:    RSHI1(reg,con1)    "\trdw %0\n\tshft 1\n\twrw %c\n" 3
```

The *Versat* processor is controlled by *picoVersat* by writing values to specific memory positions. As such, vector addressing instructions are common and should be optimized. To optimize such instructions, the compiler was run in debug mode, where it emitted the trees it was selecting. Two such cases were identified, where global vectors where indexed by literal and assigned literals: `vec[6] = 3; ptr[6] = 3` Local vectors require frame pointer indexing and are implicitly slower. The two tree patterns selected correspond to global pointers and vectors: `int vec[10], *ptr;`

```
stmt:    ASGNI1(ADDP1(INDIRP1(addrg),con),con)
            "\twrw RB\n\trdwb\n\trdwb\n\taddi %1\n\twrw RB\n\tldi %2\n\twrwb\n" 7
stmt:    ASGNI1(ADDP1(addrg,con),con)
            "\taddi %1\n\twrw RB\n\tldi %2\n\twrwb\n" 4
```

4.7 ASM support

The use of an `asm` call is important to have a low level control over the *Versat*. Since the compiler did not support `asm` calls (see section 2.3.6) a generic support was added. Any *back-end* can activate the `doasm` flag in its `progbeg` routine and a `CALLASM` instruction is generated for selection.

The variable is declared in **c.h**:

```
extern int doasm;
```

The variable is defined in **main.c**:

```
int doasm; /* accept asm("assembly code") */
```

If the flag is set and the function is named `asm`, a string literal (`SCON`) is read. Otherwise an error is issued by `expect`. Then a `CALL` node is built. The change were added after line 297 in the file `expr.c`:

```
  if (doasm && p->u.sym && !strcmp(p->u.sym->name, "asm")) {
    Type ty = func(voidtype, NULL, 1);
    int tk;
    t = gettok();
    tk = t;
    expect(SCON);
    if (tsym && tk == SCON) {
      Symbol s = malloc(sizeof *tsym);
      *s = *tsym;
      s->name = strdup("asm");
      p->u.sym = s;
      p->u.sym->x.name = strdup(tsym->u.f.pt.file);
```

```
      expect(')');
      return tree(mkop(CALL, ty), ty, p, NULL);
    }
  }
```

The selection accepts a pointer to a global variable that represents the string generated.

```
addrj:   ADDRGP1    "%a"
stmt:    CALLASM(addrj)    " # ASM\n%0\n"    1
```

Note that only literals at compile time can be used because the values must be outputted to the assembly file. Since register assignment is performed by the compiler, there is no way of knowing which register will be assigned to a given variable. Exceptions are global variables, always referred by name, and locals that have a fixed offset to the frame pointer. However, in the later case the user must write simple routines, with no register spilling, so that the offsets are known.

4.8 Program bootstrapping

The program bootstrapping consists on setting up the processor before calling the `main()` routine, the actual calling of the `main()` routine, and the cleaning up after calling the `main()` routine.

The setup of the main routine sets the stack pointer to the top of the stack and sets the frame pointer to zero. The top of the stack is stored in R12 and must be determined from ADDR_W and MEM_BASE: `R12=MEM_BASE+2**(ADDR_W-1)-1`

```
      ldi 1
      wrw R12
      ldi ADDR_W
      addi -1
      wrw RB
_next rdw RB
      beqi _top
      rdw R12
      shft -1
      wrw R12
      rdw RB
      addi -1
      wrw RB
      ldi 0
      beqi _next
_top  rdw R12
      addi -1
      addi MEM_BASE
      wrw R12
```

No arguments (`argc, argv, envp`) are passed, so the `main` routine is directly invoked. The return address `end` is saved at the top of the stack, the frame pointer R11 is set to zero, the `main()` routine is called and the return address is defined:

```
        wrw RB
        ldi end
        wrwb
        ldi 0
        wrw R11 #FP=0
        beqi main
end     nop
```

The `nop` instruction is executed twice, as a delay slot for `beqi` and upon return from the routine.

On debug mode, upon return, before the run is terminated, a debug feature prints the `main` return value as a single hexadecimal nibble; possible values are from `0` to `9` and then the following ASCII codes `':'`, `';'`, `'<'`, `'='`, `'>'`, `'?'`. Therefore, simple test programs just return operation values from `main`.

```
end  ldi 0xF
     and R1
     addi 0x30
     wrw CPRT_BASE
     ldi 0xa
     wrw CPRT_BASE
```

Finally, to end the program, a trap must be generated. The trap address is `MEM_BASE+2**ADDR_W-1` and is determined in the same way as the top of the stack:

```
        ldi 1
        wrw R12
        ldi ADDR_W
        wrw RB
_again  rdw RB
        beqi _trap
        rdw R12
        shft -1
        wrw R12
        rdw RB
        addi -1
        wrw RB
        ldi 0
        beqi _again
_trap   rdw R12
        addi -1
        addi MEM_BASE
        wrw RB
        wrwb
```

4.9 Runtime support

Since the assembler does not support multiple files, and there is no support for explicit file liking, the runtime support must be added by include files. These included files have the usual declarations and defines, but also routines fully coded. There is no problem of multiple definitions since there is no linking.

The include/ directory includes some basic routines to support program development and debugging. These include basic printing (putchar.h, puts.h, printi.h, printf.h), conversion (atoi.h, itoa.h, strlen.h), memory (alloca.h, malloc.h) and *versat* support (versat.h, dma.h, assign.h, xdict.h, ends.h).

The putchar routine uses the *picoVersat* debug capability to print a single ASCII character to the simulation terminal. It allows the debug and testing of the program examples used in this work.

```
void putchar(int ch) {
    asm("\trdw R11\n\taddi 2\n\twrw RB\n\trdwb\n\trdwb\n\twrw CPRT_BASE\n");
}
```

The function is called using the usual **C** language convention, and then the function argument is fetched from the *stack-frame*. The offset (2) accounts for the saved *frame-pointer* and the function return address, both pushed to the stack after the argument was saved on the stack.

The mul.h, div.h and mod.h files include the routines invoked by the compiler for the **C** operators for multiplication (* $\rightarrow$ _mul), division (/ $\rightarrow$ _div), and remainder (% $\rightarrow$ _mod).

```
int _mul(int a, int b) {
    int mul;
    for (mul = 0; b; b >>= 1, a <<= 1)
        if (b & 1)
            mul += a;
    return mul;
}

int _div(int a, int b) {
    int x = 0;
    while (a > b) a -= b, x++;
    return x;
}

int _mod(int a, int b) {
    while (a > b) a -= b;
    return a;
}
```

The implementations are very simple and inefficient but prove the concept and fit the limited memory available.

Memory allocation on the heap uses the memory between static data (functions and global variables) and the top of the stack. The top of the stack is maintained by register R12. To signal the end of the static data, the compiler inserts an integer variable, called _end, and initialized to zero (0). The value of this variable is then used as the head of an allocation block list, entirely written in **C** (malloc.h). The allocator checks whether the required memory would overlap with the stack, but since stack allocations (function calls, arguments and local variables) do not perform the inverse check, stack overruns are possible.

Implementation of memory allocation on the stack (see the alloca **C** library routine) requires the effective movement of the stack pointer. Special care must taken during expression evaluation, since temporaries may clobber the stack. The implementation provided is very simple and decreases the stack by the given amount sp -= size (in 32-bit words). The sp() routine then returns the new stack

top value, which is the allocated block lowest address, since the stack runs from high addresses to low addresses.

4.10 Application integration

The compiler **lcc** is composed of several applications that, when used in sequence, produce an executable from a given source file. The `lcc/etc` directory contains the code for the generation of the top level `lcc` application that controls the preprocessor (**cpp**), the compiler (**rcc**), the assembler (**va**) and the loader. The `lcc/etc/versat.c` file controls arguments and invocation of these sub-applications. The preprocessor is invoked with `-Dversat` so that programs can use `$ifdef` directives to select specific code. The compiler must be invoked with `-target=versat` to select the *back-end*. The assembler is invoked with an optional third argument that points to `xdict.json` file. The *verilog* compiler (**iverilog**) works as a loader allows the generation of a final executable from the assembly file generated.

The *versat* assembler requires a `xdict.json` file, for defining constants, in the current directory. For a smooth integration, this file should be passed an optional third argument, enabling it to reside in a different directory.

In order to control the assembly file read by **iverilog**, the *picoVersat* `picoversat/rtl/src/xram.v` file (line 65) was modified to support the constant `INPUT` instead of the literal `"program.hex"`.

```
$readmemh(‘INPUT,mem,0,2**(‘ADDR_W-1)-1);
```

The **iverilog** tool must now be invoked with `-Dinput=\"source.hex\"` where `source.hex` is the input file.

4.11 Data engine incorporation

The `versat` CGRA configuration is memory mapped, thus each functional unit can be configured by writing to specific memory addresses. The writing can be performed by ordinary **C** language assignment instruction of the form `*addr = value;` where the `addr` variable was previously set to the specific memory address.

The data engine controls `19` functional units: `8` memories (from `0` to `3` and `A` to `B`), `6` ALUs (`2` full ALUs and `4` *lite*), `4` multipliers (from `0` to `3`) and a BS. Each functional unit is configured by a register, from address `6176` to `6194` when `N_W=5`.

```
typedef struct versatDE {
    int mem0A, mem0B, mem1A, mem1B, mem2A, mem2B, mem3A, mem3B;
    int alu0, alu1, alulite0, alulite1, alulite2, alulite3;
    int mult0, mult1, mult2, mult3, bs0;
} VersatDE;
```

Depending on the type of functional unit, a number of parameters can be defined by writing to a set of configuration register. Each memory includes `9` parameters (`iter`, `per`, `duty`, `sela`, `start`, `shift`, `incr`,

delay, and rvrs). Each ALU includes 3 parameters (sela, selb, and fns). Each multiplier includes 4 parameters (sela, selb, lonhi, and div2). The BS includes 4 parameters (sela, selb, lna, and lnr). These $8 \times 9 + 6 \times 3 + 4 \times 4 + 4 = 110$ configuration registers, and a clear register, are mapped from address 5119 to 5229.

