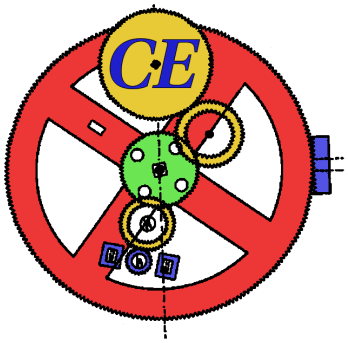

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

Porting the GCC Compiler to a VLIW Vector Processor

Roel Trienekens

CE-MS-2009-19

Abstract

Applications run on embedded DSPs become increasingly complex, while the demands on speed and power continue to grow. One method of meeting these demands is to move some of the processor complexity from hardware to the compiler. This increases the importance of the role of the compiler. This thesis describes how we ported the Gnu Compiler Collection (GCC) to the Embedded Vector Processor (EVP). GCC is a widely used open source compiler framework, which brings several advantages. Its open source nature allows the compiler developer insight into the inner workings and provides the means to change every aspect of the compiler. This allows great freedom in applying compiler optimizations as well as the ability to adapt the compiler to changes in the architecture on short notice. GCC has a large supporting community, delivering quick and accurate feedback and allows the compiler developer to benefit from the improvements contributed by its members. The EVP is a Very Long Instruction Word (VLIW) vector processor, developed at NXP Semiconductors and now property of ST-Ericsson. It is an embedded processor used in mobile communications to handle GSM, UMTS, 3G and 4G standards, amongst others. The goal of this project was to provide proof-of-concept that an EVP back end for GCC can be written that (1) supports all the vector types of the EVP, (2) supports custom operations in the form of the EVP-C intrinsic operations (an extension to the C language designed for the EVP) and (3) that takes advantage of features of the EVP such as post-increment and -decrement addressing, predicated execution and the ability to schedule operations on several different functional units. In this report we describe the implementation of support for vector data types and registers and support for EVP-C intrinsics. The Discrete Finite Automaton instruction scheduler in GCC was adapted to schedule VLIW code. In addition, support was implemented to schedule semantically equivalent operations on different functional units using different instruction mnemonics. For this we devised a new approach that allows us to maintain scheduling freedom while at the same time being able to split the instruction scheduling automaton into several smaller automata in order to avoid excessive build time and compilation time. We tested the compiler using the DejaGnu testing framework, the EEMBC Telecom benchmark suite and a Fast Fourier Transform benchmark which uses EVP-C intrinsics. The experimental results obtained show that the compiler generates correct code of high quality.

Faculty of Electrical Engineering, Mathematics and Computer Science

Porting the GCC Compiler to a VLIW Vector Processor

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Roel Trienekens
born in Gouda, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Porting the GCC Compiler to a VLIW Vector Processor

by Roel Trienekens

Abstract

Applications run on embedded DSPs become increasingly complex, while the demands on speed and power continue to grow. One method of meeting these demands is to move some of the processor complexity from hardware to the compiler. This increases the importance of the role of the compiler. This thesis describes how we ported the Gnu Compiler Collection (GCC) to the Embedded Vector Processor (EVP). GCC is a widely used open source compiler framework, which brings several advantages. Its open source nature allows the compiler developer insight into the inner workings and provides the means to change every aspect of the compiler. This allows great freedom in applying compiler optimizations as well as the ability to adapt the compiler to changes in the architecture on short notice. GCC has a large supporting community, delivering quick and accurate feedback and allows the compiler developer to benefit from the improvements contributed by its members. The EVP is a Very Long Instruction Word (VLIW) vector processor, developed at NXP Semiconductors and now property of ST-Ericsson. It is an embedded processor used in mobile communications to handle GSM, UMTS, 3G and 4G standards, amongst others. The goal of this project was to provide proof-of-concept that an EVP back end for GCC can be written that (1) supports all the vector types of the EVP, (2) supports custom operations in the form of the EVP-C intrinsic operations (an extension to the C language designed for the EVP) and (3) that takes advantage of features of the EVP such as post-increment and -decrement addressing, predicated execution and the ability to schedule operations on several different functional units. In this report we describe the implementation of support for vector data types and registers and support for EVP-C intrinsics. The Discrete Finite Automaton instruction scheduler in GCC was adapted to schedule VLIW code. In addition, support was implemented to schedule semantically equivalent operations on different functional units using different instruction mnemonics. For this we devised a new approach that allows us to maintain scheduling freedom while at the same time being able to split the instruction scheduling automaton into several smaller automata in order to avoid excessive build time and compilation time. We tested the compiler using the DejaGnu testing framework, the EEMBC Telecom benchmark suite and a Fast Fourier Transform benchmark which uses EVP-C intrinsics. The experimental results obtained show that the compiler generates correct code of high quality.

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-19

Committee Members :

Advisor:	Alex Turjan, ST-Ericsson
Chairperson:	Kees Goossens, CE, TU Delft
Member:	Ben Juurlink, CE, TU Delft
Member:	Koen Langendoen, CS, TU Delft

To my family and my girlfriend, for their support and kicking my butt when I needed it the most.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thanks the guys at ST-Ericsson for granting me the opportunity to perform my internship and work on this project, especially Alex Turjan and Dmitry Cheresiz, for all their advice, feedback, patience and help during my time in Eindhoven and Delft.

I would also like to thank Tom de Vries for his feedback and advice during the many lunch breaks at the High Tech Campus.

My thanks also goes out to Ben Juurlink for his help as my supervisor, and to Kees Goossens and Koen Langendoen for taking part in my graduation committee.

Finally I would like to thank Anca Molnos for all the hours she put in proofreading my thesis and her advice for improvements.

Roel Trienekens
Delft, The Netherlands
July 3, 2009

1 Introduction

Runtime performance, power usage and chip area are major concerns in embedded processor design. Minimizing the complexity of the hardware in general improves these metrics. However there tends to be a *conservation of complexity* in computer architecture [1]. This means that in general decreasing the complexity of the hardware results in increased complexity of the compiler. In other words, tasks that the processor cannot handle (stalling the pipeline when a data our resource hazard is detected) must be handled by the compiler. The EVP processor has a *exposed pipeline*, this means that when an instruction is issued that requires a certain hardware block that is already in use by another instruction this instruction can produce undefined results. Therefore the compiler must take care not to issue an instruction whose resource usage conflicts with an instruction issued in the same or a previous clock cycle.

The EVP is a *Very Long Instruction Word* (VLIW) processor, which means it issues multiple instructions in a single clock cycle. The VLIW approach takes advantage of *instruction level parallelism* (ILP) of a program that is run on the processor. The more ILP a program exhibits, the less clock cycles are required to execute the program, thereby improving the runtime performance of the processor. Decreased execution time by exploiting ILP in general results in lower power usage, as the hardware is more efficiently used.

Besides being a VLIW architecture the EVP can also perform operations on *vectors* of data. This exploits the *data level parallelism* (DLP) of executed programs. The general nature of the programs for which the EVP is targeted has a high degree of DLP, for example 3G and 4G protocols which have to process large streams of data [2][3].

Due to the fact that programming VLIW assembly code directly is a difficult task taking many man-hours, programming for the EVP is done using an extension to the standard C language called EVP-C[34]. This extension has intrinsic operations that map almost directly onto assembly instructions, allowing the programmer to code efficiently, while at the same time leaving the difficult tasks of instruction scheduling and register allocation to the compiler. Using EVP-C a programmer will in general be able to code more efficiently than using 'regular' C.

The GNU Compiler Collection (GCC) is an open source compiler framework that is widely used and supports many languages and architectures. GCC consists of a *front-end*, which parses the input code and transforms it into a language independent representation on which the *middle-end* operates a number of language and target independent *optimization passes*, which transform the statements in the intermediate representation into a more optimized stream of instructions. This stream is then processed by the *back end*, which performs optimizations that are specific to the target architecture as well as instructions scheduling, register allocation and assembly output generation.

In this chapter we will briefly discuss the concepts and related work relevant to this project and we will outline the goals of this project. The structure of this chapter is as follows. In section 1.1 we will discuss the background of the VLIW and vector processors and the GCC compiler framework. We will also briefly discuss some of the previous work done regarding these topics. In section 1.2

we will describe the motivation for the work done in this thesis. Section 1.3 will outline the goals we want to reach during the course of this project. In section 1.4 we will outline which parts of this thesis were implemented as part of this thesis project, and which parts were handled by the rest of the team working on the EVP-GCC project during the same time.

1.1 Background and Related Work

In this section we will discuss the background of VLIW processors, vector processors and the GCC compiler, and we will give a brief overview of any previous work related to these concepts.

Very Long Instruction Word processors. The earliest processors were capable of executing only very simple instructions, one at a time. As improvements were made, processors were able to handle increasingly complex instructions which at some point were so complex that designing the control circuits became very difficult. This difficulty was overcome by using more simple instructions called *microcode* which resided in special high-speed memory and were on-the-fly translated into a series of machine instructions. This approach, called *microprogramming* was proposed by M. Wilkes in 1951 [4]. One form of microprogramming is *horizontal microprogramming* which involves issuing multiple *micro-instructions* (the smallest unit of microcode) in parallel. A drawback of microprogramming is the fact that it needed to be hand-coded and was very machine-specific (not portable). Based on this concept A.E. Charlesworth proposed a processor architecture that contained multiple functional units which could each execute separate instructions in 1981 [5]. Based on this, J. Fisher proposed a system which would encode horizontal micro-instructions into one single *Very Large Instruction Word* [6]. Based on the *trace scheduling* technique he had proposed earlier [7] he presented the architecture for the ELI-512, the first VLIW processor. Short for *Enormous Longword Instructions*, it was a processor which at the time of publication used 1200 bits in instruction encoding.

Two companies were started in the mid-1980s, Multiflow by Joseph Fisher amongst others and Cydrome by Bob Rau and colleagues, that both produced several VLIW processors, the first delivered in 1987. These early VLIW processors were not a commercial success and both companies closed in the 90's, although some concepts first introduced then can still be found in modern day VLIW architectures. This decade interest in VLIW architectures has increased again resulting in a number of VLIW processors, amongst which Intel's IA-64 (or Itanium), Texas Instruments' TM320C062x and C6000, the SHARC by Analog Devices, the TriMedia by Philips, ST's ST200 and of course the EVP.

Vector processors. The first work done on vector processors was performed at *Westinghouse Electric Corporation* in the *Solomon* project. The first implementation of this vector processor consisted of a single CPU to which several math co-processors were added. All co-processors were fed a single common instruction, but they operated on different data. This was the first effective implementation of a *Single Instruction Multiple Data* (SIMD) machine. This work was discontinued, but picked up by the University of Illinois, where the *ILLIAC IV* was developed. The ILLIAC used a separate ALU for each data element and because of this is more often referred to as *massively parallel computing* rather than vector processing. Although this processor was not considered a success, it proved that the underlying concept was sound.

The first true vector machines are the *CDC STAR-100* and the *Texas Instruments Advanced Scientific Computer* (ASC). Due to the fact that decoding vector instructions took a long time, these machines did not yet produce the speedup that could theoretically be achieved, partly due to the fact that the data comprising these vectors had to be loaded directly from memory before every vector operation.

Especially because most of the typical workloads operate on the same vector data multiple times. The *Cray-1* developed by Cray Research in 1976 avoided this problem by adding vector registers to the architecture. The Cray-1 was the first processor to do so. This allowed for fast execution of vector instructions.

In many modern-day processors there is at least some capability of performing vector operations. This is mostly implemented through a separate vector processor that runs beside the scalar CPU.

The GNU Compiler Collection.

The GNU (GNU's Not Unix) project was started by Richard Stallman, who sought to create an operating system that would emulate Unix, without having the limitations that copyrighted software brings. For this project Stallman needed a compiler and for this he turned to the Free University Amsterdam which at the time had developed the *Vrije Universiteit Compiler Kit* (VUCK). He was told that 'although the University was free, the compiler was not'. He then obtained the source code for the *Pastel* compiler, a compiler which supported and was written in Pascal. He wrote a C front-end for it, but discovered that the compiler used too much resources for him to effectively port it, and he decided to build his own C compiler from scratch. This compiler was later called the *GNU C Compiler* (GCC) and as new front-ends were added later on was changed to *GNU Compiler Collection* [22], retaining the same acronym. To ensure the fact that the source code of GCC remained free and would allow every user to have the freedom to modify and further distribute it, Stallman founded the *Free Software Foundation* [20] (FSF) in 1985. The FSF is 'owner' of all GNU software, and distributes all the software under a number of licenses, the most important being the *GNU General Public License* [21] (GPL). This license decrees that all software published under the GPL can be freely modified and redistributed as long as any resulting software also falls under the GPL. This availability is known as *free* or *open source* software and the distribution model is known as *copyleft*, a term that emphasizes protecting the freedom of the user instead as opposed to copyright, which emphasizes the limitations of the user while protecting the rights of the original authors.

At the time of this writing, the current release of GCC is version 4.4.0, with development under way on version 4.5.0. Throughout the years a number of different language front-ends have been added, as well as ports for different architectures. In addition, each new release featured a number of changes which improved the performance and compile time of code compiled by GCC, most notably the introduction of the *middle-end* and the Tree-SSA representation [12] used therein. The middle-end contains a number of both language and target independent optimization passes.

GCC was originally designed for single-issue processors with hardware interlocks. This means the compiler emits one assembly instruction per clock cycle. The instruction scheduling takes into account dependencies between different instructions in order to reduce the need for pipeline stalls. However, in case no instruction can be issued, GCC relies on the processor hardware to generate the stalls, and therefore the generated assembly code does not contain any nops.

The most notable VLIW architectures for which a GCC back end was developed are the *Itanium* architecture by Intel [8] and the TMS320-C6000 (C6x) by Texas Instruments [9]. The GCC back end for the C6x was developed at the Faculty of Computer Science at Chemnitz University of Technology by Jan Parthey and Adrian Strätling [10] [11]. Both these architectures do not contain vector registers besides those needed for multimedia extensions.

Since version 3.1 GCC also supports vector types. These were added to support multimedia extensions to certain architectures without the programmer having to resort to using inline assembly code. This

support can also be used to implement regular (not those used for multimedia extensions) vector registers.

1.2 Motivation

GCC is an interesting candidate to target the EVP to for a number of reasons. First of all it is free, as mentioned earlier, which allows full access to the source code and the freedom to modify it any way the compiler developers might like. Second of all GCC sports a large community which is actively developing the compiler, which leads to a large number of ongoing projects to improve the compiler. This also allows easy feedback directly to and from the developers of a particular piece of the compiler which in turn allows a better understanding of the inner workings and ideas behind certain design choices. Third, GCC recently includes support for autovectorization of code, developed by the IBM team at the Haifa Labs [13][14][15][16]. Autovectorization analyzes loops in the code and tries to rewrite them using vectors, allowing a vector processor to profit from it's vector processing capabilities. Fourth, direct access to the source code allows the compiler developer to make changes to the compiler to reflect any changes made to the architecture on short notice. Not only changes in architecture can be handled, but also a lot of design space exploration can be done for upcoming versions of the processor, potentially improving the quality of future versions of the processor.

1.3 Goals

The considerations presented in the previous section have resulted in the development of an EVP back end for GCC (EVP-GCC). At the start of this project EVP-GCC only supported scalar code.

The goal of the project can be summarized as follows:

- The goal of this project is to extend the current version of the EVP-GCC compiler to include support for vector instructions and EVP-C intrinsic operations, and to evaluate the performance relative to the current EVP production compiler.

In order to achieve this goal a number of subgoals can be defined:

- Extend the compiler to support vector data types and vector registers.
- Extend the compiler to support the EVP-C extension to the C language by means of implementing support for a number of EVP-C intrinsic operations. The specific instructions that must be implemented are all EVP-C intrinsics used in the vector benchmark code.
- The EVP supports post-increment and post-decrement load and store operations. Support for generation of these instructions must be implemented in the compiler.
- Some operations can be issued on several different functional units. Support for all alternatives of these instructions must be implemented.
- Enable the autovectorization in GCC to vectorize loops in out-of-the-box benchmark code and measure the performance increase in terms of cycle count compared to non-vectorized code.

- The generated code for the benchmarks should be correct.
- Measure the performance of the generated code for the chosen benchmark.

In order to achieve these goals benchmarks were selected, both for the scalar part and for the vector part. The development of EVP-GCC that led to the compiler as it was at the start of our project had been benchmarked using the *EEMBC-Telecom* suite [26]. This is a set of benchmarks that consists of a number of applications which are relevant to the mobile and telecommunication market, and therefore relevant to the EVP. This suite was also used to test autovectorization of out-of-the-box code. For the vector part we used an FFT application that performed an FFT transformation on a number of data sets.

1.4 Contributions

In this section we will give an overview of which topics detailed in this report were implemented as part of this thesis project. Work done by the other members of the team will also be presented for completeness purposes.

The following topics were implemented as part of the thesis project:

- We added the vector data types for all different vector registers present in the EVP. All vectors used in the benchmark code have a width of 256 bits, so support for any other vector sizes was not implemented as part of this thesis work. The implementation of vector data types and registers is detailed in section 3.1.2.
- Adding vector registers caused problems with misaligned loads and stores in the function prologue and epilogue generation, this is the code generated at the beginning and end of a function to store and load register values that are overwritten in the function on the stack in order to avoid overwriting values that are needed after the function returns. We added support to fix this problem. This is presented in section 3.2.4.
- Support for all vector intrinsics needed to successfully compile the FFT code used to benchmark the vector part of the compiler. This is detailed in section 3.2.1.3.
- We have implemented a novel approach to schedule semantically equivalent instructions on several different functional units, based on the instructions already scheduled in the same cycle. Due to the complexity of the finite state automaton generated, implementing this in the traditional way this is handled in GCC resulted in a state machine that was far too large. Our approach enabled us to split the scheduling automaton into several smaller automata, without reducing the freedom of the scheduler. This was one of the main parts of this thesis and is detailed in section 3.3.4.
- We added support for post-increment and -decrement addressing that is supported by the EVP. This is detailed in section 3.5.2.
- We added verbose comments to the assembly code emitted by the compiler. This allows programmers to more efficiently interpret the assembly code generated as well as the scheduling choices the compiler made. This is described in section 3.4.

- We changed the way register constraints in instruction matching patterns (the rules for allocating a register or class of registers in an instruction) were defined in the existing EVP-GCC compiler to use the `define_constraint` construct instead of defining contraints through macros. This is detailed in section 3.2.1.1.
- Several improvements were made in the branch delay scheduler, detailed in section 3.5.1.
- To ensure correctness of the compiler with the newly added features a number of testcases were added, this is described in section 4.1.
- Many of these additions required defining some new macros in the EVP back end. These macros are detailed in section 3.1.
- The inter-basic block scheduling of the existing EVP-GCC compiler did not take into account resource constraints. The `div_to_end_bb_pass` was added to fix this. This is detailed in section 3.3.7.

The rest of the team working on EVP-GCC also made a number of improvements that are not part of this thesis project, but are discussed briefly in this report for completeness purposes. These are the following:

- Support for hardware loops. This is a method of specifying the beginning, end and number of iterations in a loop in order to avoid having to fill delay slots for each jump back to the beginning of the loop. This is detailed in section 3.5.3.
- Address-based alias analysis was added. This is an approach for alias analysis detailed in [30] that at the moment is not implemented in GCC, but results in better alias analysis which in turn results in code that has better performance. This is described in section 3.5.4.
- The existing if-conversion was improved. If-conversion is a technique where branches resulting from if-statements are replaced by instructions that are always executed but for which the result is only written to a register if the condition register holds a *true* value. Details on this can be found in section 3.5.5.
- Inter-basic block scheduling was enabled. Inter-basic block scheduling is a method of instruction scheduling that takes into account instructions in all basic blocks that are predecessors of the current basic block that is being scheduled. This is detailed in section 3.3.6

This document is split up into five chapters. Chapter 2 gives an overview of the processor architecture, the compiler framework and the EVP-C language extensions. Chapter 3 describes the changes made to the back end to implement all features of the EVP processor. Following this, in chapter 4 we will give the experimental results of correctness benchmarks, and we will present the results of performance measurements of the GCC compiler with the newly implemented scheduler as opposed to GCC without this addition. We will also present the results of our experiment with using the autovectorizer to vectorize code in the EEMBC Telecom benchmark for the AltiVec back end. Due to contractual obligations it is not possible to publish the results of the performance benchmarks compared to the current production compiler used by ST-Ericsson. This information will be supplied separately and is only available to the members of the graduation committee. Chapter 5 will contain conclusions and recommendations on further work to be done, as well as an overview of the impact of the open source nature of GCC on our project.

2 Fundamentals

In this section we will give an overview of the fundamentals of the architecture of the EVP (section 2.1), the additions to the C standard to more efficiently code applications called EVP-C (section 2.2) and a global overview of GCC (section 2.3).

2.1 EVP

The Embedded Vector Processor (EVP) [31][32] is an embedded VLIW vector DSP that was developed inside NXP Semiconductors, and is now property of ST-Ericsson. The EVP was developed to support 3G and 4G standards, for use in SoC solutions for mobile and wireless markets. 3G and 4G standards require the processing of streams of data, and for this the EVP supports multiple operations on vectors with up to 16 elements of up to 40 bits in width. This section will explain the organization of the EVP and its capabilities.

2.1.1 Architecture

This section gives an overview of the architecture of the EVP. First we will describe all functional units and registers of the EVP, followed by the layout of data in vectors and we will discuss the VLIW aspect of the EVP architecture and the assembly syntax.

2.1.1.1 Functional units

An overview of the functional units is given in Figure 2.1. The functional units in the EVP can be divided into four blocks.

PCU: The *Program Control Unit* (`PCU`) handles decoding of the VLIW instructions, takes care of branching and keeps track of the state of hardware loops.

Because the EVP features an exposed (non-interlocked) pipeline, there is no hardware to stall parts of the pipeline. Instead, this task is left to the compiler. The way in which this is handled in GCC is explained in section 3.3. The absence of this interlocking hardware is mainly done for design simplification, but also results in reduced chip area of the EVP, which in turn reduces power usage, an important aspect of embedded processor design. It also The `PCU` uses parts of the `SDCU` to perform conditional execution, in order to support conditional branches and calls.

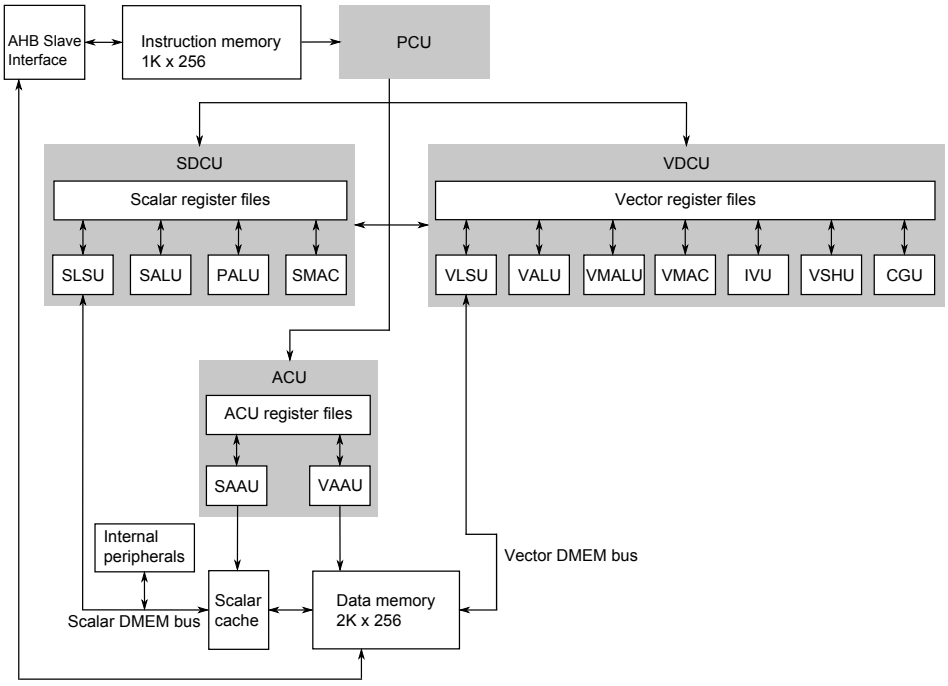

Figure 2.1: EVP functional block diagram [33]

SDCU: The *Scalar Data Computation Unit* (SDCU) executes the scalar instructions. It consists of two register files (for details for all register files, see section 2.1.1.2), a *Scalar Arithmetic Logic Unit* (SALU), a *Predicate Arithmetic Logic Unit* (PALU) which executes logical operations on 1-bit predicate registers. There is also a separate *Scalar Multiply/Accumulate* (SMAC) unit, which handles multiplication and multiply-accumulate operations. Finally there is the *Scalar Load/Store Unit* (SLSU). This unit takes care of all the loads and stores for scalar and predicate registers.

VDCU: The *Vector Data Computation Unit* (VDCU) is dedicated to all operations that involve vectors. It consists of multiple vector register files, a *Vector Arithmetic Logic Unit* (VALU) and *Vector Mask Arithmetic Logic Unit* (VMALU). These last two handle all arithmetic and logical operations on vector and vector mask registers. The *Vector Multiply/Accumulate Unit* (VMAC) handles multiplication with or without accumulation on vector registers. The *Intra Vector Unit* (IVU) takes care of operations on a single vector (taking the maximum of all elements of a vector, for example). The *Vector Shuffle Unit* (VSHU) shuffles the elements of a vector according to either a predetermined pattern or the values inside a vsp register. There is also the *Code Generation Unit* (CGU), which generates application-specific code that is inserted into the instructions stream to improve performance of the core, for example scramble or channelization code such as in the UMTS standard, part of the 3G standard[2]. The VDCU also has a *Vector Load/Store Unit* (VLSU). This unit handles all loads and stores of regular vectors and vector masks to and from the data memory.

ACU: The *Address Calculation Unit* (ACU) performs all calculations on the *pointer* and *offset* registers to compute memory addresses for the VLSU and SLSU.

Also a connection between the SDCU and VDCU exists to enable *broadcasting* of a scalar value to a vector, i.e. a vector containing all identical values equal to the value in a certain scalar register can be used as an operand in a vector operation.

2.1.1.2 Registers

The EVP has several different register files. For the scalar part there is the regular scalar register file. This consists of 32 registers that are 16 bits wide. These can be addressed as single registers, when 8- or 16-bit operands are needed, or as pairs, when there is need for operations on 32-bit data. In this case the pairs need to be two consecutive register of which the lower one is even-numbered. The registers are numbered `r0` through `r31`.

There is also a *predicate register file*, which holds 8 predicate registers, numbered `p0` through `p7`, of which `p0` is always true, and cannot be changed. Predicate registers can be appended to almost all instructions to make their execution conditional on the contents of `pN`.

There are five different vector register files, starting with the *vr register file*, which holds 16 regular data vector registers. These vectors are 256 bits wide and are named `vr0` through `vr15`. Vectors can hold 32 8-bit values, 16 16-bit values or 8 32-bit values. These vectors can also be used together to make 512 bit vectors, which can hold 16 32-bit values, or 640-bit vectors, which holds 16 40-bit values. In these cases, the vectors used need to be consecutive and the lower of the two (or three) must be a multiple of 2 or 4 respectively.

The second vector register file is the *vm register file*. This contains 8 32-bit vector mask registers, `vm0` through `vm7`. A vector mask is used to make a vector operation conditional. If a vector mask register is specified in an instruction, the result of the operation is only written to the destination register if, for each 8-bit field, the corresponding bit in the `vm` register is set. Similar to the scalar predicate registers, the value of `vm0` is fixed and all bits are set to *true*.

The *ir register file* and *im register file* contain registers that hold the results of intra-vector operations on `vr` and `vm` registers respectively. The `ir` registers are 256 bits wide and the `im` registers 64 bits. The `im` registers consist of 32 2-bit fields. Both register files contain 2 registers of their respective type. These registers are named `ir0`, `ir1`, `im0` and `im1`.

The *vsp register file* holds 16 vector shuffle pattern (`vsp`) registers. These registers are 64 bits wide and up to 4 can be used in vector shuffle operations on the `VSHU`.

There are also 4 registers available for the `CGU` unit, but because the work in this project doesn't involve the Code Generation Unit, we will not go into any further detail.

The `ACU` contains three register files. There *pointer register file* holds 16 pointer register, `ptr0` through `ptr15`. The *offset register file* holds 8 *offset* registers. These are named `ofs0` through `ofs7`. There is also a *size register file*, containing the 8 registers `size0` though `size7`. All these registers are 32 bits wide, but since the memory of the EVP only requires 24 bits, any arithmetic on `ptr` or `ofs` registers will cause the most significant 8 bits to be cleared after the operation.

The EVP uses a number of *addressing modes*, that should be supported by the compiler. A value can be loaded from or stored to memory using *direct* addressing, where the address of the memory location is an immediate value in the instruction. It is also possible to address a memory location by a value in a register, this is called *register indirect* addressing. A third method of specifying the address of a memory location is *base* + *offset* addressing. In this mode an instruction contains pointer

and an offset register, whose values will be added to obtain the address, after which the load or store is performed. The offset register can be replaced by an immediate value. Finally the EVP supports *post-increment base + offset* and *post-decrement base + offset* addressing modes. These modes are similar to the base + offset addressing mode but in this case the value in the pointer register is updated with the value in the offset register or the immediate value, whichever is applicable. We will refer to these last two modes as *post-increment* or *post-decrement* addressing.

Finally the `PCU` also contains several registers. The *program counter* (`pc`) keeps track of the address of the next instruction to be fetched, the *return address* (`ra`) register holds the address to where the current function should return to after it finishes execution. Finally there are three additional registers: `lsa`, `lea` and `lc`. These hold the start address, end address and count, respectively, of any hardware loops the program is in. For more information on hardware loops, see section 2.1.2.2.

2.1.1.3 Vectors

As mentioned in section 2.1.1.2 the main registers used in the EVP are 256 bits wide, and can be used in a number of modes. Depending on the size of the elements, the vectors contain 8, 16 or 32 elements. When in 8-bit mode, the `vr` registers hold 32 elements. In case of 16-bit mode, a single `vr` register can hold 16 elements, and when 32-bit mode is used, 8 elements fit in each `vr` register.

The EVP is also capable of storing larger numbers and storing 16 32-bit values in adjacent vector registers. For example, in the case of multiplication of two `vr` registers each containing 16 16-bit numbers, the result is 16 32-bit numbers. This data cannot be stored in a single `vr` register, therefore it is stored in two adjacent `vr` registers. The least significant 16 bits of each of the results are stored in the lower numbered register, while the most significant 16 bits go into to the higher numbered register. A multiply-accumulate operation on two 16-bit and one 32-bit operand can result in a number even larger. For this reason the EVP also supports vectors with 40-bit elements. Bits 0-15 are stored in the lowest numbered vector, bits 16-31 in the middle vector and bits 32-39 in the highest numbered vector, as depicted in Figure 2.2.

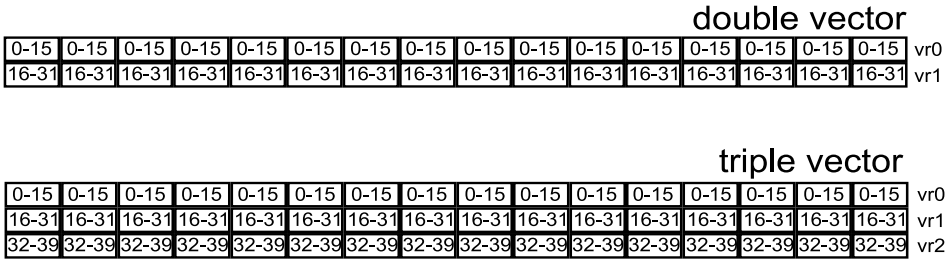

Figure 2.2: Storage of values in double and triple vectors

The EVP supports fixed point numbers as well, in either 1.7, 1.15, 1.31 and 9.31 format. The number before the decimal point signifies the number of bits of the integer part of the fixed point number, and the number after the decimal point signifies the width of the fraction field. The value of a fixed point number is effectively the value of the number when represented as an integer divided by 2^n, where n is the width of the fraction field, e.g. a 1.7 fixed point number is the value of the total eight bits represented as an integer divided by 2^7, and ranges from 0 to $2 - 2^{-7}$. These fixed point numbers are

stored similar to 8-, 16-, 32- and 40-bit integers respectively. 1.31 fixed point numbers can be stored in either one or two vector registers, whereas the 9.31 format is always stored in three vector registers.

The EVP also supports fixed point complex numbers. All fixed point formats are available as complex numbers, from a pair of 1.7 to a pair of 9.31 fixed point numbers. These are stored in the same number of vector registers as their regular fixed point counterparts, only they can hold half the number of elements.

2.1.1.4 VLIW

The EVP is a *Very Large Instruction Word* (VLIW) processor[25]. Multiple independent instructions are packed into one very long instruction word. The compiler handles register allocation and schedules the instruction into VLIW words. It takes care that the functional units required by all instructions in the VLIW word will be free in the cycle that they are needed in. VLIW processors take advantage of *Instruction Level Parallelism* (ILP) in the code. Many target applications for the EVP exhibit a high amount of ILP. It is the compilers job to take maximal advantage of this form of parallelism in the code. The EVP is capable of processing up to 13 different operations every clock cycle, provided that each operation does not use a functional unit that is already in use by a different operation, e.g. a VLIW word has only one *long immediate* field. This is a field that contains a constant value that is coded directly in the assembly code. Therefore two instructions that both use the long immediate field cannot be scheduled in the same VLIW instruction word.

2.1.1.5 Assembly

The assembly code that is generated by the compiler consists of an *opcode* along with up to three *operands* and a *predication operand*. The assembly syntax also allows comments, preceded by `//`, similar to C style comments. To signify an instruction that is to be executed in parallel with the instruction before it, it has to be preceded by `||`.

For more information on how the assembly is emitted see section 3.4.

2.1.2 EVP architecture features

The EVP employs several techniques to exploit parallelism and improve performance. In this section we will briefly discuss the techniques that are important in the scope of this project.

2.1.2.1 Predicated and masked execution

In order to minimize costly branch instructions, the EVP supports *predicated execution*. This is the conditional execution of instructions, by appending a predicate register to the instruction. Using this technique, control dependencies can be changed into data dependencies, which are far less costly due to the large number of delay slots all branch instructions require. The EVP architecture allows predication of all instructions except for jump and branch instructions.

2.1.2.2 Hardware loops

Another technique to avoid extensive generation of branches and their accompanying delay slots is the support of a hardware construct that predefines the boundaries of loops. These *hardware loops* are implemented through use of an instruction that specifies the start and end address of a loop as well as the number of iterations. This allows the processor to fetch the instruction at the beginning of the loop directly after fetching the instruction at the end, without the need for any delay slots. This allows the compiler to exploit the ILP of the code optimally.