```
typedef struct versat {
    struct mem { int iter, per, duty, sela, start, shift, incr, delay, rvrs; }
        mem0A, mem0B, mem1A, mem1B, mem2A, mem2B, mem3A, mem3B;
    struct alu { int sela, selb, fns; }
        alu0, alu1, alulite0, alulite1, alulite2, alulite3;
    struct mult { int sela, selb, lonhi, div2; }
        mult0, mult1, mult2, mult3;
    struct bs { sela, selb, lna, lnr } bs;
} Versat;
```

The xdict.h file defines the same xdict.json file constants for **C** programs. The ends.h file, made from mem_ends.h file, defines the constant values as integer literals and not as expressions, so they can be used in asm() directives, since their values must be known at compile time.

The versat() routine can be used to efficiently insert a compile time constant value in *Versat* configuration addresses, using only 2 instructions. Use the set() routine or the setvar() routine (in **C**, but slower) to insert runtime determined values. However, the fastest approach is to make a simple **C** indirect pointer assignment, where the pointer is initialized to the destination address, 10 instructions for the assignment (local or global scalar variable), and 9 to assign the address to the pointer.

```
/* ../obj/cpp mem_ends.c | ../obj/rcc -target=versat > mem_ends.va */
#include "xdict.h"
#include "ends.h"
#define str(s) #s /* stringify */
#define ALU_CLZ 13 /* program.h */
#define versat(alu,base,offset) \
    asm("\tldi " str(alu) "\n\twrc " str(base) "," str(offset) "\n")
void set(int value, int *addr) {
    asm("\trdw R11\n\taddi 2\n\twrw RB\n\trdwb\n\trdwb\n\twrw R1\n"
"\trdw R11\n\taddi 3\n\twrw RB\n\trdwb\n\trdwb\n\twrw RB\n\trdw R1\n\twrwb\n");
}
void setvar(int value, int *addr) { *addr = value; }

int main() {
    int pos = 0x123456, *addr = (int*)pos; /* see limitations */
    versat(ALU_CLZ,ALU0_CONFIG_ADDR,ALU_CONF_FNS_OFFSET);
    set(12, (int*)pos);
    setvar(13, (int*)pos);
    *addr++ = 14;
    *addr++ = 15;
    *addr = 16;
    return 1;
}
```

4.12 picoVersat versions

During the development of this work there were 3 major *picoVersat* versions, designated in this document as:

picoVersat-0: as of March 4 (`4a9c764`), is the start version. It requires a read-only `bootrom` that then transfers control to the user program. Functions and data are stored separately, requiring an `addi MEM_BASE` instruction to access data. This, however, poses a problem in **C** since function pointers are data, but the offset should not be added. The function segment size was only 256 words, restricting tests to very small examples. This is specially true because there is no `call` or stack support instructions, making those simple operations very long as detailed in this chapter.

picoVersat-1: as of July 27 (`ee36325`), is the first fully workable version. Both, `bootrom` and data offset, were removed. Memory is now 8192 words long, for both functions and data, making comprehensive testing of the compiler possible. This version exhibited problems with *flags*, but alternative code was inserted to overcome this limitation.

picoVersat-2: as of September 13 (`dda9dfa`), flags are now working and the `wrc` macro instruction was removed. All 16K-word of program and data memory are now acessible.

Chapter 5

Results

The aim of this work is to produce a workable **C** language compiler for the *Versat* architecture using the *picoVersat* instruction set. The success can be measured by the number of **C** language constructs that are working properly. Consequently, testing is of primordial importance, as are the range of tests used to exercise the compiler.

5.1 Functionality

The compiler, as far the tests were comprehensive, supports all **C** language integer constructs. Limitations are listed below.

Since the processor instruction set is reduced, as are the number of **Icc** terminals to be implemented by the compiler, the testing of each operation, on its own, is strait forward. Testing sequences of such operations may prove more difficult to test, since different **C** programs can produce different selection matches.

5.2 Testing

In the test of a compiler, where a small change can affect the generation of multiple instructions, a good set of regressive tests is very important. In order to automate the process, a `test/` directory was setup. This directory includes a set of **.c** test files and the expected output **.out**.

The `Makefile` compiles, executes and compares the new result with the previously stored result. All differences in output are printed and can then be analyzed.

Since the output from **iverilog** includes the number of clocks spent, it is easy to compare whether the changes in the compiler result in improvements, or in performance degradation.

Some tests are very simple and its output can be easily predicted. To make testing even simpler, the return value of the `main` routine is printed, unless the NORET environment variables is defined. Upon return from the `main` routine, the lowest nibble is printed as an ASCII starting at 0. This means

that values between 10 and 15 are printed as the ASCII character at the respective offset, namely the sequence: :, ;, <, =, >, ?.

More complex tests can be compiled with **gcc** and executed to access the expected output. This, however, can not be performed if the examples include `asm` calls, since the code can only be executed by the *Versat*, or by the **iverilog** simulator, and not by the native testbench processor.

A set of 86 regression tests is currently being used, ranging from specific operator testing to complex recursive and iterative examples.

5.3 Limitations

The **C** language imposes that `sizeof(char)==1` as does the **lcc** compiler (see [20, p.79]). This works fine as far as `sizeof(char)` can be 32-bits. However, additionally, the **lcc** assumes through out the code that 8 is the number of *bit-per-byte*. If it was a variable, one could set it to 32. As it is hardcoded, all address literals will be truncated to 8-bits (`8^ty->size`).

```
int *addr = (int*)0x123456;
```

This can be avoided by setting an integer to the required value and then assigning it to a pointer. This works since integer literals are 32-bit wide and the conversion to pointer, controlled by the *back-end*, does not truncate the value.

```
int *addr, value = 0x123456;
addr = (int*)value;
```

Nevertheless, defining literal pointer is never a good predictive in virtual memory machines. In *Versat* it is useful to map variables to specific addresses.

Due to the same reason, a warning message is issued (`shifting an 'int' by 12 bits is undefined`) but the code is correctly generated.

In the initial version of *picoVersat*, all global data must be added by `MEM_BASE=512`. Since this is performed when addresses are fetched, static assignments store the unadded value. Therefore, all accesses must be added by 512. Must add `512` to global pointers in `picoversat-0.0`

```
int mem[10], *base = mem;
int main() { base[6+512] = 9; return return mem[6]; }
```

This can be avoided if assignment is performed during execution (not at compile time), even if the variable is global.

```
int mem[10];
int main() { int *base = mem; base[6] = 9; return return mem[6]; }
```

Signed multiplication, division and modulus (`_mul`, `_div` and `_mod`) do not generate carry since the flags register of *picoVersat* is read-only.

The **C** programming languages relies on separate compilation, where several files are independently compiled and then linked together. However, there is no linker in the *Versat* system and the assembler **va** does not support multiple files. The solution is to perform linking with **cpp** include directives. While in normal **C** the `.c` should be included, rather compiled, the inclusion of **.h** as well as **.c** accomplishes the desired result. Since there is no linking, only multiple inclusion of files must be avoided.

The *Versat* architecture is meant to be used offline and no form of argument passing to the `main` routine is available. Consequently, the stack is initialized at the top. Therefor, even if the program declares arguments to the `main` routine (`argc, argv, envp`) they should never be accessed. Also, since the system has no memory management unit, all illegal accesses are silently ignored by the system. Highly recursive routines that exhaust the stack will have unpredictable behaviors, since they will begin to overwrite the top of the code. Even if it is not the compilers responsibility, it something that the programmer should be aware, especially when transitioning from a virtual managed memory system.

Finally, the compiler does not support floating point data types, since every operation must be supported by library routines. This is the case for many android devices, namely smartphones. However, the *Versat* purpose is to perform integer arithmetic operations fast and is not aimed at scientific programming. The error message `compiler error in _label--Bad terminal` is issued by the compiler when it cannot handle a given operation, namely floating point operations.

5.4 Register assignment

Register assignment in compiler design considers two types of registers: global registers that hold variable values and scratch registers that hold temporary values. The **lcc** compiler defines these registers by setting a mask for each type of register. It is up to each *back-end* to define the mask values according to processor capabilities. For instance, the **sparc** processor defines 4 sets of 8 registers: global, temporary, input and output; where the later two sets replace the stack for argument passing. In **i386** all 7 registers are temporary, while **mips** uses half for each purpose $(16 + 16)$.

Since the *picoVersat* has no specific register assignment, a study was carried out in order to assess the best balance between global and scratch registers. Registers `R0` and `R13` to `R15` are used to communicate with *Versat* and are invisible to the compiler. The stack is controlled by a *stack pointer* (`R12`) and a *frame pointer* (`R11`). The remaining registers (`R1` to `R10`) compose a mask `7FE`, where the lowest bit (`R0`) is omitted for register assignment, and the highest used bit `400` is `R10`. The register `R1` is used to return function values and all arguments are passed on the stack. At least two registers must be used as scratch for binary operations temporaries. The compiler allows the definition of a `tmask` for temporaries and a `vmask` for variables.

Initially, in run 1, the experiment uses all registers for temporaries. Each run adds a variable register at the expense of a temporary, until only two temporaries remain (run 9). Three examples where used: `assign`, `repeating locals` and `bubble sort`. The first two represent opposite extremes of register usage, while the last is a more balanced and realistic example.

The first example uses the **C** language right associative *assign* operator where each new assignment

to the variable `a` requires a new temporary register.

```
int f() { return 1; }
int main()
{
  int a;

  a = f() + (a =
      f() + (a =
      f() + (a =
      f() + (a =
      f() + (a =
      f() + (a =
      f() + (a =
      f() + (a =
      f() + (a =
      f() + (a =
      f() + (a =
      f() + (a =
      f() + (a = 1
                  ));
  return a;
}
```

The register usage shows that each assign uses a register 4 times at the expense of the return register `R1`. The best solution, represented by the lowest clock count, is to use only two temporaries, since more variables imply more stack (`R12`), saves and restores between each call to the function **f**.

run	vars	vmask	tmask	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	clks
1	0	000	7FE	21	4	4	4	4	4	4	4	6	44	27	82	677
2	1	400	3FE	23	4	4	4	4	4	4	6	42	17	14	82	638
3	2	600	1FE	25	4	4	4	4	4	6	42	0	17	16	77	633
4	3	700	0FE	27	4	4	4	4	6	42	0	0	17	18	72	628
5	4	780	07E	29	4	4	4	6	42	0	0	0	17	20	67	623
6	5	7C0	03E	31	4	4	6	42	0	0	0	0	17	22	62	618
7	6	7E0	01E	33	4	6	42	0	0	0	0	0	17	24	57	613
8	7	7F0	00E	35	6	42	0	0	0	0	0	0	17	26	52	608
9	8	7F8	006	39	42	0	0	0	0	0	0	0	17	28	47	603

The second example uses lots of repeating local variables so that each one is assigned a register, for its uses from the first to last line, if one is available.