2.2 EVP-C

As mentioned before, runtime performance and power usage are two important concerns for embedded processor design. The code that is executed on these processors typically consists of small kernels that are executed many times and supporting code surrounding the kernels that take up only a small percentage of the total execution time. Since this code is typically designed once and not changed much afterwards, it is hand-written and heavily optimized before compilation. This optimization is facilitated by extending the C language with a number of intrinsic functions, that map directly to assembly instructions. This superset of ANSI C is called *EVP-C*[34]. With EVP-C the programming of an algorithm comes close to writing actual EVP assembly code, without the programmer having to take into account aspects like register allocation and scheduling.

To give an example, to execute an addition of two vectors using 16 bit elements the EVP-C statement

```
v3 = evp_vadd_16(v1, v2);
```

maps directly onto the assembly instruction

```
vadd16     vrD, vrA, vrB
```

EVP-C also defines a number of *pragmas*, compiler directives that guide the compiler into more efficient scheduling, register allocation or passing along information that prevents the generation of certain blocks of code (for example to catch cases where the number of loop iterations is zero) which are not needed.

In the FFT benchmark used in this project the following pragmas are used:

```
__DataNoInit
__UnitDefaultSectionData( name )
__UnitDefaultSectionConst( name )
__UnitDefaultSectionNoInit( name )
__UnitDefaultSectionFun( name )
__DataSection( name )
__FunSection( name )
__DataAt( ofs )
__FunReturnAt( ofs )
__M1
__M2
__M3
__M4
__M5
__M6
```

```
__M7
__M8
__LoopNoHwLoop
__LoopNoSWP
__LoopIterMin( n )
__LoopIterMax( n )
__FunHwLoopLevels( l )
__FunHwUseLoopLevels( l )
```

We will discuss the use of those pragmas that are important in the scope of this project. One important pragma is the `DataAt( ... )` pragma, which tells the compiler to store a certain variable in a certain register. It is used in the FFT to mark variables that contain the step size of the data that is loaded from the array containing the input data. Storing this step size in a set of general purpose registers would cause the need for a large number of moves from these registers to an offset register before a pointer value can be updated. To prevent generation of these moves, the pragma `__DataAt(ofs)` is used, which tells the compiler to allocate this variable into an offset register.

Another set of pragmas that improve performance are the `__Loop...` pragmas. These specify that the loop following the pragma has a minimum or maximum number of iterations (thereby eliminating the need to check if the loop variable is non-zero at runtime) or to aid in or disable software pipelining for a specific loop.

The pragmas that bring the most improvement are the memory qualifiers `__M`*n*. These qualifiers tell the compiler that two pointers with different qualifiers can never point to the same memory location. This aids the alias analysis and prevents the generation of unnecessary dependencies between these pointers, thereby improving performance. As an example consider the following code:

```
int *ptrA;
int *ptrB;
int *ptrC;
for (i = 0; i < 10; i++)
{
  *ptrA++ = *ptrB++ + *ptrC++;
  *ptrA++ = *ptrB++ - *ptrC++;
}
```

This loop adds the value to which pointer `ptrB` points to the value at `ptrC` and stores the result at the location pointed to by `ptrA`. Afterwards the pointers are incremented. This process is repeated but now the value at the locations pointed to by (the new values of) `ptrB` and `ptrC` are subtracted. Without any aliasing information, there exists a *read-after-write* dependency between `ptrB` and `ptrA`. The contents of the memory location `ptrB` in the second statement points to could be changed when the value of the previous expression is written to the memory location pointed to by `ptrA` in the first statement (this is the case if `ptrB` is equal to (`ptrA - 1`). This dependency prohibits the two assignments to be scheduled in parallel. Using the EVP-C memory qualifiers the programmer can tell the compiler that these two pointers never point to the same memory location by changing the code listed above to:

```
__M1 int *ptrA;
__M2 int *ptrB;
__M3 int *ptrC;
for (i = 0; i < 10; i++)
{
  *ptrA++ = *ptrB++ + *ptrC++;
  *ptrA++ = *ptrB++ - *ptrC++;
}
```

This information now allows the compiler to schedule these two assignments in parallel while still maintaining correctness, thereby increasing performance of the code.

2.3 The GCC compiler

The entire GCC compiler toolchain, which processes code written in a certain programming language and generates a binary executable file, consists of a number of different stages. Building GCC generates an executable file which executes the following stages consecutively.

Preprocessor
The preprocessor handles the expansion of *preprocessor directives* and macros (`#define`), and inclusion (`#include`) of other files in a source file. The C preprocessor is called `cpp` and generates *preprocessed C sources*.

Compiler
The compiler itself is generated as the executable file `cc1`. This executable processes the intermediate files passed to it by the preprocessor and generates an assembly file.

Assembler
The assembler processes the assembly sources and translates them to binary object code in *object files* that can be directly executed on the target processor architecture. Besides the instructions, the object file can also contain debugging information. The assembler is invoked by executing the `vasm` file.

Linker
In a final step the linker links one or more object files together into an executable. This generated executable is the code executed by the EVP processor. The linker executable in our setup is called `vlink`.

To simulate the executables generated by the compiler toolchain, the in-house simulator `vclsim` was used.

2.3.1 Building the GCC compiler

To build the GCC compiler, we need the GCC sources, which can be downloaded from the GCC website[22]. For this project, we used GCC version 4.3.1, which was released in June, 2008. The GCC sources are placed in the *source directory*, the machine specific files will be stored in the *target directory* and the *build directory* which will contain the GCC-generated files as well as the compiler driver and compiler executable.

To build the compiler executable, GCC will generate a number of header and source files inside the build directory. Among these files are all type declarations as well as a number of supporting macros, the generated source code for the instruction scheduler, functions that recognize RTL instructions, generate the RTL instructions and generate the assembly code.

An overview of which files are used to build the compiler is given in figure 2.3.

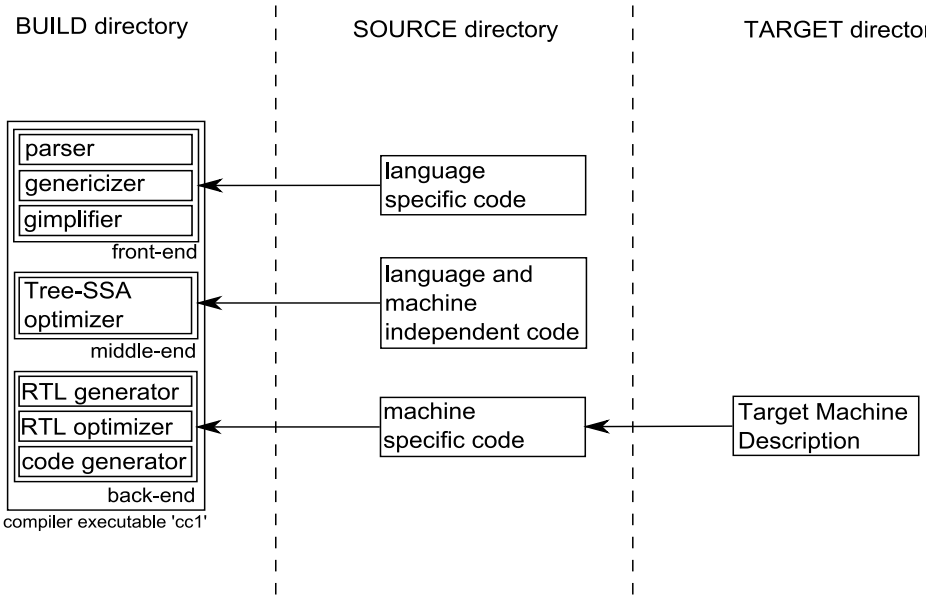

Figure 2.3: Files used in building the GCC compiler executable

2.3.2 Intermediate representations

The GCC compiler consists of several different stages that transform code from one representation to another. All stages the compiler goes through to transform input C source code to assembly code are depicted in figure 2.4. Starting with the source code, the program is transformed into different representations in the following order:

GENERIC
The front-end does the language specific transformation from source code to an intermediate language independent tree representation called *GENERIC*.

GIMPLE
The GENERIC representation is then simplified to the *GIMPLE* representation, which splits complex expressions into multiple smaller expressions. Note that at the moment, the C and C++ front end directly emit GIMPLE trees. As a general rule every GIMPLE expression should have a maximum of three operands. There are some exceptions like function calls and built-in operations.

SSA
Following this, the GIMPLE tree is transformed into the *Static Single Assignment* (SSA) representation[29]. This representation is based on the premise that all variables inside a program are assigned only once. If a variable is assigned a new value, Tree-SSA considers this a new variable. It introduces a new construct called *PHI node*, which is used when the value of variable cannot be determined at compile time (for example following an if-statement). A new variable is then declared and is assigned the value `PHI` $<value_1, value_2>$, which means the value in the new variable is either $value_1$ or $value_2$. The Tree-SSA form is the interme-

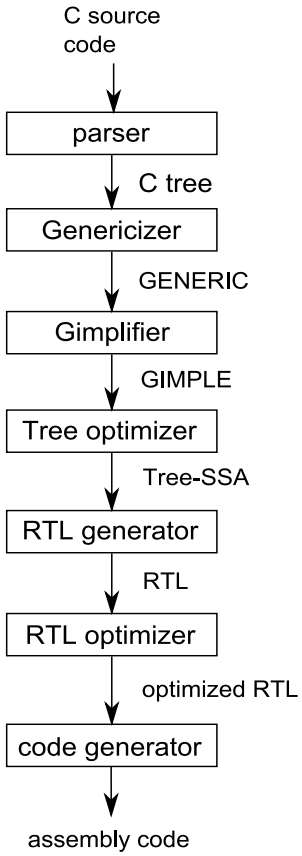

Figure 2.4: Compiler flow of the GCC compiler

diate representation on which the most of GCC's optimization passes operate. This part of the compiler is called the middle-end.

RTL
When all passes in the middle-end are finished the *RTL generation* pass transforms the Tree-SSA representation into machine specific *Register Transfer Language* (RTL) representation. This representation is a LISP-like syntax in which every expression should represent a single instruction on the target machine.

assembly code
The final pass generates assembly code suitable for the target architecture from the RTL representation.

2.3.3 Passes of the compiler

All transformations from the source code to the assembly output are done by *passes* of the GCC compiler. Each pass implements a necessary (e.g. register allocation or instruction scheduling) or optional (e.g. dead store elimination or loop optimization) transformation of a certain representation of the code. In this section we will give a brief description of the important passes in the GCC compiler.

Passes are added and removed from one version to the next. The first real 'pass' of the compiler is running the *parser*, which produces the intermediate GENERIC format, which in turn is reduced to GIMPLE representation through a process commonly called *gimplification*. The C language front end directly emits GIMPLE intermediate representation. Most passes in the compiler operate on the GIMPLE or RTL representation. Since GCC 4.3.1 invokes over 200 compiler passes, we will only outline the ones relevant to the project here.

2.3.3.1 Passes that operate on the GIMPLE intermediate representation

Remove Useless Statements
This pass removes obvious dead code statements. Simple things like if-statements with a constant condition are evaluated and replaced by the corresponding block, and unreachable code is removed. This is done early so all subsequent passes have an easier time applying their optimizations.

Lower Control Flow
In this pass several constructs are simplified in order to avoid unnecessarily complex control flow graphs. If-statements are converted into a form where they hold only two `GOTO` statements; one for the then- and one for the else-block. This results in reduced complexity of the control flow graph allowing subsequent passes to recognize oportunities for optimization more effectively.

Build Control Flow Graph
Here the *control flow graph* is built. All functions are split into a number of basic blocks and all control flow edges are defined.

Enter Static Single Assignment form
The last pass that operates on the GIMPLE representation transforms all statements into Static Single Assignment form.

2.3.3.2 Passes that operate on the Static Single Assignment representation

The following passes all operate on the Tree-SSA form:

Forward Propagation of Single-Use Variables
This pass substitutes variables that are only used once into the expression which uses them and tries to simplify the resulting expression. This results in lower register pressure, which can avoid the need for register spilling during register allocation.

Copy Renaming
This pass tries to rename compiler-generated temporary variables so they can be coalesced later. This also brings user-defined variables closer to the place where they are used. This assists in debugging the application.

PHI Node Optimizations
This pass tries to rewrite PHI node statements into conditional code, or even entirely remove it if possible. This reduces the number of jumps in the code.

May-Alias Optimization
This pass does flow-sensitive 'points-to' alias analysis of the code, marking which areas of the memory may be addressed by multiple pointers. This information is important because a value pointed to by one pointer may be changed by another. This introduces a number of dependencies that must be respected.

Dead Store Elimination
The dead store elimination pass removes any stores whose value is not used before it is overwritten by another store.

Tail Recursion Elimination
When a function includes a recursive call at the end, this can be rewritten into loop form. This pass detects such occurrences and replaces them by a loop. The overhead for a loop is much smaller than for a function call, resulting in an improvement in runtime performance.

Forward Store Motion
This pass moves stores and assignments as close as possible to the point where they are used. This reduces the *live range* of the variable and improves register allocation performance.

Redundancy Elimination
Redundancy Elimination removes expressions that are evaluated multiple times, and substitutes each later occurrence with the previously computed value, thereby reducing the number of instructions in the instructions stream.

Loop Optimization
This pass (or actually several passes) performs a number of optimizations that improve the execution time of code within loops, like *loop invariant motion* (moving code not dependent on the current loop iteration out of the loop body), some *induction variable optimizations*, *loop unrolling* and *vectorization*. All these optimizations reduce either the number of instructions in the loop body or the number of iterations of a loop.

Tree-level If-Conversion for the Vectorizer
This pass applies if-conversion to statements inside the loop body to enable the vectorizer to parallelize more loops.

Control Dependence Dead Code Elimination
This pass performs dead code elimination on control flow statements. For example jumps to a label that directly follows the jump instruction are removed.

Small Loop Unrolling
This pass completely unrolls loops that iterate a small number of times.

Reassociation
This pass rewrites some expressions to a way that can more easily be optimized at a later stage through for example vectorization.

2.3.3.3 Passes that operate on the RTL representation

The resulting optimized Tree-SSA code is now converted to the machine specific RTL representation, on which another number of optimizations are performed.

RTL generation
The RTL generation pass converts all SSA statements into RTL expressions that correspond to machine instructions on the target architecture. This is done through expansion rules (see section 3.2.1).

Common Subexpression Elimination
This pass performs the elimination of (sub)expressions that are identical by calculating the result once and substituting it in the expressions that use it.

Loop Optimization
Another set of loop optimizations is performed, similar to the pass which operates on the SSA representation.

If-Conversion
This pass performs *if-conversion*, substituting simple if-statements by conditional instructions. This removes the jump instructions resulting from these if-statements.

Web Construction
To improve the performance of other passes, web construction splits the independent uses of each pseudo-register. Common subexpression elimination and register allocation can benefit from this transformation.

Life Analysis
This pass calculates which pseudo-registers are live at every point in the program. This information is needed for register allocation and some other optimization passes. If possible, it eliminates instructions which set pseudo-registers that are never used.

Instruction Combination
The *instruction combination* pass merges two or three instruction that are connected via data flow into a simpler or less expensive instruction. This pass effectively performs peephole optimization on instructions connected in the control flow graph but that are not necessarily consecutive in the instruction stream.

Register Movement
This pass checks all pseudo-registers against the constraints in the instruction pattern and in all cases where reloading the register would result in the generation of a register-to-register move, tries to change the register in order to avoid the generation of this move.

Instruction Scheduling
This pass performs instruction scheduling of the code. Instruction scheduling is the ordering of instructions in order to obtain correct performance when the instructions are executed on the target machine. Instruction scheduling must not violate any data dependencies between the different instructions. For interlocked machines, like the EVP, the scheduler must also avoid any resource conflicts. Non-interlocked machines handle these resource conflicts by inserted NOPs during execution, and this is less of an issue. For these machines the main goal of the scheduler is to ensure correctness. For this project, this also involves scheduling the instructions into VLIW instruction words. To obtain the highest performance in terms of cycle count this involves scheduling as many instructions as possible per clock cycle without violating any constraints or dependencies.

Register Allocation

The *register allocation* pass consists of several stages. It first tries to find the preferred register for each pseudo-register, it then proceeds to allocate hard registers to all pseudo-registers, starting on a local level (i.e. within a basic block) followed by global register allocation. The *reload* pass finishes register allocation by assigning each pseudo-register a hard register, or in case no free hard registers are available, a stack slot.

Target Defined Optimizations

This pass allows the back end to call a number of target-specific optimization passes, which perform any optimizations or make changes necessary for generating correct assembly instructions. This is the pass where the passes we add in this thesis are added. These are `evp_remove_jumps_pass`, which replaces a jump to another jump instruction by a jump to the target of the second jump, `evp_div_to_end_bb_pass`, which avoids resource conflicts in inter-basic block scheduling (detailed in section 3.3.7, the `evp_delay_slot_filler_pass` which fills the delay slots of branch and jump instructions (see section 3.5.1), the `evp_hloop_pass` which performs the generation of hardware loops (see section 3.5.3) and the `evp_inter_bb_scheduler` which performs inter-basic block scheduling (see section 3.3.6).

Final

This is the last pass which actually generates the assembly instructions based on the assembly output template of the patterns of all instructions in the instruction stream. Also any other assembly code that needs to be emitted (comments, values in memory, etc.) are written here.

2.4 Summary

In this chapter we presented the basic layout of the EVP, we described the functional units, their purpose and the way they are interconnected. We discussed the registers of the EVP and the way vector data is stored in memory. We also discussed techniques featured by the EVP architecture that require support from the compiler, hardware loops and predicated execution, as well as the fact that the EVP is a VLIW architecture. Furthermore we discussed the basics of the EVP-C extension to the C language that allows programmers to program C code that closely resembles assembly code, without having to take into account instruction scheduling and register allocation. Finally we presented the basic structure of the GCC compiler, the different intermediate representations and we briefly outlined the more important optimization passes.

3 Implementation

In this chapter we will describe all the additions and changes made to the EVP back end to allow it to correctly and efficiently generate assembly code for the EVP. In section 3.1 we will first describe how we define the EVP machine properties like registers, register classes and machine modes. Following this, in section 3.2, we will describe how the transformation from Tree-SSA representation to RTL instruction is implemented. The largest part of this project, enabling the VLIW scheduler, is detailed in section 3.3. We will describe how the EVP assembly is emitted in section 3.4. In section 3.5 we will detail how some target-specific features like hardware loops, branch delay scheduling and post-increment addressing are implemented.

3.1 Specifying the machine properties

This section details how all properties of the EVP, like registers, register classes and machine modes are defined.

3.1.1 General machine properties

All the specific properties of the target must be defined by means of target macros in the EVP back end. These macros are defined in the `evp.h` file in the target directory. First of all GCC needs to know specifics like the width and availability of certain data types and even the number of bits in the smallest available unit, a byte in this case.

To set the size of different data types in our back end values are defined through use of the following macros:

```
#define BITS_BIG_ENDIAN         1
#define BYTES_BIG_ENDIAN        0
#define WORDS_BIG_ENDIAN        0
#define UNITS_PER_WORD          2
#define UNITS_PER_SIMD_WORD     32
```

GCC defines a `UNIT` as the smallest addressable memory unit, for the EVP this is a byte, so the default value for `BITS_PER_UNIT` can be used, which is 8. From this the width of different words is defined. The macro `UNITS_PER_SIMD_WORD` defines the number of units the autovectorizer can put in one vector, or SIMD word. The endianness is defined so GCC knows which byte should be loaded (or stored) when only the high or low part of a multi-byte word is required.

The ANSI C standard does not put strict requirements on the width of all C data types, so these need to be defined. This is done through use of the following macros:

```
#define SHORT_TYPE_SIZE         BITS_PER_WORD
#define INT_TYPE_SIZE           BITS_PER_WORD
#define LONG_TYPE_SIZE          (2 * BITS_PER_WORD)
#define LONG_LONG_TYPE_SIZE     (4 * BITS_PER_WORD)
#define FLOAT_TYPE_SIZE         32
#define DOUBLE_TYPE_SIZE        32
#define LONG_DOUBLE_TYPE_SIZE   64
#define TARGET_FLOAT_FORMAT     IEEE_FLOAT_FORMAT
```

The EVP has a 24-bit wide address bus, but 32-bit wide pointer registers. The arithmetic on pointer registers is 24-bit however, as explained in section 3.1.2. The `POINTER_SIZE` macro defines the width of pointers in GCC, and although the pointers contain 24-bit values, they are stored in 32-bit words in memory. Therefore the width of a pointer is defined as follows:

```
#define POINTER_SIZE            32
```

To distinguish them from other 32-bit data types, GCC uses a different data type for pointers internally, these data types are called *machine modes*. For more details see section 3.1.4.

3.1.2 Registers

Internally, GCC numbers all hard registers, starting with 0. In the back end (`evp.h` file in the target directory) we therefore need to define which registers correspond which numbers. For this the macro `REGISTER_NAMES` must be defined to hold a C initializer which contains all register names as they are referred to in the assembly output. This is an array initializer containing all register names in the following order:

- 31 general purpose registers
- 8 predicate registers
- 16 pointer registers
- the return address register
- 3 loop control registers
- 8 offset registers
- 16 general purpose vector registers
- 8 vector mask registers
- 2 intra vector registers
- 2 intra vector mask registers
- 16 vector shuffle pattern registers
- 4 CGU registers

Up to the register allocation phase all RTL expressions assume the availability of an infinite number of registers. GCC uses all register numbers above the last hard register as *virtual registers*, or *pseudo-registers*. The register allocator maps all pseudo-registers on hard registers in the *reload* pass.

Now that all registers are defined, we need to mark the registers that are fixed, i.e. those registers that the register allocator can not make use of during register allocation. These fixed registers include the stack pointer, frame pointer, argument pointer and any registers that have a fixed value and cannot be written to, such as `p0` and `vm0`.

The macro `FIXED_REGISTERS` is, similar to `REGISTER_NAMES`, an initializer that sets a '1' for all hard registers that are fixed, and a '0' otherwise.

In the EVP port the registers whose values are hard-coded (`p0`, `vm0`), the special purpose register `ra`, all loop control registers, the pointer registers `ptr13`, `ptr14`, `ptr15` as well as `r30` and `r31` are marked as fixed.

The three pointer registers are used for the argument pointer, frame pointer and stack pointer respectively. In many cases the argument pointer and frame pointer can be expressed in terms of the stack pointer. Only when an application supports dynamic stack space allocation (using *alloca*) then the correlation between these different pointers cannot be determined at compile time, and these registers are needed. At the moment, a subset of the standard C library is implemented, and *alloca* is not supported. This means that these two registers can be made available for register allocation. Because the test programs currently used do not exhibit high register pressure for the pointer registers, marking `ptr13` and `ptr14` as fixed does not incur a performance penalty.

The register pair `r31r30` is reserved for moves between `ofs` and `ptr` registers that are generated during the reload phase. This will be explained further in section 3.2.1.1.

The predicate register `p7` is also reserved. It is needs to remain free to allow the if-conversion pass to use it. For more details on if-conversion see section 3.5.5.

In addition to defining the fixed registers, we need to define for each register whether it is available across a call or not. These call-used registers are defined in the macro `CALL_USED_REGISTERS`, and must include at least all fixed registers.

This macro contains all registers that can be freely used within a function. Registers that have a '0' in `CALL_USED_REGISTERS` must be saved upon function entry (in the function prologue) and restored upon returning (in the epilogue) if they are used within a function. Similarly, if a register contains a live value and a function call is made, code is generated to save these values on the stack. This save and restore process is detailed further in section 3.2.4.

For the EVP port for GCC, we stick to the same calling convention the current production compiler uses, as detailed in [35]. This way it will be possible to link to libraries compiled with the production compiler and ensure correctness. This calling conventions specifies that the following registers are call-used:

- `r0 - r15`
- `p1 - p3`
- `ptr0 - ptr3`
- `ptr8 - ptr11`

- ofs0 - ofs4
- vr0 - vr7
- vm1 - vm3
- ir0 - ir1
- im0 - im1
- vsp0 - vsp7

In addition to these registers, we also mark the vector registers vr8 through vr15 as call used. This is done because the GCC register allocator prefers registers defined as call-used above registers not defined as call-used. Programs executed on the EVP mainly consist of code that spends the majority of the time in loops and only a small portion in the function prologue and epilogue. Therefore the performance hit that allocating a call-used register incurs (this takes place outside the main loops) is marginal compared to the loss in performance that results from sub-optimal register allocation (which affects the inner loops where register pressure can be high). This loss is caused by *false dependencies* that are introduced by re-using a certain register too close to the end of its previous live range, thereby generating an unnecessary *write-after-read dependency* that causes extra latency.

Marking these extra registers still maintains correctness when GCC-compiled programs are linked with the libraries compiled with the production compiler. At most it generates superfluous register saves. If a value in one of the registers vr8 - vr15 is live before a library call, it will unnecessarily saved before the call.

3.1.3 Register classes

In order to properly distinguish the different types of registers different register classes are defined, by means of the reg_class enumeration type. Table 3.1 depicts which registers belong to which register classes.

In conjunction with reg_class an array of C string constants must be defined that holds the name for each register class. These strings are identical to the names used in the reg_class enumeration type.

In order to connect the different register classes with previously defined hard registers, for each register class a bitmap containing a '1' in each position that corresponds to a register that belongs to that class is defined. As an example, the entry for general purpose vector registers in the reg_class array is the following:

```
{0x00000000, 0x00000000, 0x000ffff0, 0x00000000} , /* VR */ \
```

For this example, this means the registers numbered 68 through 83 belong to the VR_REGS register class.

In order to enhance register allocation, we define the macros BASE_REG_CLASS and INDEX_REG_CLASS to PTR_REGS and OFS_REGS respectively. The register allocator will actively try to load register that are used for the base part in base-plus-offset addressing in a ptr register, which will avoid any unnecessary moves. The same is applicable for the offset part, we want that as much as possible in an ofs register.

Register Class	Contents
NO_REGS	Empty register class
R_REGS	General purpose registers
P_REGS	Predicate registers
PTR_REGS	Pointer registers
RA_REGS	Return address register
LSA_REGS	Loop Start Address registers
LEA_REGS	Loop end address register class
LC_REGS	Loop counter register class
OFS_REGS	Offset register class
VR_REGS	General Purpose Vector registers
VM_REGS	Vector Mask registers
IR_REGS	Intra Vector registers
IM_REGS	Intra Vector Mask registers
VSP_REGS	Vector Shuffle Patterns registers
CGU_REGS	CGU registers
ALL_REGS	All registers

Table 3.1: Contents of the register classes

3.1.4 Machine modes

GCC uses the notion of *machine modes* to refer to a data object by it's size and the representation used [24]. A number of machine modes are predefined, and there is room for target-specific modes. These modes are defined by the file `insn-modes.h` in the GCC build directory, which is generated by functions in `genmodes.c`, in the GCC source directory. `insn-modes.h` defines the standard GCC modes from the file `machmode.def` in the GCC source directory, as well as target specific modes. These target specific modes are defined in the EVP back end, in the file `evp-modes.def` in the target directory.

By default, GCC generates bit, integer, floating point, fixed-point, complex integer, complex floating point and decimal floating point modes. In total 52 modes are predefined[24]. The EVP supports operations on 1-, 8-, 16- and 32-bit scalars. These modes which correspond to these scalar sizes are depicted in table 3.2.

The modes are named for their size relative to a four byte integer. The 32-bit integer mode is named `SImode`, for *Single Integer mode*, the 16-bit mode is called `HImode`, or *Half Integer mode* and the 8-bit mode is named `QImode`, for *Quarter Integer mode*. The 1-bit mode, needed for the predicate registers, is called `BImode` for *Bit mode*.

In addition, we need to need to introduce several extra modes for the EVP target. First of all the pointer registers, which as mentioned in section 3.1.2, are 32-bits wide but address a 24-bit memory space. For this a *Partial Single Integer* mode is defined, for which GCC has support built in. This `PSImode` is defined by adding the line

```
PARTIAL_INT_MODE (SI);
```

to `evp-modes.def`. Secondly, to implement the vector part of the port we need to define the different vector modes supported by the EVP. This is done by adding the lines

Machine Mode	Width in bits
BImode	1
QImode	8
HImode	16
SImode	32
PSImode	32
V32QImode	32 × 8
V16HImode	16 × 16
V8SImode	8 × 32
V16SImode	16 × 32

Table 3.2: Machine Modes used in EVP-GCC.

```
VECTOR_MODE (INT, SI,  8);
VECTOR_MODE (INT, HI, 16);
VECTOR_MODE (INT, QI, 32);
VECTOR_MODE (INT, SI, 16);
```

to `evp-modes.def`, GCC translates this into a definition for `V8SImode`, `V16HImode`, `V32QImode` and `V16SImode`. The first argument specifies the type of vector, all integers in our case. The second argument specifies the size of each element in the vector, whereas the last argument specifies the number of those elements. This results in three 256 bit wide vectors, consisting of either 32 `QImode` elements, 16 `HImode` elements or 8 `SImode` elements. The final vector is a 512-bit vector, consisting of 16 32-bit elements[1].

All modes relevant in the EVP port are listed in table 3.2.

In order to tie the machine modes to the register classes, the macro `HARD_REGNO_MODE_OK (REGNO, MODE)` returns *true* if a value of machine mode `MODE` can be stored in register `REGNO`. This macro is tied to the function `evp_hard_regno_mode_ok_func`, which returns *1* or *0* according to table 3.3. An entry *E* in table 3.3 signifies a mode where only even numbered hard registers are valid. `evp_hard_regno_mode_ok_func` checks for this and returns a *1* in case of even numbered registers, and a *0* otherwise.

For some register classes[2], multiple registers are required to hold a value of a certain machine mode. The macro `HARD_REGNO_NREGS (REGNO, MODE)` is set to return the number of registers needed to store a value of mode `MODE` in register `REGNO`. Closely related to this is the macro `CLASS_MAX_NREGS (class, mode)` which should return the maximum number of registers of class `class` to hold a value of mode `mode`. This information assists the register allocator with handling multi-word moves during the reload phase.

Values in a certain machine mode sometimes need to be accessed in a different mode. Most of the time this incurs extra overhead when an intermediate operation is required, such as a sign-extend or zero-extend. Some values can be accessed in a different mode without any intervention. These modes are called *tieable*, and we let GCC know by defining the macro `MODES_TIEABLE_P (mode1, mode2)`. This macro returns *true* when all values residing in

[1] This port of the EVP does not include full support for vectors larger than 256-bit. Although the mode V16SImode is defined, we will not use in this project

[2] Those marked with an *E* in table 3.3.

		Register Class									
		R	P	PTR	OFS	VR	VM	IR	IM	VSP	CGU
Mode	BImode	0	1	0	0	0	0	0	0	0	0
	QImode	1	0	0	0	0	0	0	0	0	0
	HImode	1	0	0	0	0	0	0	0	0	0
	SImode	E	0	0	0	0	0	0	0	0	0
	PSImode	E	0	1	1	0	0	0	0	0	0
	V32QImode	0	0	0	0	0	1	0	0	0	0
	V16HImode	0	0	0	0	1	0	1	1	0	0
	V8SImode	0	0	0	0	0	0	0	0	1	0
	V16SImode	0	0	0	0	E	0	0	0	0	0

Table 3.3: Return value of the evp_hard_regno_mode_ok_func.

valid registers for `mode1` can also be accessed in those same registers in `mode2`. In effect this is equal to all combination of modes in Table 3.3 whose rows are identical. In this case only (besides the trivial case where `mode1` is equal to `mode2`) if `mode1` and `mode2` are `QImode` and `HImode`. Knowing the interchangeability of these two modes will improve register allocator performance.

When all vector data types and registers are implemented, there are two ways vectors can be used in C code:

- Allow GCC to generate vector operations from a series of scalar operations that operate on scalar data types. This is called *vectorization*. Support for vectorization is included in GCC and we performed a number of experiments to see how much performance improvement vectorization can bring on our benchmark code and what steps need to be taken to allow GCC to effectively vectorize this code. The results of these experiments show that vectorization will not bring much improvement without support for masked vector execution (predicated execution with vector registers) and this is not yet implemented in EVP-GCC. These experiments are presented in more detail in section 4.3.
- Generate vector operations from EVP-C intrinsic operations. This is the method currently used in product code for the EVP. The vector benchmarks used in this project also make use of EVP-C intrinsic to generate vector operations.

3.2 Expanding Tree-SSA into RTL

This section describes how the transformation from Tree-SSA level representation to RTL representation is made. Section 3.2.1 describes defining all expansion rules for the EVP back end. Section 3.2.2 details the implementation of function calls. Argument passing is discussed in section 3.2.3 and function prologue and epilogue expansion is detailed in section 3.2.4.

3.2.1 Defining expansion rules

To make the transition from the list of SSA statements generated by the GCC middle-end to the list of RTL instructions used in the back end of the compiler, GCC requires *expansion rules* to be set. GCC

uses a series of standard names for these expansion rules, which are automatically used (if available) during the RTL generation pass.

Expansion rule are defined by an RTL expression of the following form:

```
(define_expand
  <name>
  <RTL template>
  <condition>
  <preparation statements>
)
```

The `name` field contains a name for the expansion rule. Each rule should have a name, otherwise the expansion rule will never be called. This name comes in the form of a string, and should match one of the standard names. It is used by the RTL generation pass to expand the corresponding SSA statement into RTL. For example,

```
my_int1_3 = my_int1;
```

which is a move from one integer variable to another (both `HImode` in our configuration) will use the expansion rule `movhi`, which generates the correct RTL expression for a move from one `HImode` pseudo-register to another. It is also possible to define an expansion rule with a non-standard name, however it will not be called from the GCC mainline, but can be used by the back end where needed. For example we use a call to the generated function `gen_movhi_salu_or_slsu` to generate a move from one scalar register to another which can be issued on either the `SALU` or `SLSU`. How this is done exactly is explained in more detail in section 3.3.4.

The RTL template field contains the template for the RTL expression that should be generated. Continuing with the example above, for the `movhi` rule this is

```
(set (match_operand:HI 0 "general_register_operand" "")
     (match_operand:HI 1 "general_register_operand" ""))
```

These two `match_operand` expressions match the zeroth and first operands of the SSA statement to a register that should be valid for both the mode (`HI`) and the predicate (see section 3.2.1.5).

The third field is a condition for when this expansion rule should be used. This is a boolean expression which will be evaluated when the rule is used. An empty expression here always evaluates to *true*.

The last field is a block of C code that is evaluated before the expansion is done. This code can make changes to the operands that are used, or emit extra RTL instructions in the instruction stream. Two special macros can be used in this block of code, `FAIL`, which treats the rule as if the condition field returned *false* and `DONE`, which ends the generation of the pattern without emitting the RTL in the template field. Otherwise, the RTL template will be emitted after the preparation statements are executed.

Most expansion rules are relatively straightforward. There are a few exceptions, which we will discuss here.