```
int func(int a, int b, int c, int d, int e, int f, int g, int h, int i, int j, int k) {
    a = a + b - c - d - e + f - g + h + i + j + k;
    b = a - b + c + d - e - f + g - h + i - j - k;
    c = a + b - c - d + e + f + g - h + i + j - k;
    d = a - b - c + d - e + f - g + h - i - j - k;
    e = a + b + c - d - e - f - g + h - i + j - k;
    f = a - b - c + d - e + f + g - h - i - j + k;
    g = a + b - c - d + e + f + g - h + i + j - k;
    h = a - b + c + d - e - f - g + h + i - j - k;
    i = a + b - c - d - e + f + g + h + i + j - k;
    j = a - b - c + d - e + f + g - h - i - j - k;
    k = a + b + c - d + e - f - g - h - i + j + k;
    return a + b + c + d + e + f - g + h - i + j - k;
}

int main() {
    return func(10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0);
}
```

As expected, the best solution is to use highest of temporaries in order to reduce frame pointer (`R11`) accesses to stack saved values.

run	vars	vmask	tmask	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	clks
1	0	000	7FE	45	27	12	13	48	46	40	45	54	62	68	56	956
2	1	400	3FE	53	31	13	54	46	40	47	54	62	0	78	51	1001
3	2	600	1FE	61	35	52	46	48	49	56	62	0	0	89	46	1051
4	3	700	0FE	89	39	59	63	51	56	62	0	0	0	101	41	1106
5	4	780	07E	109	43	75	49	74	79	0	0	0	0	113	36	1161
6	5	7C0	03E	146	45	84	58	105	0	0	0	0	0	124	31	1211
7	6	7E0	01E	156	88	112	94	0	0	0	0	0	0	138	26	1278
8	7	7F0	00E	171	117	176	0	0	0	0	0	0	0	154	21	1356
9	8	7F8	006	231	249	0	0	0	0	0	0	0	0	172	16	1447

The last example, the *bubble sort*, uses a mixture temporaries and variable reuses.

```
#include "printi.h"

int bubble(int list[], int n) {
    int c, d, t, swap, cnt = 0;

    for (c = 0; c < n - 1; c++) {
```

```
        for (swap = 0, d = n - 1; d > c; d--)
            if (list[d - 1] > list[d]) {    /* Swapping */
                swap++;
                t = list[d];
                list[d] = list[d - 1];
                list[d - 1] = t;
            }
        if (!swap)
            break;
        cnt++;
    }
    return cnt;
}

int v[] = { 7, 4, 9, 6, 2, 1, 3, 5, 8, 0 };

int main() {
    int i, size = sizeof(v) / sizeof(v[0]), cnt = bubble(v, size);
    for (i = 0; i < size; i++) {
        putchar(v[i] + '0');
        putchar(' ');
    }
    printi(cnt, 10);
    putchar('\n');
    return 0;
}
```

This example exploits the tradeoff between global and temporary register usage. In the first runs the compiler is unable to use all temporaries. In the last runs some variable registers are left unassigned and the number of required execution clocks rises again.

run	vars	vmask	tmask	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	clks
1	0	000	7FE	2	0	0	0	0	4	5	15	30	45	31	36	9855
2	1	400	3FE	2	0	0	0	0	4	7	29	41	11	24	36	8193
3	2	600	1FE	2	0	0	0	4	5	27	37	7	11	19	41	7714
4	3	700	0FE	2	0	0	0	5	25	37	4	7	11	17	41	7399
5	4	780	07E	2	0	0	5	19	37	6	4	7	11	13	46	7022
6	5	7C0	03E	2	0	5	19	33	6	6	4	7	11	9	49	6949
7	6	7E0	01E	2	5	19	33	0	6	6	4	7	11	9	49	6949
8	7	7F0	00E	5	19	33	0	0	6	6	4	7	11	9	44	6921
9	8	7F8	006	21	28	0	0	0	6	6	4	7	11	13	41	7492

Based on experience with the examples above, a balanced approach should work best in most cases. Therefor, the first five registers, `R1` to `R5`, are used as temporaries (`tmask=0x003E`) and the remaining five, `R6` to `R10`, are used as variables (`vmask=0x07C0`).

5.5 Efficiency considerations

Calls are very expensive operations for any processor. *Intel Inc.* has made a significant effort over the year to address this problem. In the last years, its high end processors provide faster *calls* than *jumps* at the expense of higher transistor count. In a processor like *picoVersat*, the problem is magnified since no stack specific registers or opcodes are available.

A function call in the **C** programming language requires:

1. **argument passing** by pushing values to the stack;
2. **calling** the desired routine;
3. **saving used registers** before the routine destroys its values;
4. **frame pointer** saving to access arguments and locals;
5. **allocate space** for local variables;
6. actually performing the routine operations;
7. **restoring frame pointer** of the previous routine;
8. **restoring used registers** previous values;
9. **returning** to the calling routine;
10. **removing arguments** from stack.

The present compiler detects when a routine accesses no arguments or locals and does not emit frame pointer code. So, if a routine only uses global variables, the call becomes a bit more efficient. Some of the tests used become upto 5% faster by removing the frame pointer in routines where it not needed.

As any routine can be called many times, even recursively, the compiler must save, at the beginning, and restore, at the end, all the registers the routine uses. This means that, at the start of the program,

the `main` routine will spill all registers it will use, although they have no defined value. Such procedure is required since the routine may be recursively called. However, in most cases, the `main` routine is only invoked once, at the start of the program. The NOSAV environment variable can be set if the `main` routine is not used recursively and no registers will saved by the compiler. This special hack can be dangerous to use, but it makes `main` based programs more efficient.

The *picoVersat* controls *Versat* by setting specific values to predefined memory positions. The use of a routine to perform such a task is a very expensive way to change memory positions, either through `asm` directives or standard **C** code, as the tests `set.c` and `setvar.c` show, respectively. Memory values can be efficiently changed by assigning to a pointer `*addr=val` (see Limitations, above).

During this work, the *picoVersat* evolved. The use of a single memory, for program code and data, removed the need for a `addi MEM_BASE` instruction for each variable load and store, resulting in a 5% improvement over all the regression test in use, at the time.

Finally, the compiler some times generates a register read after the same register was written by another instruction selection. At least, the read can be suppressed, but **lcc** provides no peephole optimizer for final code cleanup.

5.6 Compiler instalation

The compiler itself, **lcc**, can be invoqued directly with the `-target=versat` option, as long as the input file has already been preprocessed (**cpp**). The compiler output is a *picoVersat* assembly, that can then fed to the *versat* assembler (**va**).

However, the complete compilation process, from **C** language source file to **iverilog** simulation executable, can be integrated as in a standard high-level compiler.

Section 4.10 describes the requirements for such an integration. The compiler `Makefile`s, in the main and `versat/` directories, can be used to provide the instalation of all required files for a complete development environment. By default, without any changes to the `Makefile`s, the compiler development environment is placed under the `/usr/local/versat` directory.

The default directories for the compiler installation (`make install`) are predefined as `/usr/local/versat/lcc` for the compiler files (`lcc`, `cpp`, `rcc`, `va`, and `xdict.json`), and can be redefined at compile time or using the LCCDIR environment variable at runtime. The *picoVersat* `rtl/` files (`include/`, `src/`, and `testbench/`) should be copied to `/usr/local/versat/pico` (defined at compile time). Also the **iverilog** compiler is defined at compile time as residing in `/usr/local/bin/`.

The structure of the installed files, in the current version is:

```
/usr/local/versat/lcc/lcc
/usr/local/versat/lcc/cpp
/usr/local/versat/lcc/rcc
/usr/local/versat/lcc/va
/usr/local/versat/lcc/xdict.json
/usr/local/versat/lcc/include/strlen.h
/usr/local/versat/lcc/include/umod.h
/usr/local/versat/lcc/include/errno.h
/usr/local/versat/lcc/include/malloc.h
```

```
/usr/local/versat/lcc/include/umul.h
/usr/local/versat/lcc/include/itoa.h
/usr/local/versat/lcc/include/Makefile
/usr/local/versat/lcc/include/stdarg.h
/usr/local/versat/lcc/include/mem_ends.h
/usr/local/versat/lcc/include/xdict.h
/usr/local/versat/lcc/include/atoi.h
/usr/local/versat/lcc/include/puts.h
/usr/local/versat/lcc/include/dma.h
/usr/local/versat/lcc/include/versat.h
/usr/local/versat/lcc/include/ends.h
/usr/local/versat/lcc/include/mul.h
/usr/local/versat/lcc/include/xdictinc
/usr/local/versat/lcc/include/div.h
/usr/local/versat/lcc/include/ends.cbc
/usr/local/versat/lcc/include/udiv.h
/usr/local/versat/lcc/include/printf.h
/usr/local/versat/lcc/include/mod.h
/usr/local/versat/lcc/include/alloca.h
/usr/local/versat/lcc/include/printi.h
/usr/local/versat/lcc/include/gnuc.h
/usr/local/versat/lcc/include/putchar.h
/usr/local/versat/lcc/include/assign.h
/usr/local/versat/pico/testbench/sim_xtop.cpp
/usr/local/versat/pico/testbench/xtop_tb.v
/usr/local/versat/pico/include/xdefs.vh
/usr/local/versat/pico/src/xaddr_decoder.v
/usr/local/versat/pico/src/xctrl.v
/usr/local/versat/pico/src/xram.v
/usr/local/versat/pico/src/xregf.v
/usr/local/versat/pico/src/xcprint.v
/usr/local/versat/pico/src/xtop.v
```

After adding the `/usr/local/versat/lcc` directory to the PATH environment variable, an executable example can be produced with the command:

```
lcc example.c -o example
```

The example can then be run with:

```
./example
```

Please note that the *versat* memory dump `.hex` file is stored in the `/tmp` directory.

Chapter 6

Conclusion

The development of a full compiler for the *Versat* architecture is a complex task. As any compiler, the main purpose is to provide code translation from a high-level language, like the **C** programming language, to the assembly language accepted by the *Versat* architecture. Although a simple compiler exists, it only provides for very simple language constructs. It does not support variables or declarations, only the processor registers are available. Also, no functions or structures are supported, making the algorithm development difficult. This work aims at providing a complete **C** language compiler for the *Versat* architecture.

The first step was to decide whether to improve the existing compiler or to develop a new compiler. Due to the limitations of the existing compiler, the lack of documentation or support, it was decided to develop a new compiler altogether. However, there are quite a few compiler frameworks, namely for the **C** language, that can be used to build a compiler for a new architecture. Although they do not provide out of the box support a processor like *Versat*, they are retargetable compilers, i.e., they offer support for many different processors. After an analysis of some of the more widespread retargetable **C** compilers, the **lcc** compiler framework was selected. It provided a code generation selection tool to ease the code generation process, and select the lowest cost instruction sequence. The compiler was supported by good documentation and examples for the most widespread high-end processors.

The bulk of the work is developed in a *back-end* file that describes the capabilities of the *picoVersat* assembler, from the point of view of a **C** language compiler. The *picoVersat back-end* is configured through a structure that parameterizes the code generator, and includes a set of routine pointers to specific *back-end* procedures. These procedures handle specific parts of the code generation, such as the preamble or the activation record for each user defined routine. However, most operations are described by a tree grammar. The tree grammar describes most target processor operations with an associated cost. Instruction selection optimization is achieved by providing different tree grammar combination of different costs.

The `asm` extension was added to **lcc** to provide direct control over *Versat* and *picoVersat*. This extension required changes to the compiler *front-end*, since **lcc** is an ANSI compiler. The `asm` extension was initially introduced by **gcc**, but has been added to other compilers, although it is not standard **C**.

The register assignment can be tailored to each processor characteristics. For *picoVersat*, different combinations of temporary (`tmask`) and variable (`vmask`) register assignment combinations were tested. The register usage and total execution clock time was measured for a set examples with different register requirements. A balanced version, with almost the same number of temporary and variable register was selected.

6.1 Achievements

The present work introduces the CGRA concept and analyses the *Versat* accelerator and its *picoVersat* controller. It highlights the *Versat* characteristics from a programming point of view, so that a compiler can be developed. It introduces the basic compiler development techniques and surveys existing retargetable **C** language compilers. After a comparative analysis, a *Versat* compiler is introduced as an evolution of an existing **C** language compiler.

The development work first addressed the support for *picoVersat* as an **lcc** compiler *back-end*. Integration of the compiler with **lcc**'s own preprocessor (**cpp**), the *Versat* assembler (**va**) and the *Verilog* simulator (**iverilog**) was added for a smooth compilation of examples from source to execution. A large set of regression tests ensure that future changes do not compromise existing functionality. Then mechanisms were added for the control of the *Versat* CGRA, using **C** language structures and assembler macros.

The resulting compiler provides all the requirements initially set out for this work as referred in section 1.3.

6.2 Future Work

As previously concluded, the task to be undertaken required the development of the compiler for the *Versat* architecture using the existing retargetable **lcc** compiler framework for the **C** language. Future work should essentially address some of the limitations referred in section 5.3, namely *bits-per-byte*, wide integers and floating point numbers.

Since *Versat* only manipulates 32-bit quantities and the **C** language imposes that `sizeof(char)==1`, the usual 8 *bits-per-byte* of most compilers, including **lcc**, do not hold true in the *Versat* architecture. To allow `sizeof(char)` to be 32-bits wide, significant changes had to be made to the core of the **lcc** compiler. These changes were not performed, resulting in the limitation, referred in section 5.3, that a literal must first be assigned to an integer variable and only then this variable can be assigned to an integer pointer, since the conversion does not truncate the original literal value. Additionally, shift operations emit a warning message when shifted by a literal for more than 8 bits.

Support for 64-bit wide integers, known as `long long` in **C**, can be added. However all operations must be performed by software routines, since *picoVersat* only handles 32-bit integers. Nevertheless, this requires changes to the compiler, since even basic operations, like addition or subtraction, must emit function calls to the respective software routines, instead of the *picoVersat* assembly instructions. As for

64-bit wide integers, floating point support would require the same approach. Until then, the compiler emits an error message whenever those data types are used in the programs being compiled.

Appendix A

picoVersat back-end for lcc

```
%{
/* versat.md by gcrs IST,2019 */
#define ACC 0
#include "c.h"
#define NODEPTR_TYPE Node
#define OP_LABEL(p) ((p)->op)
#define LEFT_CHILD(p) ((p)->kids[0])
#define RIGHT_CHILD(p) ((p)->kids[1])
#define STATE_LABEL(p) ((p)->x.state)
static void address(Symbol, Symbol, long);
static void blkfetch(int, int, int, int);
static void blkloop(int, int, int, int, int, int[]);
static void blkstore(int, int, int, int);
static void defaddress(Symbol);
static void defconst(int, int, Value);
static void defstring(int, char *);
static void defsymbol(Symbol);
static void doarg(Node);
static void emit2(Node);
static void export(Symbol);
static void clobber(Node);
static void function(Symbol, Symbol [], Symbol [], int);
static void global(Symbol);
static void import(Symbol);
static void local(Symbol);
static void progbeg(int, char **);
static void progend(void);
static void segment(int);
static void space(int);
static void target(Node);
extern int ckstack(Node, int);
static Symbol intregw, intreg[32]; /* must be 32! */
static int cseg;
static char *invalid[] = { "and", "bneq", "IMM_W", "rdwb", "ldi",
"DELAY_SLOTS", "MEM_BASE", "SEL_ADDR_W", "beqi", "R14", "R15", "xor",
```

```
"sub", "R10", "R11", "wrw", "add", "EXT_BASE", "RB", "ldih", "addi",
"R4", "R5", "R6", "R7", "R0", "R1", "R2", "R3", "RC", "R8", "REGF_BASE",
"R9", "CPRT_BASE", "DATA_W", "R12", "OPCODESZ", "R13", "rdw",
"beq", "ADDR_W", "shft", "wrwb", "INSTR_W", "bneqi", "REGF_ADDR_W",
"nop", 0 };
static int blkregs[] = { ACC+1, ACC+2 };
%}
%start stmt
%term CNSTI1=1045 CNSTU1=1046 CNSTP1=1047
%term ARGB=41
%term ARGI1=1061 ARGU1=1062 ARGP1=1063
%term ASGNB=57
%term ASGNI1=1077 ASGNU1=1078 ASGNP1=1079
%term INDIRB=73
%term INDIRI1=1093 INDIRU1=1094 INDIRP1=1095
%term CVII1=1157 CVIU1=1158
%term CVPP1=1175 CVPU1=1176
%term CVUI1=1205 CVUU1=1206 CVUP1=1207
%term NEGI1=1221
%term CALLASM=215 CALLV=216 CALLB=217
%term CALLI1=1237 CALLU1=1238 CALLP1=1239
%term RETV=248 RETI1=1269 RETU1=1270 RETP1=1271
%term ADDRGP1=1287 ADDRFP1=1303 ADDRLP1=1319
%term ADDI1=1333 ADDU1=1334 ADDP1=1335
%term SUBI1=1349 SUBU1=1350 SUBP1=1351
%term LSHI1=1365 LSHU1=1366
%term MODI1=1381 MODU1=1382
%term RSHI1=1397 RSHU1=1398
%term BANDI1=1413 BANDU1=1414
%term BCOMI1=1429 BCOMU1=1430
%term BORI1=1445 BORU1=1446
%term BXORI1=1461 BXORU1=1462
%term DIVI1=1477 DIVU1=1478
%term MULI1=1493 MULU1=1494
%term EQI1=1509 EQU1=1510
%term GEI1=1525 GEU1=1526
%term GTI1=1541 GTU1=1542
%term LEI1=1557 LEU1=1558
%term LTI1=1573 LTU1=1574
%term NEI1=1589 NEU1=1590
%term JUMPV=584
%term LABELV=600
%term LOADB=233
%term LOADI1=1253 LOADU1=1254 LOADP1=1255
%term VREGP=711
%%
reg:   INDIRI1(VREGP)   "# read register\n"
reg:   INDIRU1(VREGP)   "# read register\n"
reg:   INDIRP1(VREGP)   "# read register\n"
```