3.2.1.1 Expansion of unsupported move instructions

The EVP does not support all types of move instructions. One such instruction is a move between two memory locations. If the expansion rule for a move is called and both operands are memory locations, regular expansion will result in an unrecognized RTL instruction. Therefore the preparation statements include checks for cases where both operands are memory locations, and generates a new pseudo-register and emits two moves into the instruction stream. One move from the source memory location to the new pseudo-register and one from the pseudo-register to the target memory location. This code is added to all move expansion rules. For example. the following lines have been added to the preparation statements for the `movhi` expansion rule:

```
if ((reload_in_progress | reload_completed) == 0
     && !register_operand (operands[0], HImode)
     && !register_operand (operands[1], HImode))
{
  rtx temp = force_reg (HImode, operands[1]);
  temp = emit_move_insn (operands[0], temp);
  DONE;
}
```

First both operands are checked, which are contained in the `rtx`-array `operands[]`. If they both are not in either a register or pseudo-register we use the built-in GCC function `force_reg` to force one of the memory locations into a register, and emit two move instructions into the instruction stream. By doing so we generate a new pseudo-register, which will hold the value that needs to be copied and still resides in memory at the location `operands[1]` points to. We must take care not to do so after the reload phase, because after register allocation we cannot generate any new pseudo-registers. Therefore the check `(reload_in_progress | reload_completed)` must evaluate to zero. These two global variables are set by the *reload* pass to signify that reloading the pseudo-registers has started and no new pseudo-registers should be generated.

3.2.1.2 Expanding vector instructions

To implement the vector instructions, we first define the expansion rules. The move instructions require rules for the standard pattern names `movv32qi`, `movv16hi` and `movv8si`. These rules are automatically selected whenever an SSA statement which contains a vector move is encountered. For example the expansion rule for a vector move with machine mode `V16HImode` is:

```
(define_expand "movv16hi"
 [(set (match_operand:V16HI 0 "nonimmediate_operand" "")
       (match_operand:V16HI 1 "nonimmediate_operand" ""))]
 ""
 { evp_vector_expand_move( V16HImode, operands );
   DONE;
 }
)
```

The RTL template in this rule is not relevant, all vectors moves are expanded via the function `evp_vector_move`. This function takes care of the expansion, and catches the case where both operands do not reside in a register or pseudo-register, similar to the preparation statements for scalar moves, described in section 3.2.1.1.

3.2.1.3 Expanding Vector Intrinsics

To support all the EVP-C vector intrinsics, expansion rules for all 1406 intrinsics have to be added to the back end. In the testbench used, only 99 are employed, so these are the ones implemented. For all these intrinsics, expansion rules are defined in the files `vbuiltins.md` and `builtins.md`. All intrinsics are expanded into an RTL expression that makes use of the `unspec` operation, which denotes an operation whose functionality is unspecified. As an example, we give the expansion rule for the EVP-C intrinsic `evp_vadd_16`, which returns the addition of two vectors assuming 16 bit elements. Inside the back end files, all intrinsics are referred to as *builtins*, and the `evp_` part of the intrinsic names is replaced with `__builtin_` in the `evp.h` header file which should be included in all files containing EVP-C intrinsics that are to be compiled with EVP-GCC.

```
(define_expand "__builtin_vadd_16"
  [(set (match_operand:<vector_mode> 0 "vr_register_operand" "=Yvr")
    (unspec:V16HI
      [ (match_operand:V16HI 1 "vr_register_operand" "Yvr")
        (match_operand:V16HI 2 "vr_register_operand" "Yvr")
      ] UNSPEC___builtin_vadd_16) )]
  ""
  ""
)
```

The `unspec` construct takes a vector with a number of arguments (two in this case) and a number to distinguish this RTL expressions from other `unspec` expressions. The term `UNSPEC___builtin_vadd_16` is a constant defined earlier. These constants, one for each intrinsic, are defined using a `define_constants` statement in `vbuiltins.md`.

```
(define_constants
 [
    (UNSPEC___builtin_vstore 1000)
    (UNSPEC___builtin_p_vstore 1001)
    (UNSPEC___builtin_vadd_8 1002)
    (UNSPEC___builtin_vadd_16 1003)
     ...
    (UNSPEC___builtin_vmov_ptr_on_vlsu 1083)
 ]
)
```

These constants start counting from 1000 onward for vector intrinsics, and from 2000 onward for scalar intrinsics.

The expansion rule above causes the SSA statement (taken from the testbench file `fft_btail_vvi.c`)

```
v_0109 = __builtin_vadd_16 (v_0, v_4);
```

to the following RTL expression:

```
(insn 104 103 105 4 fft_btail_vvi.c:241 (set (reg:V16HI 261)
        (unspec:V16HI [
                (reg/v:V16HI 175 [ v_0109 ])
                (reg/v:V16HI 169 [ v_2115 ])
            ] 1003)) -1 (nil))
```

Besides defining the expansion rules and instruction patterns, we have to initialize the builtin functions to GCC. This is done in two target hooks, `TARGET_INIT_BUILTINS` and `TARGET_EXPAND_BUILTIN`. To simplify adding intrinsics to the target, we generate this initialization and expansion through the use of two definition files, `builtins.def` and `vbuiltins.def`, for the scalar and vector builtins, respectively. These files contain a simple definition for all builtins according to their EVP-C syntax, and store information like the intrinsic name, builtin code, name of the corresponding expansion rule, the arity and type of arguments needed. Continuing with the example we give the definition of the previously mentioned `evp_vadd_16` intrinsic in `vbuiltins.def`:

```
DEF_BUILTIN( "__builtin_vadd_16",
             BUILTIN___builtin_vadd_16,
             __builtin_vadd_16,
             2,
             V16HI_type_node, V16HI_type_node, V16HI_type_node )
```

This definition contains all the information needed to implement an intrinsic operation. Depending on the required information, the macro `DEF_BUILTIN` can be defined locally in a way that extracts only the fields required.

GCC handles all builtins as function calls, and has to recognize them as such. For this the target hook `TARGET_INIT_BUILTINS` must be defined, which initializes all builtins. This hook is tied to `evp_init_builtins`, and contains the following:

```
void evp_init_builtins(void)
{

#define DEF_BUILTIN(name, code, rtl_name, arity, ...)                          \
  add_builtin_function(name, build_function_type_list (__VA_ARGS__, NULL_TREE), \
                       code, BUILT_IN_MD, NULL, NULL_TREE) ;

#include "builtins.def"

#undef DEF_BUILTIN

  evp_vector_init_builtins();
}
```

When the `DEF_BUILTIN` macro is expanded, this will result in a call to the GCC front-end function `add_builtin_function` that will add the builtin to the internal GCC hash table so it can be recognized, based on it's name (the first field of the `DEF_BUILTIN` construct). It also calls `build_function_type_list` which returns a `tree` of the types of all arguments of the builtin.

When all scalar builtins are initialized and can now be recognized by the compiler, `evp_init_builtins()` calls a separate function that initializes the vector builtins.

```
void evp_vector_init_builtins(void)
{
    V32QI_type_node = build_vector_type(intQI_type_node, 32);
    V16HI_type_node = build_vector_type(intHI_type_node, 16);
    V16SI_type_node = build_vector_type(intSI_type_node, 16);
    V8SI_type_node = build_vector_type(intSI_type_node, 8);

    unsigned_V32QI_type_node = build_vector_type(unsigned_intQI_type_node, 32);
    unsigned_V16HI_type_node = build_vector_type(unsigned_intHI_type_node, 16);
    unsigned_V16SI_type_node = build_vector_type(unsigned_intSI_type_node, 16);
    unsigned_V8SI_type_node = build_vector_type(unsigned_intSI_type_node, 8);
```

```
    p_V32QI_type_node = build_pointer_type(V32QI_type_node);
    p_V16HI_type_node = build_pointer_type(V16HI_type_node);
    p_V16SI_type_node = build_pointer_type(V16SI_type_node);
    p_V8SI_type_node = build_pointer_type(V8SI_type_node);

#define DEF_BUILTIN(name, code, rtl_name, arity, ...) \
  add_builtin_function(name, build_function_type_list (__VA_ARGS__, NULL_TREE), \
       code, BUILT_IN_MD, NULL, NULL_TREE) ;

#include "vbuiltins.def"

#undef DEF_BUILTIN
}
```

This function does the same thing for the vector builtins as `evp_init_builtins` does for scalar builtins, but a few extra types are needed, since the target-specific types are not known to GCC. For this a few new types are defined. For each vector mode a type node is generated by calling the GCC function `build_vector_type`, supplying the type of each element and the number of elements as arguments.

When the compiler encounters a builtin during the expansion pass and it is recognized as such, it expands the macro `TARGET_EXPAND_BUILTIN`, which calls the function `evp_expand_builtin()`.

```
rtx evp_expand_builtin(tree exp, rtx target, rtx subtarget ATTRIBUTE_UNUSED,
                       enum machine_mode mode ATTRIBUTE_UNUSED, int ignore ATTRIBUTE_UNUSED)
{
  tree fndecl  = TREE_OPERAND (CALL_EXPR_FN (exp), 0);
  tree arglist = CALL_EXPR_ARGS (exp);
  unsigned int fcode = DECL_FUNCTION_CODE (fndecl);

  if (evp_vector_builtin_p(fcode))
    return evp_vector_expand_builtin(exp, target, subtarget, mode, ignore);

#define DEF_BUILTIN(name, code, rtl_name, arity, ...) \
  if (fcode == code) \
  { \
    return evp_expand_nary_builtin (arity, CODE_FOR_##rtl_name,        \
                                    arglist, target); \
  } else

#include "builtins.def"
  return 0;

#undef DEF_BUILTIN

  return 0;
}
```

This function first extracts the arguments and the `code` of the builtin, and then checks if we are dealing with a vector or a scalar builtin. In case of a vector builtin, the function `evp_expand_vector_builtin` is called (which is a copy of this function, only it includes `vbuiltins.def`). Then, through use of including the right builtin definition file and testing the function code of the builtin against the builtin code defined in the `.def` file, the function `evp_expand_nary_builtin` is called and supplied the arity, builtin code, arguments and target operand.

`evp_expand_nary_builtin` does the real expansion to RTL for all builtins.

```
rtx evp_expand_nary_builtin(int arity, int icode, tree arglist, rtx target)
{
  rtx pat, op[arity];
  tree aux, args[arity];
  enum machine_mode target_mode, modes[arity];
  int i;

  aux = arglist;
  for (i = 0; i < arity; i++, aux = TREE_CHAIN(aux))
    args[i] = TREE_VALUE(aux);

  target_mode = insn_data[icode].operand[0].mode;
  for (i = 0; i < arity; i++)
    modes[i] = insn_data[icode].operand[i + 1].mode;

  for (i = 0; i < arity; i++)
    op[i] = expand_expr(args[i], NULL_RTX,  modes[i] , 0);

  for (i = 0; i < arity; i++)
    if (VECTOR_MODE_P(modes[i]))
      op[i] = evp_safe_vector_operand(op[i], modes[i]);

  if (target == NULL_RTX)
    target = gen_reg_rtx(target_mode);

  for (i = 0; i < arity; i++)
  {
    if (GET_CODE (op[i]) == SUBREG && GET_MODE (op[i]) == QImode
        && ((icode == CODE_FOR___builtin_cmpsi_eq || icode == CODE_FOR___builtin_cmpsi_lte)
            || (icode == CODE_FOR___builtin_cmphi_eq || icode == CODE_FOR___builtin_cmphi_lte) ))
    {
      rtx bi_target = gen_reg_rtx(BImode);

      op[i] = bi_target;
      target = op[i];
    }
    else if (GET_CODE(op[i]) == SUBREG && GET_MODE(op[i]) == QImode && modes[i] == BImode)
    {
      rtx bi_target = gen_reg_rtx(BImode);

      emit_move_insn(bi_target,
                     gen_rtx_SUBREG(BImode, SUBREG_REG(op[i]), SUBREG_BYTE(op[i])));
      op[i] = bi_target;
    }
    else if (GET_MODE(op[i]) == modes[i] || GET_MODE(op[i]) == VOIDmode)
      op[i] = force_reg(modes[i], op[i]);
  }

  if (arity == 1)
    pat = GEN_FCN(icode) (target, op[0]);
  else if (arity == 2)
    pat = GEN_FCN(icode) (target, op[0], op[1]);
  else if (arity == 3)
    pat = GEN_FCN(icode) (target, op[0], op[1], op[2]);
  else if (arity == 4)
    pat = GEN_FCN(icode) (target, op[0], op[1], op[2], op[3]);
  else if (arity == 5)
    pat = GEN_FCN(icode) (target, op[0], op[1], op[2], op[3], op[4]);
  else if (arity == 6)
    pat = GEN_FCN(icode) (target, op[0], op[1], op[2], op[3], op[4], op[5]);
  else if (arity == 7)
    pat = GEN_FCN(icode) (target, op[0], op[1], op[2], op[3], op[4], op[5], op[6]);
  else
    error("no expand builtin for this arity!!");

  if (!pat)
```

```
    return 0;
  emit_insn (pat);

  return target;
}
```

This function extracts the arguments of the builtin and stores them in the arity-sized array `args[]`, extracts the modes af all the operands and stores them in the array `modes[]`, and builds an array with the operands of the builtin called `op[]`. Following this the RTL of the target (if it is non-empty) is generated and has the proper mode.

Then all arguments to the builtin are forced into registers via the GCC `force_reg` function. In some cases, forcing the register is not enough. GCC treats booleans standard as typecasted ints, and the operand supplied to a builtin that needs a `BImode` operand is a `SUBREG` of an `HImode` register, and simply forcing this into a `BImode` will not work. Therefore we catch these cases and manually generate a move from the current register into a `BImode` register.

After all arguments are of a mode expected by the `define_expand` rules, the generated expansion function corresponding to this builtin is called by means of the GCC macro `GEN_FCN (icode)`, which results in the generation of an RTL expression that matches the `define_insn` of the builtin we are currently expanding. If this RTL expression is generated, it is then emitted into the instruction stream.

3.2.1.4 Defining the instruction patterns

When all SSA statements are expanded, each RTL instruction in the instruction stream should correspond to a machine instruction. These machine instructions are defined separately using the `define_insn` construct in the machine description files. These instruction patterns are similar to the expansion rules, with the following fields:

```
(define_expand
  <name>
  <RTL template>
  <condition>
  <output template>
  <instruction attributes>
)
```

The first three fields are the same as for an expansion rule, with the remark that the name field is optional. GCC assigns all patterns a number, and defines for named patterns a constant `CODE_FOR_<pattern name>`. This constant can be used to recognize instructions in back end functions. Alternatively, patterns can be named with a string preceded by a '`*`'. This is effectively the same as an unnamed pattern, but is printed in debugging dump and assists with debugging the compiler.

The *output template* field contains a string (or a block of C code returning a string) that is used when the assembly code is generated at the end of compilation. For more information on this, see section 3.4.

The (optional) *instruction attributes* field contains attributes for the operation. Attributes are defined in the machine description files (see section 3.2.1.7) and consist of enumeration types containing

different possible values for the attribute. One of the uses of these attributes is to mark the usage of functional units for the scheduler (see section 3.3.2).

These instructions patterns are defined for all possible RTL expressions that can be generated by the *RTL generation* pass.

To continue the example of the `evp_vadd_16` instruction described in section 3.2.1.3, the instruction pattern is defined as follows:

```
(define_insn "*__builtin_vadd_16"
 [(set (match_operand:V16HI 0 "vr_register_operand" "=Yvr")
   (unspec:V16HI
    [ (match_operand:V16HI 1 "vr_register_operand" "Yvr")
      (match_operand:V16HI 2 "vr_register_operand" "Yvr")
    ] UNSPEC___builtin_vadd_16)
  )
 ]
 ""
 "vadd16 %0, %1, %2%# evp_vadd_16"
 [(set_attr "vector_type" "valu_l")]
)
```

Each `match_operand` rule in the RTL template specifies the valid operands to this instruction. The last two fields contain a *predicate* and the *constraints*. The predicate is a rule that returns *true* or *false* based on the mode of the operand and its register class. The predicates used in EVP-GCC are detailed in section 3.2.1.5. In this case `vr_register_operand` checks if the operand is in a `vr` register (or a pseudo-register if we are still before the *reload* pass). The constraint `Yvr` also denotes a `vr` register. The '=' symbol marks this operand as an output operand, i.e. it is being written to in this instruction. This information is used in optimization passes like *global common subexpression elimination* and *dead code elimination*. The constraints defined for the EVP are detailed in section 3.2.1.6.

A predicate can cover more than one register class. Generally the constraints should cover all possible combinations of these register classes. One such case is the load of a `PSImode` value from memory into a register. A value of this mode can reside in two general purpose registers, one pointer or one offset register.

```
(define_insn "psi_simple_load"
  [(set (match_operand:PSI 0 "psi_register_operand" "=b,b,Z,Z,d,d")
        (mem:PSI (match_operand:PSI 1 "simple_addr_operand" "i,b,i,b,i,b")))]
  ""
  "load %0, %1 %)%#"
  [(set_attr "type" "load")
   (set_attr "mode" "SI")
   (set_attr "length" "3")
   (set_attr "long_imm" "yes,no,yes,no,yes,no")])
```

Here the predicate `psi_register_operand` is valid for `PSImode` for the three different register classes mentioned above, and `simple_addr_operand` accepts a pointer register or immediate address. To reflect this in the constraints, it is a comma separated list that has all possible combinations of these registers and immediates. The first pair of constraints to match the operands determines the output of the assembly statement. The attributes can also change depending on which set of constraints is matched. In this case, of the second operand is a pointer register, the `long_imm` attribute is set to `no`, if it is an immediate address, it is set to `yes`.