```
stmt:   ASGNI1(VREGP,reg)   "# write register\n"
stmt:   ASGNU1(VREGP,reg)   "# write register\n"
stmt:   ASGNP1(VREGP,reg)   "# write register\n"
stmt:   reg   ""

stmt: ARGB(INDIRB(reg))  "# ARGB\n"
stmt: ASGNB(reg,INDIRB(reg))  "# ASGNB\n"

con:   CNSTI1   "%a" range(a, -134217728, 134217727)
con:   CNSTU1   "%a" range(a, -134217728, 134217727)
con:   CNSTP1   "%a" range(a, -134217728, 134217727)

reg:   CNSTI1   "# long constant\n" 2
reg:   CNSTU1   "# long constant\n" 2
reg:   CNSTP1   "# long constant\n" 2

fpN:   ADDRFP1 "%a" 1
fpN:   ADDRLP1 "%a" 1

reg:   fpN " # fpN\n\trdw R11\n\taddi %0\n\twrw %c\n" 3
reg:   INDIRI1(fpN) " # INDIRI1(fpN)\n\trdw R11\n\taddi %0\n\twrw RB\n\trdwb\n\trdwb\n\twrw %c\n" 6
reg:   INDIRU1(fpN) " # INDIRU1(fpN)\n\trdw R11\n\taddi %0\n\twrw RB\n\trdwb\n\trdwb\n\twrw %c\n" 6
reg:   INDIRP1(fpN) " # INDIRP1(fpN)\n\trdw R11\n\taddi %0\n\twrw RB\n\trdwb\n\trdwb\n\twrw %c\n" 6
stmt:   ASGNI1(fpN,reg)   " # ASGNI1(fpN,reg)\n\trdw R11\n\taddi %0\n\twrw RB\n\trdw %1\n\twrwb\n" 5
stmt:   ASGNU1(fpN,reg)   " # ASGNU1(fpN,reg)\n\trdw R11\n\taddi %0\n\twrw RB\n\trdw %1\n\twrwb\n" 5
stmt:   ASGNP1(fpN,reg)   " # ASGNP1(fpN,reg)\n\trdw R11\n\taddi %0\n\twrw RB\n\trdw %1\n\twrwb\n" 5
stmt:   ASGNI1(fpN,con)   " # ASGNI1(fpN,con)\n\trdw R11\n\taddi %0\n\twrw RB\n\tldi %1\n\twrwb\n" 5
stmt:   ASGNU1(fpN,con)   " # ASGNU1(fpN,con)\n\trdw R11\n\taddi %0\n\twrw RB\n\tldi %1\n\twrwb\n" 5
stmt:   ASGNP1(fpN,con)   " # ASGNP1(fpN,con)\n\trdw R11\n\taddi %0\n\twrw RB\n\tldi %1\n\twrwb\n" 5

addrg:   ADDRGP1      " # ADDRGP1\n\tldi %a\n\twrw %c\n"   3
reg:   INDIRP1(addrg)   " # INDIRP1(addrg)\n\trdw %0\n\twrw RB\n\trdwb\n\trdwb\n\twrw %c\n" 5
reg:   INDIRU1(addrg)   " # INDIRI1(addrg)\n\trdw %0\n\twrw RB\n\trdwb\n\trdwb\n\twrw %c\n" 5
reg:   INDIRI1(addrg)   " # INDIRI1(addrg)\n\trdw %0\n\twrw RB\n\trdwb\n\trdwb\n\twrw %c\n" 5
stmt:   ASGNI1(addrg,reg)   " # ASGNI1(addrg,reg)\n\trdw %0\n\twrw RB\n\trdw %1\n\twrwb\n" 4
stmt:   ASGNU1(addrg,reg)   " # ASGNU1(addrg,reg)\n\trdw %0\n\twrw RB\n\trdw %1\n\twrwb\n" 4
stmt:   ASGNP1(addrg,reg)   " # ASGNP1(addrg,reg)\n\trdw %0\n\twrw RB\n\trdw %1\n\twrwb\n" 4

reg:   con      " # reg: con\n\tldi %0\n\twrw %c\n"   2
reg:   INDIRI1(reg)   " # reg: INDIRI1(reg)\n\trdw %0\n\twrw RB\n\trdwb\n\trdwb\n\twrw %c\n"   5
reg:   INDIRU1(reg)   " # reg: INDIRU1(reg)\n\trdw %0\n\twrw RB\n\trdwb\n\trdwb\n\twrw %c\n"   5
reg:   INDIRP1(reg)   " # reg: INDIRP1(reg)\n\trdw %0\n\twrw RB\n\trdwb\n\trdwb\n\twrw %c\n"   5
reg:   addrg   "%0"

reg:   LOADI1(reg)   " # LOADI1(reg)\n\trdw %0\n\twrw %c\n" move(a)
reg:   LOADU1(reg)   " # LOADU1(reg)\n\trdw %0\n\twrw %c\n" move(a)
reg:   LOADP1(reg)   " # LOADP1(reg)\n\trdw %0\n\twrw %c\n" move(a)
```

```
reg:   ADDI1(reg,con)   " # ADDI1(reg,con)\n\trdw %0\n\taddi %1\n\twrw %c\n" 3
reg:   ADDI1(reg,reg)   " # ADDI1(reg,reg)\n\trdw %0\n\tadd %1\n\twrw %c\n" 3
reg:   ADDU1(reg,con)   " # ADDU1(reg,con)\n\trdw %0\n\taddi %1\n\twrw %c\n" 3
reg:   ADDU1(reg,reg)   " # ADDU1(reg,reg)\n\trdw %0\n\tadd %1\n\twrw %c\n" 3
reg:   ADDP1(reg,con)   " # ADDP1(reg,con)\n\trdw %0\n\taddi %1\n\twrw %c\n" 3
reg:   ADDP1(reg,reg)   " # ADDP1(reg,reg)\n\trdw %0\n\tadd %1\n\twrw %c\n" 3
reg:   SUBI1(reg,reg)   " # SUBI1(reg,reg)\n\trdw %0\n\tsub %1\n\twrw %c\n" 3
reg:   SUBP1(reg,reg)   " # SUBP1(reg,reg)\n\trdw %0\n\tsub %1\n\twrw %c\n" 3
reg:   SUBU1(reg,reg)   " # SUBU1(reg,reg)\n\trdw %0\n\tsub %1\n\twrw %c\n" 3

reg:   BANDI1(reg,con)   " # BANDI1(reg,con)\n\tldi %1\n\tand %0\n\twrw %c\n" 3
reg:   BANDI1(reg,reg)   " # BANDI1(reg,reg)\n\trdw %0\n\tand %1\n\twrw %c\n" 3
reg:   BANDU1(reg,con)   " # BANDU1(reg,con)\n\tldi %1\n\tand %0\n\twrw %c\n" 3
reg:   BANDU1(reg,reg)   " # BANDU1(reg,reg)\n\trdw %0\n\tand %1\n\twrw %c\n" 3
reg:   BORI1(reg,reg)   " # BORI1 \n\tldi -1\n\txor %0\n\twrw %c\n\tldi -1\n\
\txor %1\n\tand %c\n\twrw %c\n\tldi -1\n\txor %c\n\twrw %c\n" 10
reg:   BORU1(reg,reg)   " # BORU1 \n\tldi -1\n\txor %0\n\twrw %c\n\tldi -1\n\
\txor %1\n\tand %c\n\twrw %c\n\tldi -1\n\txor %c\n\twrw %c\n" 10
reg:   BXORI1(reg,con)   " # BXORI1(reg,con)\n\tldi %1\n\txor %0\n\twrw %c\n" 3
reg:   BXORI1(reg,reg)   " # BXORI1(reg,reg)\n\trdw %0\n\txor %1\n\twrw %c\n" 3
reg:   BXORU1(reg,con)   " # BXORU1(reg,con)\n\tldi %1\n\txor %0\n\twrw %c\n" 3
reg:   BXORU1(reg,reg)   " # BXORU1(reg,reg)\n\trdw %0\n\txor %1\n\twrw %c\n" 3
reg:   BCOMI1(reg)   " # NOT\n\tldi -1\n\txor %0\n\twrw %c\n" 3
reg:   BCOMU1(reg)   " # NOT\n\tldi -1\n\txor %0\n\twrw %c\n" 3
reg:   NEGI1(reg)   " # NEG\n\tldi -1\n\txor %0\n\taddi 1\n\twrw %c\n" 4

con5:   CNSTI1   "%a"   range(a, 0, 32)
con5:   CNSTU1   "%a"   range(a, 0, 32)
reg5:   con5      " # reg5: con5\n\tldi %0\n\twrw %c\n"   2
reg5:   reg   "%a"
reg:   LSHI1(reg,reg5)   "# SHL\n" 8
reg:   LSHU1(reg,reg5)   "# SHL\n" 8
reg:   RSHI1(reg,reg5)   "# SHR\n" 8
reg:   RSHU1(reg,reg5)   "# SHR\n" 8

con1:   CNSTI1   "%a"   range(a, 1, 1)
reg:   LSHI1(reg,con1)   " # SHL 1\n\trdw %0\n\tshft -1\n\twrw %c\n" 3
reg:   RSHI1(reg,con1)   " # SHR 1\n\trdw %0\n\tshft 1\n\twrw %c\n" 3

reg:   MULI1(reg,reg)   "# call _mul\n" 1
reg:   MULU1(reg,reg)   "# call _umul\n" 1
reg:   DIVI1(reg,reg)   "# call _div\n" 1
reg:   DIVU1(reg,reg)   "# call _udiv\n" 1
reg:   MODI1(reg,reg)   "# call _mod\n" 1
reg:   MODU1(reg,reg)   "# call _umod\n" 1

reg:   CVII1(reg)   "# extend\n"   move(a)
reg:   CVIU1(reg)   "# extend\n"   move(a)
reg:   CVUI1(reg)   "# extend\n"   move(a)
```

```
reg:   CVUU1(reg)   "# extend\n"   move(a)
reg:   CVPU1(reg)   "# extend\n"   move(a)
reg:   CVUP1(reg)   "# extend\n"   move(a)
reg:   CVPP1(reg)   "# extend\n"   move(a)

addr:   reg   "%0"   1
stmt:   ASGNI1(addrg,reg)   " # ASGNI1(addrg,reg)\n\trdw %0\n\twrw RB\n\trdw %1\n\twrwb\n"   5
stmt:   ASGNI1(addrg,con)   " # ASGNI1(addrg,con)\n\trdw %0\n\twrw RB\n\tldi %1\n\twrwb\n"   5
stmt:   ASGNI1(addr,reg)   " # ASGNI1(addr,reg)\n\trdw %0\n\twrw RB\n\trdw %1\n\twrwb\n"   4
stmt:   ASGNU1(addr,reg)   " # ASGNU1(addr,reg)\n\trdw %0\n\twrw RB\n\trdw %1\n\twrwb\n"   4
stmt:   ASGNP1(addr,reg)   " # ASGNP1(addr,reg)\n\trdw %0\n\twrw RB\n\trdw %1\n\twrwb\n"   4
stmt:   ASGNI1(addr,con)   " # ASGNI1(addr,con)\n\trdw %0\n\twrw RB\n\tldi %1\n\twrwb\n"   4
stmt:   ASGNU1(addr,con)   " # ASGNU1(addr,con)\n\trdw %0\n\twrw RB\n\tldi %1\n\twrwb\n"   4
stmt:   ASGNP1(addr,con)   " # ASGNP1(addr,con)\n\trdw %0\n\twrw RB\n\tldi %1\n\twrwb\n"   4

stmt:   ASGNI1(ADDP1(INDIRP1(addrg),con),con) " # ASGN(ADD(INDIR(addrg),con),con)\n\
\twrw RB\n\trdwb\n\trdwb\n\taddi %1\n\twrw RB\n\tldi %2\n\twrwb\n" 7
stmt:   ASGNI1(ADDP1(addrg,con),con) " # ASGN(ADD(addrg,con),con)\n\taddi %1\n\twrw RB\n\tldi %2\n\twrwb\n" 4

stmt:   ARGI1(reg)   " # ARGI1(reg)\n\trdw R12\n\taddi -1\n\twrw R12\n\twrw RB\n\trdw %0\n\twrwb\n"   6
stmt:   ARGU1(reg)   " # ARGU1(reg)\n\trdw R12\n\taddi -1\n\twrw R12\n\twrw RB\n\trdw %0\n\twrwb\n"   6
stmt:   ARGP1(reg)   " # ARGP1(reg)\n\trdw R12\n\taddi -1\n\twrw R12\n\twrw RB\n\trdw %0\n\twrwb\n"   6
stmt:   ARGI1(con)   " # ARGI1(con)\n\trdw R12\n\taddi -1\n\twrw R12\n\twrw RB\n\tldi %0\n\twrwb\n"   6
stmt:   ARGU1(con)   " # ARGU1(con)\n\trdw R12\n\taddi -1\n\twrw R12\n\twrw RB\n\tldi %0\n\twrwb\n"   6
stmt:   ARGP1(con)   " # ARGP1(con)\n\trdw R12\n\taddi -1\n\twrw R12\n\twrw RB\n\tldi %0\n\twrwb\n"   6

addrj:   ADDRGP1   "%a"
stmt:   JUMPV(addrj)   " # JUMPV(addrj)\n\tldi 0\n\tbeqi %0\n\tnop\n"   3
stmt:   JUMPV(reg)   " # JUMPV(reg)\n\trdw %0\n\twrw RB\n\tldi 0\n\tbeqi RB\n\tnop\n"   5
stmt:   LABELV   " # LABELV\n%a\tnop\n"

stmt:   EQI1(reg,reg)   " # EQI1\n\trdw %0\n\tsub %1\n\tbeqi %a\n\tnop\n"   3
stmt:   NEI1(reg,reg)   " # NEI1\n\trdw %0\n\tsub %1\n\tbneqi %a\n\tnop\n"   3
stmt:   GEI1(reg,reg)   " # GEI1\n\trdw %0\n\tsub %1\n\twrw RB\n\tldi 0\n\tldih 0x8000\n\tand RB\n\tbeqi %a\n\tnop\n"   8
stmt:   LTI1(reg,reg)   " # LTI1\n\trdw %0\n\tsub %1\n\twrw RB\n\tldi 0\n\tldih 0x8000\n\tand RB\n\tbneqi %a\n\tnop\n"   8
stmt:   GTI1(reg,reg)   " # GTI1\n\trdw %0\n\tsub %1\n\taddi -1\n\twrw RB\n\
\tldi 0\n\tldih 0x8000\n\tand RB\n\tbeqi %a\n\tnop\n"   9
stmt:   LEI1(reg,reg)   " # LEI1\n\trdw %0\n\tsub %1\n\taddi -1\n\twrw RB\n\
\tldi 0\n\tldih 0x8000\n\tand RB\n\tbneqi %a\n\tnop\n"   9
stmt:   EQU1(reg,reg)   " # EQU1\n\trdw %0\n\tsub %1\n\tbeqi %a\n\tnop\n"   3
stmt:   NEU1(reg,reg)   " # NEU1\n\trdw %0\n\tsub %1\n\tbneqi %a\n\tnop\n"   3
stmt:   GEU1(reg,reg)   " # GEU1\n\trdw %0\n\tsub %1\n\tldi 1\n\tand RC\n\tbneqi %a\n\tnop\n"   6
stmt:   LTU1(reg,reg)   " # LTU1\n\trdw %0\n\tsub %1\n\tldi 1\n\tand RC\n\tbeqi %a\n\tnop\n"   6
stmt:   GTU1(reg,reg)   "# GTU1\n"   9
stmt:   LEU1(reg,reg)   " # LEU1\n\trdw %0\n\tsub %1\n\tbeqi %a\n\tldi 1\n\tand RC\n\tbeqi %a\n\tnop\n"   7

reg:   CALLI1(reg)   "# call\n" 1
reg:   CALLU1(reg)   "# call\n" 1
reg:   CALLP1(reg)   "# call\n" 1
```

```
stmt:   CALLV(reg)   "# call\n" 1

reg:   CALLI1(addrj)   "# call\n" 1
reg:   CALLU1(addrj)   "# call\n" 1
reg:   CALLP1(addrj)   "# call\n" 1
stmt:   CALLV(addrj)   "# call\n" 1
stmt:   CALLASM(addrj)   " # ASM\n%0\n"   1

stmt:   RETI1(reg)   "# ret\n"
stmt:   RETU1(reg)   "# ret\n"
stmt:   RETP1(reg)   "# ret\n"
%%
static void progbeg(int argc, char *argv[]) {
   int i, noret = getenv("NORET") ? 1 : 0;
   parseflags(argc, argv);
   for (i = 0; i < 10; i++)
      intreg[i] = mkreg("R%d", i+1, 1, IREG);
   intregw = mkwildcard(intreg);

   tmask[IREG] = 0x003e; /* R1-R5 */
   vmask[IREG] = 0x07c0; /* R6-R10 */
   cseg = 0;
   doasm = 1;
   print("# VERSAT@lcc (IST: gcrs 2019)\n");
   print("# R12 = SP; R11 = FP\n");
   print("\tldi 1\n\twrw R12\n\tldi ADDR_W\n\taddi -1\n\twrw RB\n"
      "_next\trdw RB\n\tbeqi _top\n\trdw R12\n\tshft -1\n\twrw R12\n\trdw RB\n"
      "\taddi -1\n\twrw RB\n\tldi 0\n\tbeqi _next\n_top\trdw R12\n\taddi -1\n"
      "\taddi MEM_BASE\n\twrw R12\n");
   print("\twrw RB\n\tldi end\n\twrwb\n\tldi 0\n\twrw R11 #FP=0\n\tbeqi main\n"
   "\tnop\nend\t");
   if (!noret) print("ldi 0xF\n\tand R1\n\taddi 0x30\n\twrw CPRT_BASE\n"
      "\tldi 0xa\n\twrw CPRT_BASE\n\t");
   print("ldi 1\n\twrw R12\n\tldi ADDR_W\n\twrw RB\n_again\trdw RB\n"
      "\tbeqi _trap\n\trdw R12\n\tshft -1\n\twrw R12\n\trdw RB\n\taddi -1\n"
      "\twrw RB\n\tldi 0\n\tbeqi _again\n_trap\trdw R12\n\taddi -1\n"
      "\taddi MEM_BASE\n\twrw RB\n\twrwb\n");
}
static Symbol rmap(int opk) {
   switch (optype(opk)) {
   case B:   case P:   case I:   case U:
      return intregw;
   default:
      return 0;
   }
}
static void segment(int n) {
   if (n == cseg)
      return;
```

```
   cseg = n;
   if (cseg == CODE)
      print("# TEXT\n");
   else if (cseg == DATA)
      print("# DATA\n");
   else if (cseg == LIT)
      print("# RODATA\n");
   else if (cseg == BSS)
      print("# BSS\n");
}
static void progend(void) {
   print("_end\t.memset 0\n"); /* for malloc */
   print("### The end ###\n");
}

/* CALL e RET no 'R1'(ACC) */
static void target(Node p) {
   assert(p);
   switch (specific(p->op)) {
   case CALL+I:   case CALL+U:   case CALL+P:
      setreg(p, intreg[ACC]);
      break;
   case RET+I:   case RET+U:   case RET+P:
      rtarget(p, 0, intreg[ACC]);
      break;
   /*
   case DIV+I: case DIV+U:
   case MOD+I: case MOD+U:
   case MUL+I: case MUL+U:
      setreg(p, intreg[ACC]);
      rtarget(p, 0, intreg[ACC]);
      rtarget(p, 1, intreg[ACC+1]);
      break;
   */
   }
}

/* spill the clobbered registers before the instruction */
static void clobber(Node p) {
   static int nstack = 0;

   assert(p);
   nstack = ckstack(p, nstack);
   switch (specific(p->op)) {
   case ASGN+B:
      spill(ACC|(ACC+1), IREG, p);
      break;
   case CALL+I:   case CALL+U:   case CALL+P:
      spill(ACC, IREG, p);
```