As a general rule, all permutations of the predicates should be represented in the constraints field. If not all permutations are valid, the instruction pattern must be split up into multiple patterns. One such case is the move from one intra-vector mask register to another. The corresponding register class is `V16HImode`, the same as for a `vr` register. An `im` register does not contain the same number of bits as a `vr` register. However, the GCC register allocator is not aware of this, and tries to spill `vr` registers into `im` registers instead of into memory. The move back from the `im` register to the `vr` register results in a loss of data. Therefore the instruction pattern for a move between `V16HImode` operands is split into two instruction patterns:

```
(define_insn "move_v16hi"
  (set (match_operand:V16HI 0 "vr_or_ir_register_operand" "=Yvr,Yvr,Yir,Yir")
       (match_operand:V16HI 1 "vr_or_ir_register_operand" "Yvr,Yir,Yvr,Yir"))
  ""
  "vmove %0, %1%)%#"
  [(set_attr "vector_type" "valu_1")
   (set_attr "length" "1")
   (set_attr "long_imm" "no")]
)

(define_insn "move_vm_to_vm_v32qi"
 [(set (match_operand:V32QI 0 "vm_register_operand" "=Yvm")
       (match_operand:V32QI 1 "vm_register_operand" "Yvm"))
 ]
 ""
 "vmand %0, %1, vm0%)%#"
 [(set_attr "vector_type" "vmalu_1")
  (set_attr "length" "1")
  (set_attr "long_imm" "no")]
)
```

This way the reload pass will recognize there is no available move from `vr` to `im` registers and back, and will spill onto the stack.

3.2.1.5 Defining the predicates

For all RTL expressions not every operand is valid. We cannot load a scalar from an address referenced by a vector for example. Therefore all operands in an RTL expression need to be checked and allowed or disallowed as necessary. For this GCC uses *predicates* and *constraints*. Predicates are checked when GCC expands a statement into an RTL expression or when an RTL expression is matched to a certain `define_insn`. GCC has a number of built in standard predicates, and for more specific tests target-specific predicates can be defined. The standard GCC predicates are[24]:

immediate_operand

const_int_operand

const_double_operand

register_operand

pmode_register_operand

scratch_operand

memory_operand

address_operand

indirect_operand

push_operand

pop_operand

nonmemory_operand

general_operand

These built in predicates are all implemented like functions in GCC, with the operand and the mode of the operand passed as arguments. As said earlier also target-specific predicates need to be defined.

To define target-specific predicates, we use the `define_predicate` construct, which consists of the name of the predicate, a RTL expression and an optional block of C code, all of which should return *true* if an operand matches the predicate. As an example, in order to construct the predicate `general_register_operand` we add the following lines to the `predicates.md` file:

```
(define_predicate "general_register_operand"
  (match_code "reg,subreg")
  {
    if (! register_operand(op, mode))
      return 0;
    if (GET_CODE (op) == SUBREG)
      op = SUBREG_REG (op);
    return GENERAL_REGNO(REGNO(op))||PSEUDO_REGNO(REGNO(op));
  })
```

which checks if the operand code is either `REG` or `SUBREG`, and proceeds to evaluate the C code, which will return *true* if and only if the operand satisfies the `register_operand` predicate and if the register number of the operand is either an `r` register or a pseudo-register.

In a similar way the following predicates are defined:

general_register_operand
: Matches any general purpose (`r`) register or pseudo-register that is of `QImode`, `HImode`, `SImode` or `PSImode`. This predicate also matches subregisters which meet these requirements.

strict_general_register_operand
: Identical to `general_register_operand` but doesn't match pseudo-registers.

predicate_register_operand
: Matches any predicate register or pseudo-register that is of `BImode`.

base_register_operand
: Matches any register that is either a `ptr` register or a pseudo-register and is of `PSImode`.

strict_base_register_operand
: Identical to `base_register_operand` but doesn't accept pseudo-registers.

index_register_operand
Matches any `ofs` register or pseudo-register of `PSImode`.

strict_index_register_operand
Identical to `index_register_operand` but doesn't accept pseudo-registers.

loop_counter_register_operand
Only matches either the `lc` register or a pseudo-register.

r_register_operand
Checks whether the operand is an `r` register or pseudo-register, but does not accept subregisters.

r_reg_or_imm_operand
Matches any operands that match either the `r_register_operand` or the standard `const_int` predicate.

simple_addr_operand
This predicate matches any operand that matches either the `base_register_operand`, `const_int` or `symbol_ref` predicate. This is any address operand that refers to a valid memory location to load/store a value to/from.

simple_addr_operand1
Identical to `simple_addr_operand`, only accepts no `ptr` registers. This means immediate addresses only.

address_operand1
Matches any memory address of either a label (inside program memory) or a constant memory location (inside data memory).

immediate_addr_plus_immediate
Matches all addresses that come in the form of an immediate plus an offset that is also an immediate. This predicate is only used in the `nonaddress_operand` predicate, to conveniently exclude these construction.

nonaddress_operand
Matches any immediate or register operand that is not a symbol reference, label reference or matches the `immediate_addr_plus_immediate` predicate.

reg_or_const_int_operand
Matches any operand that matches either the `register_operand` or the `const_int` predicate.

register_or_simple_addr_operand
This predicate matches everything that matches either the `register_operand` or `base_register_operand` predicates. In other words, any `r`, `ptr` or pseudo-register that is of `QImode`, `HImode`, `SImode` or `PSImode`.

general_register_or_index_operand

general_or_base_register_operand

immediate_or_general_register_operand

immediate_index_general_register_operand

base_immediate_operand

base_or_index_operand

base_index_or_immediate_operand
These predicates are all combinations of `general_register_operand`, `base_register_operand`, `index_register_operand` and `const_int`. These are inclusive or so a match for either sub-predicate returns a match for this predicate.

psi_register_operand
This predicate matches all possible register operands that can hold a `PSImode` value. This is a combination of `base_register_operand`, `index_register_operand` and `general_register_operand`.

offset_operand
Can be either a `ptr`, `ofs`, immediate or `PSImode` pseudo-register operand, with the exception of constant references to a label or symbol. Basically any operand that can function as an offset in pointer-offset addressing mode.

offset_operand1
Similar to `offset_operand`, but cannot match `ptr` registers.

offset_ptr_update
Any operand that a pointer can be updated with matches this predicate. Either in a regular `ptr_update` instruction or with post-increment loads or stores. This means anything that either matches `index_register_operand` or has a `const_int` code, with the exception of `SYMBOL_REF`s or `LABEL_REF`s, as these can only be assigned to a pointer through a move.

general_or_ra_register_operand

base_or_ra_register_operand

base_index_or_ra_register_operand
These three predicates are combination of the `ra` register with the `general_register_operand`. `base_register_operand` and `base_or_index_operand` predicates respectively.

evp_call_operand
This predicate is valid for any operand that can be a target for a call instruction.

Not all of these predicates are implemented with `(match_operand_...)` or similar expressions. For some predicates a supporting function inside `evp.c` was defined, named `evp_ok_for_...` This was done for predicate, pointer, offset, loop counter and general purpose registers. As an example, we will give the `evp_reg_ok_for_base_p` function here.

```
int evp_reg_ok_for_base_p(rtx op, enum machine_mode mode, int strict)
{
    if (GET_CODE(op) != REG)
        return false;
    if (mode != Pmode)
```

```
        return false;
    if (strict)
        return PTR_REGNO (REGNO (op));
    else
        return (PTR_REGNO (REGNO (op)) || PSEUDO_REGNO (REGNO (op)));
}
```

Where the `op` argument is the operand to be checked, the `mode` argument is the machine mode the operand should have, and `strict` is a boolean that signifies whether pseudo-registers are also matched against this predicate. `strict` is set to *true* for all predicates ending in `_strict`, and *false* otherwise.

The vector register predicates in `vpredicates.md` are rather straightforward.

vr_register_operand

vm_register_operand

ir_register_operand

im_register_operand

vsp_register_operand

cgu_register_operand
: These predicates make a simple check if the operand is a certain type of register. For each vector register a supporting function `evp_ok_for_regno_..._p` is made and checked every time this predicate is checked.

vr_or_ir_register_operand
: This predicate is used in those instructions that can accept either a `vr` or an `ir register`.

allequal_vector_const_operand

allequal_vector_const_int_operand

allequal_vector_const_int_0_operand

allequal_vector_const_int_2_operand
: These four predicates check whether a vector operand is made up of all identical constants. `allequal_vector_const_int_0_operand` and `allequal_vector_const_int_2_operand` check if this value is equal to zero or two respectively. These two predicates are used for the vector clear instruction, and the `vsub_div2` and `vadd_div2` instructions, which add or subtract two vectors and divide the result by 2.

3.2.1.6 Defining Constraints

When the predicates inside a `define_insn` are matched, there still can be differences in the instruction attributes or the assembly mnemonic that has to be emitted for a set of specific operands. Let's take for example the `movpsi_general2` instruction:

```
(define_insn "movpsi_general2"
  [(set (match_operand:PSI 0 "general_or_base_register_operand"   "=b,d")
        (match_operand:PSI 1 "address_operand1"  ","))]
  ""
  "@
   move_slsu %0, %1 %)%#
   move %0, %1 %)%#"
  [(set_attr "type" "slsu_s1,alu")
   (set_attr "mode" "SI")
   (set_attr "length" "1")
   (set_attr "long_imm" "yes,yes")]
)
```

Here we see a `set` RTL expression that moves an operand that matches the `address_operand1` predicate to one that matches the `general_or_base_register_operand`. Since the target operand can be either a pointer or a general purpose register and the assembly mnemonic is different for either one, GCC needs a way to make this distinction. For this the constraints are specified. For the target operand in this case this is `"=b,d"`. The = signifies that this operand is an *output operand*, and is changed in this instruction. The `b` matches any `ptr` register, and the `d` any `r` register. The `","` string in the constraints field of the source operand matches any register or constant. When for example the compiler has generated a `PSImode` move of the address of the global variable `foo` to the register pair `r1r0`, the second set of constraints is met and in the assembly generation phase, GCC uses the second alternative in the assembly output template and correctly outputs the instruction

```
move r1r0, @foo
```

Not only can the constraints result in different assembly output generation, but also we can specify attributes based on which alternative is selected. As seen above we have specified the `type` attribute for the first alternative to be `slsu_s1`, and `alu` for the second. If only one value is given, it applies to all alternatives. The `type` attribute helps to determine on which functional unit an operation is to be scheduled. For more on this, see section 3.3.2.

GCC has several predefined constraints, of which we only use the `i` constraint, which signifies any constant immediate value, also symbolic constants that are not known until compile time.

To further specify the constraints GCC provides the `define_constraint` and `define_register_constraint` constructs. These target-specific constraints we have defined in the file `constraints.md`, included from the main machine description file, `evp.md`.

The `define_register_constraint` allows us to directly tie a register class to a constraint letter or letters, which we have done for most register classes:

```
(define_register_constraint "d" "R_REGS" "")
(define_register_constraint "j" "P_REGS" "")
(define_register_constraint "b" "PTR_REGS" "")
(define_register_constraint "y" "RA_REGS" "")
(define_register_constraint "Z" "OFS_REGS" "")
(define_register_constraint "T" "LC_REGS" "")
```

```
(define_register_constraint "Yvr" "VR_REGS" "")
(define_register_constraint "Yvm" "VM_REGS" "")
(define_register_constraint "Yvs" "VSP_REGS" "")
(define_register_constraint "Yir" "IR_REGS" "")
(define_register_constraint "Yim" "IM_REGS" "")
(define_register_constraint "Ycg" "CGU_REGS" "")
```

These are all letters not in use by GCC, and are specifically kept free to allow the back end to create their own target-specific constraints.

Because the EVP has a complex instruction coding scheme, the size of an immediate argument in the instruction stream can influence the usage of functional units in the EVP. Only one instruction each cycle can make use of the long immediate field of the instruction word, and depending on the instruction the maximum size or value of an immediate determines whether or not using this long immediate field is necessary. To avoid using this field as much as possible, we have to be able to specify exactly which immediate GCC can and should use. To this end we define thirteen new constraints all pertaining to these immediates.

C0
: A constant with the value 0.

C1
: A constant with the value 1.

I
: A 32-bit unsigned integer (i.e. between 0 and 0xFFFFFFF).

J
: A 24-bit signed integer (between 0x800000 and 0x7FFFFF).

K
: A 24-bit unsigned integer (between 0 and 0xFFFFFF).

L
: A 16-bit unsigned integer (between 0 and 0xFFFF).

M
: A 16-bit signed integer (between 0x8000 and 0x7FFF).

N
: A constant integer in the range [-32, 31].

O
: A constant integer in the range [-16, 15].

P
: A power two (with a positive or negative sign) in the range [-32, 32].

U3
: A constant integer in the range [0, 7].

U4
A constant integer in the range [0, 15].

U5
A constant integer in the range [0, 31].

Most constraints can be handled by a boolean expression, but for the `P` constraint a new function is introduced, `evp_LSIMM_immediate (int)`. This function returns whether or not a constant satisfies the `P` constraint.

When GCC checks whether a certain alternative is valid, it returns the first alternative that fits. This means that if a certain set of constraints is a subset of another, the most optimal one from a scheduling point of view should occur first in the constraints field of the RTL.

3.2.1.7 Defining the instruction attributes

As mentioned in section 3.2.1.4, each instruction definition can have several function attributes. These attributes can be queried through use of a function generated by GCC. For example the function used to query the `type` attribute of an instruction is `get_attr_type(rtx insn)`. To check the returned value, an enumeration type is defined containing all possible values of the attribute. Querying the type attribute of an instruction that has `ptr_ofs_move` as its type will result in the value corresponding to `TYPE_PTR_OFS_MOVE`.

To define these attributes we use the `define_attr` construct, which takes as arguments the name of the attribute, the possible values and a default value which is applied if the attribute is not specified in the instruction pattern. For example, the `long_imm` attribute is specified as follows:

```
(define_attr "long_imm" "yes,no" (const_string "no"))
```

3.2.2 Function calls

To expand function calls GCC has the predefined pattern names `call` and `call_value`, which are expanded when a function call is met in Tree-SSA. The `call` pattern expands a function call that doesn't return a value, whereas `call_value` is called for functions that return a value.

GCC passes three arguments to the `define_expand`, the first is the function to call, the second the number of bytes passed as a `const_int`, and the third the number of registers that are used as operands. In case of a non-void function a fourth argument is passed to signify the hard register that should contain the return.

The function `evp_expand_call` handles function call expansion. Several things must be taken care of for a function call. In case of a function pointer, the function address first has to be moved to two `r` register containing the address in `SImode`, because the EVP only accepts either an immediate address or a pair of `r` registers as operands to a `call` assembly instruction. In case of a function that is passed a variable number of arguments on the stack (the `printf` function from the standard C library for example) we need to make the register signified in `next_arg_reg` point to the last argument that was pushed onto the stack, and then emit the call instruction in parallel with a `clobber` and a `use` expression, to tell the compiler that the value in `next_arg_reg` may not be valid after

returning from the call and that the value in `next_arg_reg` is used inside the callee function, so it should be optimized away.

When a non-stdargs function is called, we simply emit an RTL expression of the form

```
(set (match_operand:<ret_val_mode> 0 "register_operand" "=d,d")
     (call (mem (match_operand:SI 1 "immediate_or_general_register_operand" "r,i")
           (const_int 0)))
```

for a function returning a value and one of the form

```
(call (mem (match_operand:PSI 0 "immediate_or_general_register_operand" "r,i"))
      (const_int 0))
```

for calls to functions not returning anything. To support the parallel RTL expression generated for stdarg functions, also instruction patterns are written that recognize the instruction with the `use` and `clobber` expressions.

3.2.3 Function argument passing

In order to pass arguments to callee functions there is a specific convention to be followed. For this we use the same convention the production compiler currently uses[35]. Arguments can only be passed in call-used registers (see section 3.1.2). For each register type, the lower-numbered registers are used first, with the exception of pointer registers. For these the order is: `ptr0`, `ptr8`, `ptr1`, `ptr9`, `ptr2`, `ptr10`, `ptr3` and `ptr11`.

To get all arguments in their correct registers, GCC calls the target hook `FUNCTION_ARG` for each argument that the function being called requires. To assist with this, a target-specific struct `CUMULATIVE_ARGS` must be defined and initialized through `INIT_CUMULATIVE_ARGS`. This struct contains all necessary information about the function call, such as the function declaration tree, whether or not it is a library call, the number of arguments left to pass and the number of arguments already passed in each register type.

The steps in `evp_function_arg` are:

```
evp_function_arg
{
  if (argument cannot be passed in a register)
    return NULL_RTX
  else
    if (there is a no free register available of required type)
      return NULL_RTX
    else
      increase number of registers used
      return the register used
}
```

GCC then emits a move of the required value to the register returned or, if no register was returned, pushes the argument onto the stack.

3.2.4 Function prologue and epilogue generation

The function prologue and epilogue should contain all context save and restore instructions. This includes saving the old frame pointer on the stack. These instructions are emitted in the functions `evp_expand_prologue` and `evp_expand_epilogue`. These functions are called from the prologue and epilogue expansion rules.

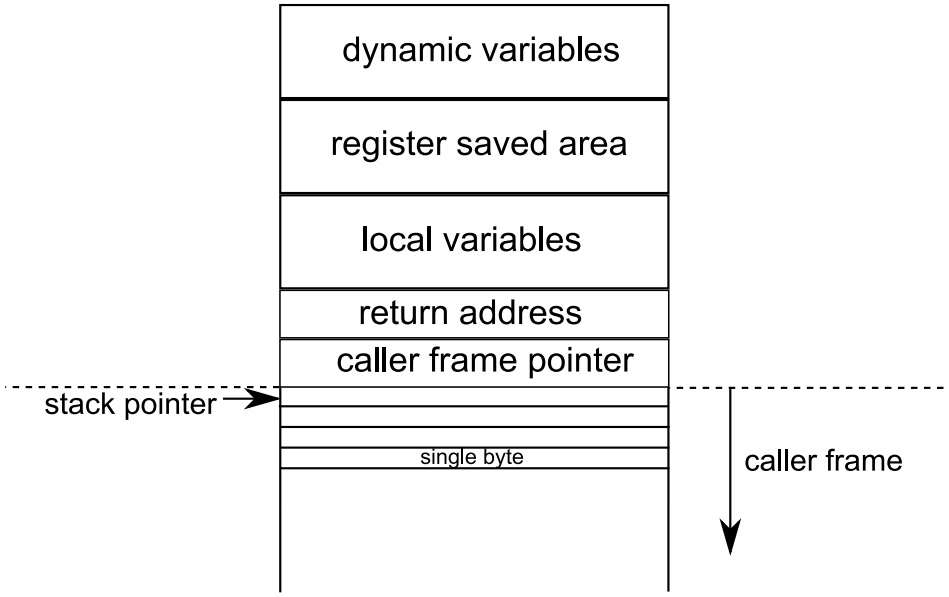

Figure 3.1: EVP-GCC Stack frame layout

The layout of the stack frame, as per the calling convention of the production compiler, is depicted in figure 3.1. First the prologue and epilogue expansion function calls the supporting function `evp_stack_info`, which calculates for all hard registers whether they should be saved on the stack or not, and at which offset from the stack pointer.

This information is stored in a struct, which contains an array which in turn holds a struct for each register that stores several pieces of information. After some initialization, the `evp_stack_info` function checks to see if the return address register and old frame pointer need to be stored. For the `ra` register, this is a simple check against the variable `current_function_is_leaf`, and the frame pointer needs to be stored if the size of the frame is non-zero.

Following this, space is reserved on the stack for local variables that need to be stored during the function execution. A simple call to `get_frame_size()` determines the space needed. After this, the start of the area where the call saved registers are stored is aligned to a 32-byte boundary.

The function now traverses all registers to check if they should be saved on the stack and at which offset. Registers should be saved if they are not marked as *call-used* (see section 3.1.2) and if they are live within the function. This check if a register is live or not is done via a call to the GCC mainline function `df_regs_ever_live_p`, which returns whether a register is ever live inside the current function or not. If both conditions hold, the variable `offset`, which holds the current offset relative to the stack pointer, is increased by the size of the register, and stored in the `reg_offset` array at

the location of the corresponding register. Similarly, a field in the `save_p` array is set denoting that the register should be saved, and the field in the `reg_mode` array stores the mode of the register.

To minimize the number of context save/restore instructions, we can merge instructions that load or store two consecutive `r` registers into one. To facilitate this, we first check all even `r` registers to see if they are live and not call-used together with the next odd register. If this is the case, the even register is marked to be saved or restored as an `SImode` value, and the following odd numbered register is marked as not saved. A separate variable keeps track of all registers marked in this way and they will be skipped when traversed individually.

Another problem that arises with the layout of the stack frame is the fact that regular vector loads and stores require the data to be aligned to a 256-bit boundary[3]. In order to meet this requirement, the offset value is aligned to a multiple of 32 before all vector register offsets are calculated. This guarantees alignment of all vector registers to the required boundaries.

3.3 VLIW Scheduling

Another of the major tasks in the EVP-GCC port is the scheduler of the compiler. The EVP is a VLIW processor, which means that several instructions can be scheduled in a single clock cycle and are executed in parallel[25]. This section describes the steps taken to implement VLIW scheduling in GCC.

3.3.1 The Deterministic Finite Automaton

The scheduling pass of GCC makes use of a *Deterministic Finite Automaton* (DFA) as proposed by Vladimir Markov in [28]. This DFA scheduler determines if a new instruction can be scheduled based on the current state of the pipeline, which contains information on the state (occupied or free) of all functional units in the pipeline in all stages. The scheduler maintains a linked list of instructions whose data dependencies have been resolved (the *ready list*) and tries to schedule the instruction with the highest priority. The priority of all instructions is determined at the start of new cycle and is based on whether or not an instruction is part of the critical path to the end of the basic block. Instruction priority can also be modified through use of the `TARGET_SCHED_ADJUST_PRIORITY` target hook (for more details on changes in instruction priorities, see section 3.3.5). If a new instruction can be scheduled, the finite state machine advances to a new state which reflects the state of the pipeline after the instruction is scheduled in the current cycle. If an instruction cannot be scheduled due to resource conflicts, it is delayed, and the scheduler tries to schedule another instruction from the ready list. If the ready list is empty or all functional units are occupied, the scheduler advances a cycle, recalculates priorities and continues the scheduling. This cycle repeats until all instructions in a basic block are scheduled.

To model the processor, we have to define all functional units, the resource reservation patterns and match all instructions to a specific pattern.

[3]The EVP also supports unaligned memory accesses, but they have a higher latency and use additional functional units. Therefore they should be avoided if possible.

3.3.2 Defining the resource usage patterns

Besides data dependencies, a limiting factor for the number of instructions that can be scheduled in parallel is the availability of free functional units to schedule the instructions on. In order to model the usage of these functional units, each instruction has a certain *time shape*, which defines the occupation of functional units in different pipeline stages. These time shapes are divided into classes. Each instruction belongs to a certain class.

The EVP pipeline has a depth of 9 and consists of the following stages:

```
FET1 FET2 FET3 DEC1 DEC2 DEC3 RRF EXE1 EXE2 EXE3 EXE4
```

Note that an instruction can stay in some stages for more than one cycle. Because the resource usage in the first five stages (instruction fetch and decode) is identical for all instruction classes, we will model the resource reservations starting with the `DEC3` stage.

The functional units of a processor are defined through the `define_cpu_unit` construct. For example, to model the `PALU` functional unit, we define

```
(define_cpu_unit "r_palu" "evp")
```

This construct takes a name and the automaton it is part of.

Resource reservation patterns are modeled using two different constructs. The patterns itself are defined through a `define_reservation` construct, and instructions are matched to a reservation pattern via the `define_insn_reservation` construct. To define the pattern for the reservation class which uses the `PALU` we write

```
(define_reservation "c_palu" "nothing * 2, r_palu, nothing * 3")
```

This signifies that the `c_palu` reservation class uses the `r_palu` functional unit in the third cycle (the `EXE1` stage). In all other cycles no units are occupied.

In order to match instructions to certain reservation patterns, we use the `type` attribute defined in the attributes vector of all instructions. For vector instructions this is the `vector_type` attribute. The matching rule is then set up as follows:

```
(define_insn_reservation "palu" 1 (eq_attr "type" "palu") "c_palu")
```

The first field defines the name of this match, which is only used in debugging dumps and the second field specifies the latency of the instruction. The third field is an expression that matches the `type` attribute. In this case, if the `type` attribute is equal to `palu`, it matches this expression.

When a value is written to a register (for an instruction with latency 1) it can typically be used by a following instruction two cycles later (the value is ready at the end of the writeback stage, while the value is read at the start of the read stage). The EVP contains an extensive *bypass* network, which allows the read stage to fetch the value from the bypass network instead of from the register file where it will be available a cycle later. When defining the latency of instructions through the second field of the `define_insn_reservation` construct, this bypass network is already taken into account. However, not all register files are connected in such a way. For this we have to tell the compiler

that data dependencies between some instruction classes are different from the standard value defined above. This is managed through the `define_bypass` construct. For example, an instruction of the `slsu_1` class cannot access a value calculated by a `salu_move` class instruction through the bypass network. Therefore the latency has to be increased to 2 through the following bypass definition:

```
(define_bypass 2 "salu_move" "slsu_s1")
```

The first field denotes the new latency followed by the source and destination instruction classes.

3.3.3 Modeling the long immediate field

This method of describing the pipeline can do more than just model functional units. The way instructions are encoded in the EVP allows only one instruction per VLIW word to use the long immediate field of the opcode. This puts an extra constraint on scheduling. To model this long immediate constraint, an extra 'unit' is added to the pipeline description. This unit, called `long_imm_unit` is used in several extra reservation patterns which are added next to the already existing ones. To model a single cycle `SLSU` instruction with one operand being a long immediate, the following reservation pattern is introduced next to the `c_slsu_1` pattern:

```
(define_reservation "c_slsu_1" "nothing, r_slsu, nothing * 4")
(define_reservation "c_slsu_1_long_imm" "long_imm_unit, r_slsu, nothing * 4")
```

This new reservation pattern now occupies the long immediate field in the `DEC3` stage. Other instructions that also require this field now cannot be scheduled in the same cycle, thereby guaranteeing correctness.

3.3.4 Scheduling semantically equivalent operations

This section refers to scheduling semantically equivalent operations with different mnemonics on different functional units. Another complication during targeting the GCC scheduler was the fact that the EVP allows scheduling of semantically equivalent operations on several different functional units. For example moving data between two general purpose registers can be issued on two different functional units `c_salu_1` or `c_slsu_s1`. For this move instruction the instruction class `c_hi_move` is introduced. This class is defined as the OR of the `c_salu` and `c_slsu_s1` instruction classes:

```
(define_reservation "c_hi_move" "c_salu_1 | c_slsu_s1")
```

The scheduling automaton generated when these new instruction classes are introduced has a deterministic behavior, meaning that it will always try to match the first instruction class (`c_salu_1` in this case) and only if it cannot schedule the instruction on that unit, it will try to schedule it using the second pattern[4]. Scheduling semantically equivalent operations on different functional units improves the performance of the generated code (cycle count-wise), but involves some complication. The complication is due to the fact that GCC emits an assembly mnemonic based on which alternative of an

[4]The scheduler can also generate a non-deterministic automaton, in which the choice for a specific pattern is delayed until the last moment. This could potentially improve the performance, but requires a redesign of the evp_automaton_query function.

RTL instruction an expression belongs to. This alternative only selects based on the RTL constraints which do not provide resource allocation information. In order to find a way to encode this information, an extra RTL expression is emitted in parallel with all instructions that can be issued on different units. This expression is of the form `(clobber (match_operand N "const_int"))`. The value of this operand can now be used to include the information in the RTL instruction. With this information the generated assembly can be dependent on the functional unit on which it is issued, for example for the `movhi_salu_or_slsu` instruction:

```
(define_insn "movhi_salu_or_slsu"
 [(set (match_operand:HI 0 "r_register_operand" "=d")
       (match_operand:HI 1 "r_register_operand" " d"))
  (clobber (match_operand 2 "const_int_operand" "=i"))]
 ""
 "*
  {
    switch (INTVAL(operands[2]))
    {
      case 33:
        return \"move_slsu %0, %1 %)%#\";
      case 22:
        return \"move %0, %1 %)%#\";
      case 2:
return \"FAIL: value is still 2\";
      default:
        break;
    }
    fatal_insn(\"something went wrong...\", insn);
  }"
 [(set_attr "type" "hi_move")
  (set_attr "mode" "HI")
  (set_attr "length" "1")
  (set_attr "long_imm" "no")]
)
```

The information contained in operand 2 (the clobber expression) is used to emit the correct assembly instruction. This value is set in the function `evp_automaton_query`. The implementation of which is detailed later in this section.

In this way all instructions that have multiple options for scheduling are processed, but this puts a heavy burden on the scheduling automaton. When all instructions are implemented the generation time of the scheduling automaton at build time grew very large[5]. In order to cut down the build time of the compiler the automaton was split into two separate automata: all functional units belonging to the `SDCU` and `ACU` are coupled in the automaton `scalar`, and all functional units of the `VDCU` in the `vector` automaton. This split resulted in two much smaller automata, and except for the long immediate unit, all instruction reservation patterns consist of units out of a single automaton. The only problem here was the `movpsi_slsu_vlsu` instruction, which could issue a pointer update instruction on either the `VLSU` or the `SLSU`. We cannot define an instruction reservation in which both alternatives are from different automata. The rules for splitting automata state that if a unit from a given automaton in present in one alternative cycle, a unit from the same automaton should be present in all alternatives for a given cycle[28].

We therefore required another approach to schedule these semantically equivalent operations. For this we implemented the `evp_automaton_query` function, which is tied to the target hook

[5] At the moment the number of states in the two automata is roughly 37000 combined, and the number of arcs roughly 250,000. For a single automaton these numbers are estimated to be at least a 1000 times larger.

TARGET_SCHED_DFA_NEW_CYCLE, which is called every time a new instruction is about to be scheduled. This function is passed the instruction to be scheduled as an RTL expression. In case this instruction is an instruction that can be scheduled on multiple different functional units, we create a number of new instructions, one for each alternative functional unit. We assign each newly created instruction one of the alternative resource usage patterns and consequently attempt to schedule them on the current state of the instruction scheduling automata. This is done via a call to the function internal_state_transition, which makes the transition in the automata states as if the current instruction would be scheduled. If internal_state_transition signifies that the instruction can be scheduled, we then proceed to trick the compiler into thinking the original instruction belongs to the instruction class (and has the according resource reservation pattern) of the alternative that could be scheduled. The compiler maintains an array that contains for each instruction the instruction class to which it belongs, and we simply change that through use of the following code:

```
int uid = INSN_UID(ready);
dfa_insn_codes[uid] = internal_dfa_insn_code(insn_copy);
```

This first retrieves the unique id number of the instruction and consequently changes the entry in the dfa_insn_codes[] array to match the instruction code from insn_copy, which is the copy of the original instruction that belongs to the alternative that can be scheduled. The values stored in dfa_insn_code[] are codes that represent the internal DFA codes of all instruction classes. The code corresponding to the instruction class we require is obtained via a call to internal_dfa_insn_codes. When these values are set the last thing evp_automaton_query does is set the value in the clobber field of the original instruction to match the right key, which allows the assembly generation pass to emit the correct mnemonic.

This implementation is not constrained by the fact that all resource used by a the alternatives of a given instruction should contain elements from the same automaton. This allows for much more freedom in splitting the scheduling automata, and avoids large automata which increase the build time and compile time of the compiler.

Adding these clobber expressions to the instruction patterns required significant rewriting of all instructions that can be scheduled on multiple functional units. These include the pointer or offset moves described above and moves of HImode values, but also vector moves and moves of QImode and SImode values. To generate the new instruction patterns the preparation statements of the expansion rules (or functions called in these preparation statements, see section 3.2.1) were modified to emit the new form of RTL expressions. As an example, to the end of the code in the movhi expansion rule the following lines were added:

```
if (register_operand(operand0, HImode)
    && register_operand(operand1, HImode)
    && GET_CODE(operands[0]) == REG
    && GET_CODE(operands[1]) == REG)
{
  emit_insn (gen_movhi_salu_or_slsu (operand0, operand1, GEN_INT(2)));
  DONE;
}
```

This results in the generation of an instruction for which the RTL expression matches the movhi_salu_or_slsu instruction pattern as described previously. This is one of the reasons to name an instruction pattern. If such a pattern is named, GCC generates the function gen_<name>

which takes all required operands as arguments and returns an instruction matching the corresponding pattern.

An additional problem that arose is that when instructions accompanied by these clobbers are register allocated, it can occur that the instruction is transformed into a move between a register and a memory location (e.g. when a spill is needed). In this case the instruction no longer matches any instruction pattern, because all load and store instructions are `SET` expressions without a clobber in parallel. To support the cases where this would occur we added additional instruction patterns to the machine description. To stay with the case of a move of an `HImode` value, the new instruction pattern for a simple load is:

```
(define_insn "hi_simple_load"
  [(set (match_operand:HI 0 "general_register_operand" "=d,d")
        (mem:HI (match_operand:PSI 1 "simple_addr_operand" "i,b")))]
  ""
  "load %0, %1 %)%#"
  [(set_attr "type" "load")
   (set_attr "mode" "HI")
   (set_attr "length" "3")
   (set_attr "long_imm" "yes,no")])

(define_insn "hi_simple_load_clobber"
  [(set (match_operand:HI 0 "general_register_operand" "=d,d")
        (mem:HI (match_operand:PSI 1 "simple_addr_operand" "i,b")))
   (clobber (match_operand:SI 2 "const_int_operand" "=i,i"))]
  ""
  "load %0, %1 %)%#"
  [(set_attr "type" "load")
   (set_attr "mode" "HI")
   (set_attr "length" "3")
   (set_attr "long_imm" "yes,no")])
```

The first pattern is the one of the regular load instruction, the second one is the pattern that has to be added. A total of 49 new patterns were added to the machine description.

3.3.5 Specifying scheduling priorities

Every time a new scheduling cycle is started, GCC orders the instructions in the ready list and tries to schedule instructions with a higher priority before other instructions. These priorities are based on how close an instruction is to the critical path in the basic block. The target hook `TARGET_SCHED_ADJUST_PRIORITY` is called when these priorities are determined. This allows the back end to give preference to certain instructions. To improve the branch delay scheduling we want the instruction loading the target of a branch instruction as early as possible (see section 3.5.1), so there are a maximum number of instructions available to be moved into a branch delay slot later on, without compromising the minimum latency between the load instruction and the branch instruction. The following lines are therefore added to the `TARGET_SCHED_ADJUST_PRIORITY` target hook:

```
if (evp_use_registers_p(insn, p_registers_p)
    || evp_use_registers_p(insn, ra_register_p) )
    return (priority << 3);
```

The check to see if an instruction contains a predicate register is related to the fact that we want the maximum distance also between a conditional jump and the instruction where the corresponding predicate register is set.

3.3.6 Inter-basic block scheduling

The GCC scheduler schedules all basic blocks individually, but the EVP has some instructions that have a long latency, like the division, multiplication and loads. This can cause dependencies between two instructions that are not within the same basic block. To make sure these dependencies are respected we introduce a new pass in the machine dependent `reorg` phase.