```
   }
}
static void emit2(Node p) {
   int op = specific(p->op);
   /* fprintf(stderr, "%d: specific = %d generic = %d (%d)\n", p->op, op, generic(op), CNST+I); */
   if (generic(op) == CNST) {
      int reg = getregnum(p);
      print("\tldi 0x%x\n\tldih 0x%x\n\twrw R%d\n", p->syms[0]->u.value & 0xFFFFF, p->syms[0]->u.c.v.u >> 16, reg);
   }
   else if (generic(op) == LOAD || generic(op) == CVI || generic(op) == CVU || generic(op) == CVP) {
      char *dst = intreg[getregnum(p)]->x.name;
      char *src = intreg[getregnum(p->x.kids[0])]->x.name;
      int a = getregnum(p->x.kids[0]); /* %0 */
      int c = getregnum(p); /* %c */
      assert(opsize(p->op) <= opsize(p->x.kids[0]->op));
      if (dst != src)
         print("# MOV R%d,R%d\n\trdw R%d\n\twrw R%d\n", c, a, a, c);
      if (a != c)
         print("# MOV %s,%s\n\trdw %s\n\twrw %s\n", dst, src, src, dst);
   }
   else if (generic(op) == CALL) {
      int l = genlabel(1);
      char *n = p->syms[0]->x.name; /* %a */
      print("# CALL\n\trdw R12\n\taddi -1\n\twrw R12\n\twrw RB\n\tldi L%d\n\twrwb\n", l);
      if (p->kids[0]->syms[0]) {
         char *f = p->kids[0]->syms[0]->x.name; /* %0 */
         print("\tldi 0\n\tbeqi %s\nL%d\tnop\n", f, l);
      } else {
         char *r = p->kids[0]->syms[2]->name; /* %0 */
         print("\trdw %s\n\twrw RB\n\tldi 0\n\tbeq RB\nL%d\tnop\n", r, l);
      }
      if (p->syms[0]->u.c.v.i > 0) print("\trdw R12\n\taddi %s\n\twrw R12\n", n);
   }
   else if (generic(op) == LSH) {
      int l = genlabel(2);
      int a = getregnum(p->x.kids[0]); /* %0 */
      int b = getregnum(p->x.kids[1]); /* %1 */
      int c = getregnum(p); /* %c */
      print("# SHL\n\trdw R%d\n\twrw R%d\n\trdw R%d\n\twrw RB\n\tbeqi L%d\n\tnop\nL%d\trdw R%d\n"
            "\tshft -1\n\twrw R%d\n\trdw RB\n\taddi -1\n\twrw RB\n\tbneqi L%d\nL%d\tnop\n",
a, c, b, l+1, l, c, c, l, l+1);
   }
   else if (generic(op) == GT) {
      int l = genlabel(1);
      int a = getregnum(p->x.kids[0]);
      int b = getregnum(p->x.kids[1]);
      char *n = p->syms[0]->x.name; /* %a */
      print("# GTU1\n\trdw R%d\n\tsub R%d\n\tbeqi L%d\n\tldi 1\n\tand RC\n\tbeqi L%d\n\tldi 0\n\tbeqi %s\nL%d\tnop\n",
            a, b, l, l, n, l);
```

```
   }
   else if (generic(op) == RSH) {
      int l = genlabel(2);
      int a = getregnum(p->x.kids[0]);
      int b = getregnum(p->x.kids[1]);
      int c = getregnum(p);
      print("# SHR\n\trdw R%d\n\twrw R%d\n\trdw R%d\n\twrw RB\n\tbeqi L%d\n\tnop\nL%d\trdw R%d\n"
            "\tshft 1\n\twrw R%d\n\trdw RB\n\taddi -1\n\twrw RB\n\tbneqi L%d\nL%d\tnop\n",
a, c, b, l+1, l, c, c, l, l+1);
   }
   else if (generic(op) == MUL || generic(op) == DIV || generic(op) == MOD) {
      int l = genlabel(1);
      int a = getregnum(p->x.kids[0]);
      int b = getregnum(p->x.kids[1]);
      int c = getregnum(p);
      char *opc = (op & 3) == 2 ? "_umul" : "_mul";
      if (generic(op) == DIV) opc = (op & 3) == 2 ? "_udiv" : "_div";
      if (generic(op) == MOD) opc = (op & 3) == 2 ? "_umod" : "_mod";
      print("# %s\n\trdw R12\n\taddi -1\n\twrw RB\n\trdw R%d\n\twrwb\n\trdw R12\n\taddi -2\n"
            "\twrw RB\n\trdw R%d\n\twrwb\n\trdw R12\n\taddi -3\n\twrw R12\n\twrw RB\n\tldi L%d\n"
            "\twrwb\n\tldi 0\n\tbeqi %s\nL%d\trdw R12\n\taddi 2\n\twrw R12\n\trdw R1\n\twrw R%d\n",
            opc, b, a, l, opc, l, c);
   }
   else if (op == ASGN+B) {
      int a = getregnum(p->x.kids[0]);
      int b = getregnum(p->x.kids[1]);
      print("## ASGNB=%d ## R%d -> #%d!%d -> R%d\n", p->op, a, p->syms[0]->u.c.v.i, p->syms[1]->u.c.v.i, b);
      blkloop(a, 0, b, 0, p->syms[0]->u.c.v.i, blkregs);
   }
   else if (op == ARG+B)
      print("## ARGB ##=%d\n", p->op);
   else if (generic(op) == RET)
      print("## RET=%d ##\n", p->op);
   else if (generic(op) == ASGN)
      print("## ASGN=%d ##\n", p->op);
   else if (generic(op) == INDIR)
      print("## INDIR=%d ##\n", p->op);
   else
      print("## emit2=%d ##\n", p->op);
}
static void doarg(Node p) {
   assert(p && p->syms[0]);
   mkactual(1, p->syms[0]->u.c.v.i);
}
static void blkfetch(int size, int off, int reg, int tmp) {
   if (size != 1) print("## blkfetch\n");
   else print("# blkfetch\n\trdw R%d\n\taddi %d\n\trdwb\n\trdwb\n\twrw R%d\n", reg, off, tmp);
}
static void blkstore(int size, int off, int reg, int tmp) {
```

```
   if (size != 1) print("## blkstore\n");
   else print("# blkstore\n\trdw R%d\n\taddi %d\n\twrw RB\n\trdw R%d\n\twrwb\n", reg, off, tmp);
}
static void blkloop(int dreg, int doff, int sreg, int soff, int size, int tmps[]) {
   int l = genlabel(2); /* use l and l+1 */
   print("## blkloop: copy R%d words from R%d to R%d (uses R%d to store value)\n", tmps[0], sreg, dreg, tmps[1]);
   if (soff) print("\trdw R%d\n\taddi %d\n\twrw R%d\n", sreg, soff, sreg); /* add offset to src */
   if (doff) print("\trdw R%d\n\taddi %d\n\twrw R%d\n", dreg, doff, dreg); /* add offset to dst */
   print("\tldi %d\n\twrw R%d\n", size, tmps[0]); /* init counter */
   print("L%d\trdw R%d\n\tbeqi L%d\n\twrw R%d # beqi decrements RA\n", l, tmps[0], l+1, tmps[0]);
   print("\trdw R%d\n\twrw RB\n\taddi 1\n\twrw R%d\n\trdwb\n\trdwb\n\twrw R%d\n", sreg, sreg, tmps[1]);
   print("\trdw R%d\n\twrw RB\n\taddi 1\n\twrw R%d\n\trdw R%d\n\twrwb\n", dreg, dreg, tmps[1]);
   print("\tldi 0\n\tbeqi L%d\nL%d\tnop\n", l, l+1);
}
static void local(Symbol p) {
   if (askregvar(p, (*IR->x.rmap)(ttob(p->type))) == 0)
      mkauto(p);
}
static void function(Symbol f, Symbol caller[], Symbol callee[], int ncalls) {
   int i, nargs = 0, dont_opt = 0, nofp = 0;

   for (i = 0; invalid[i]; i++)
      if (!strcmp(invalid[i], f->x.name))
         error("'%s' can not be used as a function name, in the versat backend\n", f->x.name);

   usedmask[0] = usedmask[1] = 0;
   freemask[0] = freemask[1] = ~(unsigned)0;
   offset = 1 + 1;
   for (i = 0; callee[i]; i++) {
      Symbol p = callee[i];
      Symbol q = caller[i];
      assert(q);
      p->x.offset = q->x.offset = offset;
      p->x.name = q->x.name = stringf("%d", offset);
      p->sclass = q->sclass = AUTO;
      offset += roundup(q->type->size, 1);
      if (isstruct(p->type))
         dont_opt = 1;
   }
   nargs = i;
   assert(caller[i] == 0);
   dont_opt |= variadic(f->type)
      || (i > 0 && strcmp(callee[i-1]->name, "va_alist") == 0)
      || isstruct(freturn(f->type));
   offset = maxoffset = 0;
   gencode(caller, callee);

   print("# %s ncalls=%d nargs=%d used=%x\n", f->x.name, ncalls, nargs, usedmask[IREG]);
   if (!strcmp(f->x.name, "main") && getenv("NOSAV")) {
```

```
      usedmask[IREG] = 0;
      print("# NOSAV in main\n");
   }
   print("%s\tnop\n", f->x.name);
   offset = 1 + 1;
   for (i = 2; i <= 10; i++) /* Don't save R1 */
      if (usedmask[IREG]&(1<<i)) {
         print("# PUSH r%d\n\trdw R12\n\taddi -1\n\twrw R12\n\twrw RB\n\trdw R%d\n\twrwb\n", i, i);
         offset += 1;
      }

   for (i = 0; callee[i]; i++) {
      Symbol p = callee[i];
      Symbol q = caller[i];
      assert(q);
      p->x.offset = q->x.offset = offset;
      p->x.name = q->x.name = stringf("%d", offset);
      offset += roundup(q->type->size, 1);
   }
   framesize = roundup(maxoffset, 1);
   if (!nargs && !framesize) nofp = 1;
   if (!nofp)
      print("# save fp: PUSH fp; MOV fp, sp\n\trdw R12\n\taddi -1\n\twrw R12\n\twrw RB\n"
            "\trdw R11\n\twrwb\n\trdw R12\n\twrw R11\n");
   if (framesize > 0) {
      print("# alloc var space: SUB sp, %d\n\trdw R12\n\taddi %d\n\twrw R12\n", framesize, -framesize);
   }
   print("# %s {begin} framesize=%d nofp=%d\n", f->x.name, framesize, nofp);
   emitcode();
   print("# %s {end}\n", f->x.name);
   if (!nofp)
      print("# restore fp: MOV sp, fp; POP fp\n\trdw R11\n\twrw R12\n\twrw RB\n\trdwb\n\trdwb\n"
            "\twrw R11\n\trdw R12\n\taddi 1\n\twrw R12\n");
   for (i = 10; i >= 2; i--)
      if (usedmask[IREG]&(1<<i))
         print("# POP r%d\n\trdw R12\n\twrw RB\n\trdwb\n\trdwb\n\twrw R%d\n\trdw R12\n\taddi 1\n\twrw R12\n", i, i);
   print("# RET\n\trdw R12\n\twrw RB\n\trdwb\n\trdwb\n\twrw RB\n\trdw R12\n\taddi 1\n\twrw R12\n"
         "\tldi 0\n\tbeq RB\n\tnop\n\n");
}
static void defsymbol(Symbol p) {
   if (p->scope >= LOCAL && p->sclass == STATIC)
      p->x.name = stringf("L%d", genlabel(1));
   else if (p->generated)
      p->x.name = stringf("L%s", p->name);
   else if (p->scope == GLOBAL || p->sclass == EXTERN)
      p->x.name = stringf("%s", p->name);
   else
      p->x.name = p->name;
}
```

```
static void address(Symbol q, Symbol p, long n) {
   if (p->scope == GLOBAL
   || p->sclass == STATIC || p->sclass == EXTERN) {
      q->x.name = p->x.name;
      if (n > 0)
         q->x.name = stringf("%s\n\taddi %D", p->x.name, n);
   } else {
      assert(n <= INT_MAX && n >= INT_MIN);
      q->x.offset = p->x.offset + n;
      q->x.name = stringd(q->x.offset);
   }
}
static void defconst(int suffix, int size, Value v) {
   if (suffix == I || suffix == U || suffix == P)
      print("\t.memset %u\n", v.u); /* va does not sign extend */
   else {
      print("\t.memset %g\n", v.d);
      error("'%g' floating point not supported.\n", v.d);
   }
}
static void defaddress(Symbol p) {
   print("\t.memset %s\n", p->x.name);
}
static void defstring(int n, char *str) {
   char *s;

   for (s = str; s < str + n; s++)
      print("\t.memset 0x%x\n", *s);
}
static void export(Symbol p) {
   print("# global %s\n", p->x.name);
}
static void import(Symbol p) {
   if (p->ref > 0) {
      print("# extern %s\n", p->x.name);
   }
}
static void global(Symbol p) {
   int i;
   for (i = 0; invalid[i]; i++)
      if (!strcmp(invalid[i], p->x.name))
         error("'%s' can not be used as a name, in the versat backend\n", p->x.name);
   print("%s", p->x.name);
   if (p->u.seg == BSS)
      for (i = 0; i < p->type->size; i++)
         print("\t.memset 0x00\n");
}
static void space(int n) {
   int i;
```

```
   if (cseg != BSS)
      for (i = 0; i < n; i++)
         print("\t.memset 0x00\n");
}
Interface versatIR = {
   1, 1, 0, /* char */
   1, 1, 0, /* short */
   1, 1, 0, /* int */
   1, 1, 0, /* long */
   2, 1, 0, /* long long */
   1, 1, 1, /* float */
   2, 1, 1, /* double */
   2, 1, 1, /* long double */
   1, 1, 0, /* T * */
   0, 1, 0, /* struct; so that ARGB keeps stack aligned */
   1,   /* little_endian */
   1,   /* mulops_calls */
   0,   /* wants_callb */
   0,   /* wants_argb */
   0,   /* left_to_right */
   0,   /* wants_dag */
   0,   /* unsigned_char */
   address,
   blockbeg,
   blockend,
   defaddress,
   defconst,
   defstring,
   defsymbol,
   emit,
   export,
   function,
   gen,
   global,
   import,
   local,
   progbeg,
   progend,
   segment,
   space,
   0, 0, 0, 0, 0, 0, 0,
   { 1, rmap,
      blkfetch, blkstore, blkloop,
      _label,
      _rule,
      _nts,
      _kids,
      _string,
      _templates,
```

```
      _isinstruction,
      _ntname,
      emit2,
      doarg,
      target,
      clobber,
   }
};
static char rcsid[] = "$Id: versat.md,v 1.05 2019/09/10 22:07:46 gcrs Exp $";
```

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 2 edition, 2006.

[2] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[3] The Programming Language B. S. johnson and b. kernighan. Technical report, Bell Labs, Murray Hill NJ, USA, January 1973. Technical Report CS TR 8.

[4] Doug Brown, John Levine, and Tony Mason. *lex & yacc*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1995 edition, 1995.

[5] P. Cao, B. Liu, J. Yang, J. Yang, M. Zhang, and L. Shi. Context management scheme optimization of coarse-grained reconfigurable architecture for multimedia applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(8):2321–2331, Aug 2017.

[6] Salvatore M. Carta, Danilo Pani, and Luigi Raffo. Reconfigurable coprocessor for multimedia application domain. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 44(1-2):135–152, 2006.

[7] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on CGRAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3):21:1–21:25, September 2014.

[8] Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1 edition, 2003.

[9] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. Ramp: Resource-aware mapping for CGRAs. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 127:1–127:6, New York, NY, USA, 2018. ACM.

[10] J.T. de Sousa, V.M.G. Martins, N.C.C. Lourenco, A.M.D. Santos, and N.G. do Rosario Ribeiro. Reconfigurable coprocessor architecture template for nested loops and programming tool, September 25 2012. US Patent 8,276,120.

[11] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. Coarse-grained reconfigurable array architectures. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 449–484. Springer US, 2010.

[12] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. *Coarse-Grained Reconfigurable Array Architectures*, pages 449–484. Springer US, Boston, MA, 2010.

[13] Charles Donnelly and Richard M. Stallman. *Bison Manual for Version 1.875*. GNU Press, 59 Temple Street, Suite 330, Boston, MA 02111-1307, 8 edition, 2003.

[14] Sandeep Dutta. Anatomy of a compiler. *Cirtcuit Cellar Magazine*, 121:30–35, August 2000.

[15] Christopher W. Fraser and David R. Hanson. A code generation interface for ansi c. *Softw. Pract. Exper.*, 21(9):963–988, August 1991.

[16] Christopher W. Fraser and David R. Hanson. The lcc 4.x code-generation interface. Technical Report MSR-TR-2001-64, Microsoft Inc., July 2001. The report posted here is a corrected version posted in June 2003.

[17] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, September 1992.

[18] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg: Fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, April 1992.

[19] J. Gu, S. Yin, L. liu, and S. Wei. Stress-aware loops mapping on CGRAs with dynamic multi-map reconfiguration. *IEEE Transactions on Parallel and Distributed Systems*, 29(9):2105–2120, Sep. 2018.

[20] David R. Hanson and Christopher W. Fraser. *A Retargetable C Compiler: Design and Implementation*. John Wiley and Sons, New York, NY, USA, 1 edition, 1995.

[21] Reiner Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 564–570, New York, NY, USA, 2001. ACM.

[22] S. C. Johnson. A portable compiler: Theory and practice. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 97–104, New York, NY, USA, 1978. ACM.

[23] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Reinhart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report July 31, 1978.

[24] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[25] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. *SIGPLAN Not.*, 53(4):296–311, June 2018.

[26] Chris Lattner. *LLVM: An Infrastructure for Multi-Stage Optimization*. PhD thesis, University of Illinois at Urbana-Champaign, Urban, December 2002.

[27] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Palo Alto, California, 2004. IEEE Computer Society.

[28] Hochan Lee, Mansureh S. Moghaddam, Dongkwan Suh, and Bernhard Egger. Improving energy efficiency of coarse-grain reconfigurable arrays through modulo schedule compression/decompression. *ACM Trans. Archit. Code Optim.*, 15(1):1:1–1:26, March 2018.

[29] Michael E. Lesk and Eric Schmidt. Lex: A lexical analyzer generator. In *UNIX Programmer's Manual*, volume 2, pages 388–400. Holt, Reinhart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report in 1975.

[30] L. Liu, D. Wang, M. Zhu, Y. Wang, S. Yin, P. Cao, J. Yang, and S. Wei. An energy-efficient coarse-grained reconfigurable processing unit for multiple-standard video decoding. *IEEE Transactions on Multimedia*, 17(10):1706–1720, Oct 2015.

[31] J.D. Lopes and J.T. de Sousa. Versat, a minimal coarse-grain reconfigurable array. In *Proc. of the 12th Int. Meeting on High Performance Computing for Computational Science*, VECPAR, Porto, Portugal, June 2016.

[32] João Dias Lopes. Versat, a compiler-friendly reconfigurable processor-architecture. Master's thesis, Instituto Superior Técnico, 2017.

[33] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10296–, Washington, DC, USA, 2003. IEEE Computer Society.

[34] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1 edition, 1997.

[35] M. Mukherjee, A. Fell, and A. Guha. Dfgentool: A dataflow graph generation tool for coarse grain reconfigurable architectures. In *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*, pages 67–72, Jan 2017.

[36] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 1 edition, 2007.

[37] Thomas W. Parsons. *Introduction to Compiler Construction*. W. H. Freeman, 41 Madison Avenue, New York, NY 10010, 1 edition, 1992.

[38] Todd Proebsting. Burg, iburg, wburg, gburg: So many trees to rewrite, so little time. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-based Programming*, RULE '02, pages 53–54, New York, NY, USA, 2002. ACM.

[39] Ken Kennedy Randy Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1 edition, 2001.

[40] Rui Santiago. Compilador para a arquitectura reconfigurável versat. Master's thesis, Instituto Superior Técnico, 2016.

[41] Rui Santiago, José T. de Sousa, and João D. Lopes. Compiler for the Versat architecture. In *XIII Jornadas de Sistemas Reconfiguráveis*, pages 41–48, January 2017.

[42] Alex T. Schreiner and H. George Friedman. *Introduction to Compiler Construction With Unix*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1 edition, 1985.

[43] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA, 2009.

[44] Richard M. Stallman and GCC DeveloperCommunity. *GCC 8.0 GNU Compiler Collection Internals*. 12th Media Services, Suwanee, GA, 2018.

[45] Andrew S. Tanenbaum, Hans van Staveren, E. G. Keizer, and Johan W. Stevenson. A practical tool kit for making portable compilers. *Commun. ACM*, 26(9):654–660, September 1983.

[46] Justin L Tripp, Jan Frigo, and Paul Graham. A survey of multi-core coarse-grained reconfigurable arrays for embedded applications. *Proc. of HPEC*, 2007.

[47] M. A. A. Tuhin and T. S. Norvell. Compiling parallel applications to coarse-grained reconfigurable architectures. In *2008 Canadian Conference on Electrical and Computer Engineering*, pages 001723–001728, May 2008.

[48] David A. Wheeler. Countering trusting trust through diverse double-compiling (ddc). In *Proceedings of the Twenty-First Annual Computer Security Applications Conference (ACSAC)*, pages 28–40, New York, NY, USA, 2005. ACM.