This pass, called `evp_inter_bb_scheduler` traverses all basic blocks, identifies these dependencies and inserts nops where needed. This function performs the following steps:

```
evp_inter_bb_scheduler
{
  for (all basic blocks)
  {
    for (all hard registers)
    {
      if (register is live in basic block)
      {
        for (all predecessor basic blocks)
        {
          if (register is live at end of predecessor basic block)
          {
            set DISTANCE to number of cycles latency continues in current basic block
          }
        }
      }
    }
  }
  Insert DISTANCE nops at the beginning of current basic block
}
```

In this function we traverse all hard registers, checking for each basic block if they are live upon entry. This is done using the GCC function `df_get_live_in`, which returns a bitmap of all registers that are live upon basic block entry. We check the bit corresponding to the current hard register and, if true, we check all instructions in the current basic block. We proceed to check all possible predecessor basic blocks to see if the hard register is live upon exit. For those basic blocks we traverse all instructions once again to find any instructions that set this register. Now we have a number of instruction pairs that are dependent on each other, and for each we check if the number of cycles between the set and the use of the register is equal to the latency needed. This latency is calculated through `insn_latency1`, a target-specific function that internally calls the GCC generated function `internal_insn_latency`, that checks the minimum latency between two instructions. If the current latency is smaller than needed, the difference is stored in the `distance` variable.

In case one of the predecessor basic blocks is empty, a predecessor of that one can cause problems. For this a check is made for each predecessor basic block, and if it's empty, also all it's predecessors are checked.

If for a particular instruction the `distance` variable is non-zero at the end, we insert `distance` of nops to satisfy all the inter-basic block dependencies.

To calculate the latency needed between two instructions, it is not enough to simply take the `length`-attribute of each instruction. Bypasses defined in the machine description can cause this value to differ dependent on the consumer instruction. For the scheduler, GCC internally generates the function `insn_latency`, that also takes into account all these bypasses. One drawback is the fact that `insn_latency` does more than just checking the latency. It can return a latency of 0 incorrectly depending on the instruction scheduled in the current cycle. Internally, `insn_latency`

calls `internal_insn_latency`, and this function returns the value we need. In order to make `internal_insn_latency` accessible from our back end we had to make a minor change in the mainline. In the file `genautomata.c` we remove the keyword `static` from the generation of the function declaration of `internal_insn_latency`, so the function will also be available outside its own source file.

3.3.7 Resource conflict avoidance

Besides violating dependency constraints, also resource conflicts can be a problem when an instruction in a basic block does not finish before the next basic block begins. One instruction likely to cause a problem like this is the `div` instruction, that has the time shape

```
(define_reservation "c_salu_div" "nothing, r_salu_rp, r_salu + r_salu_div + r_salu_rp,
                                  r_salu_div * 4, (r_salu_div + r_salu_r_wp) * 5")
```

Especially the use of the `SALU` read port in the final cycles can cause problem with other `SALU` instructions scheduled in a next basic block. In order to counter this another machine dependent pass was added to the *machine dependent optimizations* phase. In this pass, called `evp_div_to_end_bb_pass` all basic blocks are traversed and checked for instructions with the `c_salu_div` time shape. When such an instruction is encountered the number of instructions until the end of the basic block is counted, and checked against the minimum number needed in order to guarantee correct execution. If the required distance is not sufficient, nops are inserted right after the `div` instruction.

```
static void evp_div_to_end_bb_pass(void)
{
  basic_block cur_bb;
  rtx cur_insn, tmp_nop;
  const int dist_required = 10;
  int dist, i;

  compute_bb_for_insn();

  FOR_EACH_BB(cur_bb)
  {
    FOR_BB_INSNS(cur_bb, cur_insn)
    {
      if (recog_memoized(cur_insn) > 0
          && NONJUMP_INSN_P(cur_insn)
          && get_attr_type(cur_insn) == TYPE_DIV)
      {
        dist = insn_cycle_count_left(cur_bb, cur_insn);

        if (dist < dist_required)
        {
          rtx end_of_cycle = cur_insn;

          while (INSN_P(NEXT_INSN(end_of_cycle))
                 && GET_MODE(NEXT_INSN(end_of_cycle)) != TImode)
            end_of_cycle = NEXT_INSN(end_of_cycle);

          for (i = 0; i < dist_required - dist; i++)
          {
            rtx tmp_nop = make_insn_raw(const0_rtx);

            add_insn_after(tmp_nop, end_of_cycle, cur_bb);
```

symbol	resulting output
%	%
#	comments containing information about the current instruction
)	a predicate register in case of a predicated instruction

Table 3.4: Valid characters to follow the % symbol in the assembly template.

```
            PUT_MODE(tmp_nop, TImode);
          }
        }
      }
    }
  }
  return;
}
```

Because increasing the time between two instructions does not cause any dependencies that were previously valid to be invalidated, we can safely add these nops to the instructions stream. At the moment the `div_to_end_bb_pass` does not check whether instructions in subsequent basic blocks actually use the `SALU` read port, this approach is a conservative one. Performance could be improved by involving the instructions in subsequent basic blocks in deciding the number of nops to insert.

3.4 Emitting assembly code

The goal of the compiler is to output EVP assembly code, ready to be processed by the EVP assembler. There are several aspects that need to be processed in order to correctly output the assembly. First all basic symbols need to be defined, such as the syntax to describe a data section, read-only section, start of a function, labels and so on. These are defined through a number of macros, for example `DATA_SECTION_ASM_OP`, which in the EVP back end is defined to `"\t.data \"readwrite\"`.

To support the VLIW aspect, each instruction that is issued in parallel with the instruction on a previous line is preceded by a `||` symbol. This symbol is emitted by the target hook `final_prescan_insn`. This hook is called before the assembly string in an RTL rule is emitted and processed. This allows some last minutes changes or additions to the assembly. To generate these parallel instructions, each instruction is checked for their mode, and for all instructions that are not `TImode` (i.e. do not signify the start of a new cycle), `||` is emitted.

After this the string (possibly returned by the C statement) from the instruction definition is processed character by character. Every time a `%` symbol is encountered, the output depends on the symbol following it. Any number results in the output of the corresponding operand, but several other symbols are also valid, as defined in the macro `PRINT_OPERAND_PUNCT_VALID`. These are listed in table 3.4.

For every `%` followed by a number, GCC outputs the according operand. The exact formatting for these operands can differ depending on the mode of the operand. For this the `PRINT_OPERAND` macro is used, which is tied to the function of the same name which correctly outputs the operand. Also it is possible to format the output of an operand by inserting a letter between the `%` and the digit. Valid letters for this are listed in table 3.5.

symbol	resulting formatting
C	comparison operator
N	reverse of comparison operator
H	upper 16 bits of integer
L	lower 16 bits of integer
S	symbol reference
A	high part of SImode register
B	low part of SImode register
U	unsigned integer
G	signed integer

Table 3.5: Valid characters to specify the output of an operand.

In order to assist both the programmer and compiler developer to better analyze the generated code, a comment containing some details is emitted at the end of the line of each assembly instruction. To properly align these comments, we keep track of every character that is emitted into the output stream. For this a global variable current_pos_in_line is introduced, that is reset whenever a new instruction is processed for output. For this GCC has the target ASM_OUTPUT_OPCODE available, that (when defined) is called just before GCC processes the assembly field in the instruction rtx. As a parameter a pointer is passed that can be incremented by the hook and then returned to the caller, and the compiler will resume outputting the string from the pointer location. In this case we merely step over all characters in the string, incrementing the value of current_pos_in_line by 1 for each character that is emitted 1-to-1 to the output stream, and skipping all other characters (like %0 or %B1). Finally 8 is added for the leading tab and the function returns with the character pointer unchanged.

Each time a register name, constant, label or symbol reference is emitted, this variable is also increased by the length of the just emitted string. This is done by converting the value to a string if it is not one already via a call to sprintf and then taking its length using the strlen function.

When the comments symbol is then processed, the difference between the alignment position and the number of bytes already emitted is filled up with spaces and a comment with the following syntax is generated:

```
|| ptr_update ptr15, -64            // i-m- SLSU  fft_btail_ii.c:104
```

The comment details the functional unit on which the instruction is executed, along with four flags, the file name and file number that this instruction originated from. The first flag displays an i if the function uses the long immediate field. The third flag is an m if the instruction is one that can be issued on multiple functional units, and the fourth flag is a c in case the instruction is a post-increment addressing instruction. The second field is currently not in use, however adding new flags is not much work.

The information for these flags can easily be extracted from the instruction, but the problem here is that the PRINT_OPERAND macro, which processes the output of these comments, is not passed the RTL instruction, and hence cannot extract this information. To supply this information several global variables are introduced, that are set in the final_prescan_insn. Here the long_imm attribute is checked and the c flag is set if the vector_type attribute is equal to either vlsu_ld_post

or `vlsu_1_post`. The file name and line number is stored by GCC inside each rtx, and can be extracted by simply calling the GCC functions `insn_file()` and `insn_line()`.

To easily pass the functional unit information an enumeration type containing all functional units is defined, and a global variable of that type is set based on the `type` or `vector_type` attribute of the current instruction. In case the instruction can be issued on multiple units (i.e. has type `ptr_update`, `ptr_ofs_move`, `hi_move` or `si_move` or vector type `vmove`) the key is extracted and based on this value the correct functional unit is determined. The flag `m` is set if key is not equal to 2 (the default value).

When `PRINT_OPERAND` outputs the comments, it retrieves the information stored in these global variables to determine the correct values for the functional unit, flags, file name and file number to be printed.

Another thing that makes it easier for the compiler developer to see if a certain addition results in performance improvement is a cycle count added to the assembly output. This is done by adding the following lines to the `final_prescan_insn` function:

```
if (GET_MODE(insn) == TImode
    && insn_code != CODE_FOR_doloop_begin
    && insn_code != CODE_FOR_dohi && insn_code != CODE_FOR_dosi)
{
    if (BLOCK_NUM(insn) != curr_bb)
    {
        /* Start of new Basic block */
        curr_bb = BLOCK_NUM(insn);
        insn_counter = 1;
        fprintf(asm_out_file, "// [%d]\n", insn_counter++);
    }
    else                    // Still in same BB
    {
        fprintf(asm_out_file, "// [%d]\n", insn_counter++);
    }
}
```

This lets the compiler developer or programmer see immediately how many cycles a performance critical loop or other section of code contains. All these improvements results in verbose assembly code, to give an example this is a snippet from one of the generated assembly files for the FFT benchmark, `fft_btail_ii.asm`:

```
// [13]
        cmpneq p1, r10, 0                   // ---- SALU  fft_btail_ii.c:459
||      vshu_bfy vr4, vr3, +4, vm3          // ---- VSHU  fft_btail_ii.c:405   evp_mo_vshu_fftbfy_32
||      vsub16 vr2, vr13, vr14              // ---- VALU  fft_btail_ii.c:374   evp_vsub_16
// [14]
        vshu_bfy vr3, vr6, +4, vm1          // ---- VSHU  fft_btail_ii.c:404   evp_mo_vshu_fftbfy_32
||      vmul_rnd_satc16 vr13, vr2, vr12     // ---- VMAC  fft_btail_ii.c:376   evp_vmul_rndsat_cf16
||      move ptr1, r1r0, p1                 // ---- SALU  fft_btail_ii.c:462   evp_po_u32_to_ptr
||      ptr_update ptr8, -4, p1             // --m- SLSU  fft_btail_ii.c:459   evp_p_add_ptr
// [15]
        vshu_bfy vr9, vr11, +8, vm2         // ---- VSHU  fft_btail_ii.c:359   evp_mo_vshu_fftbfy_64
||      vadd16 vr5, vr4, vr3                // ---- VALU  fft_btail_ii.c:481   evp_vadd_16
||      vmove_vlsu8 vr11, vr10              // --m- VLSU  fft_btail_ii.c:408
||      move ptr9, r3r2, p1                 // ---- SALU  fft_btail_ii.c:461   evp_po_u32_to_ptr
// [16]
        vadd16 vr14, vr1, vr9               // ---- VALU  fft_btail_ii.c:379   evp_vadd_16
||      move r9r8, r1r0                     // ---- SALU  fft_btail_ii.c:442   evp_mov_32
||      vstore_post_update ptr2, ofs1, vr5  // ---c VLSU  fft_btail_ii.c:482
||      vmove_vshu8 vr5, vr7                // --m- VSHU  fft_btail_ii.c:414
```

3.5 Implementing Target-Specific Features

This section details how some features of the EVP are enabled in the EVP-GCC compiler. The implementation of post-increment addressing is an entirely new feature built in as part of the project, and is detailed in section 3.5.2. In sections 3.5.3, 3.5.4 and 3.5.5 we will briefly discuss hardware loop generation, address-based alias analysis and if-conversion. These three parts were either already in place at the start of the project, or added and developed by the other members of the team working on EVP-GCC during the course of this project. Therefore they will be briefly discussed here but fall outside the scope of this thesis project. Branch delay scheduling was partly in place at the start of the project, but was improved and extended as part of the MSc project.

We will start by outlining features that we addressed during the course of this project, followed by the work done by the other members of the team.

3.5.1 Branch delay scheduling

Due to the size of the EVP pipeline, all call and branch instructions are followed by a number of instructions that are always executed. The number of these *branch delay slots* depend on whether the target of the branch is an immediate address or is stored in a register. GCC supports scheduling with branch delay slots, but because it was not designed to support exposed pipeline architectures (architectures without hardware interlocks) the branch delay scheduler does not insert nops into the instruction stream. This causes the GCC branch delay scheduler to generate incorrect code for the EVP. To avoid this the branch delay scheduling is implemented in a machine specific pass. GCC allows the back end to include target-specific passes in the *machine dependent optimizations* pass.

This `evp_fill_delay_slots` function implements this pass. It moves all call or branch instructions backward through the instruction stream in order to push as many instructions as possible before it into the delay slots, without breaking any dependencies or causing any resource conflicts. As far as dependencies between other instructions go, increasing the distance between two instructions never causes a conflict, so we can safely move the jump instruction backward.

The delay slot filler pass traverses all instructions, and when a call or branch instruction is encountered, we move backward through the instruction stream. All instructions encountered can be moved into the branch delay slots as long as the following constraints are met.

1 Instructions that are part of previous basic blocks cannot be moved into a branch delay slot because it cannot be guaranteed that these instructions would be executed before the call or branch instruction in question. Nor can instructions already in branch delay slots of previous branch instructions be moved into a delay slot of another branch. This would effectively move the branch instruction into a delay slot of another branch, which is not allowed.

2 Moving an instruction into a delay slot must not violate any previously satisfied data dependencies. Since increasing the distance between two instructions never violates any dependencies between them this poses no problem.

3 Resource constraints also must not be violated when moving an instruction into a delay slot. In most cases this will not cause any problems, but for several instructions this might cause conflicts. The only cases where a problem occurs is when two instructions with different resource usage patterns use the same resource in different cycles. The only case where this may occur

at the moment is a division instruction which uses the `c_salu_div` reservation pattern (see section 3.3.7), which uses the `r_salu_r_wp` in the last cycles. To avoid any conflicts this resource is also reserved one cycle before it is actually needed. To give an example, suppose an instruction with the `c_salu_1` reservation pattern is scheduled 5 cycles later than a `div` instruction. Now this instruction is moved into a branch delay slot. This means the instruction now effectively uses the `r_salu_r_wp` 6 cycles after the div instruction uses it. This results in a resource conflict. If we reserve the functional unit one cycle earlier than actually needed in the `c_salu_div` reservation pattern, increasing the distance will still result in a correct schedule.

4 Finally, any instruction that sets the register which holds the branch target can not be moved into a delay slot, because this would violate the data dependency between the branch instruction and the instructions that sets the register holding the target. This problem was solved by not filling any branch delay slots if an instruction was encountered that set the target register. Later on we improved this by taking into account the distance required between these two instructions and calculating the number of instructions that could be moved into the delay slots without violating this constraint. We will describe later on how this was implemented.

Any instructions that satisfy all above criteria can now be moved into a delay slot. First the number of instructions that do not violate any of the conditions above is calculated, and then the branch instruction was moved before the last instruction that was moved into a delay slot. The remaining delay slots are filled with nops.

To check if the first constraint is satisfied, we stop checking instructions when we encounter a note that marks the beginning of a basic block. These notes are generated automatically by GCC.

To meet the second condition we must somehow mark instructions that are moved into branch delay slots. This is done through the introduction of a new instruction pattern:

```
(define_insn "mark_for_delay_slot_filler"
  [(const_int -1)]
  ""
  "// ----------"
)
```

This mark is emitted after the last instruction in the delay slots. This way if we walk backwards over the instructions and encounter a `mark_for_delay_slot_filler` instruction, we stop. This guarantees that no instructions are moved from a delay slot of one branch into the delay slot of another.

The third condition is satisfied through changes in the resource reservation patterns, as mentioned earlier.

The last condition is satisfied by checking each instruction we encounter for the register that contains the branch target. If the branch target is an immediate address, this condition is automatically met. Originally[6] up to the number of instructions that could be moved into a branch delay slot were checked. In rare cases, this could result in violation of the data dependency. Table 3.6 describes a situation where moving 7 instructions into branch delay slots causes this problem, without the branch delay scheduler actually encountering an instruction that sets the register containing the branch target.

[6]The state of the compiler at the moment this project was started.

Code before BDS	(Incorrect) code after BDS
`load ra`	.
$insn_1$	.
$insn_2$	.
$insn_3$	.
$insn_4$	.
$insn_5$	.
$insn_6$	.
$insn_7$	`load ra`
$insn_8$	$insn_1$
`branch ra`	`branch ra`
	$insn_2$
	$insn_3$
	$insn_4$
	$insn_5$
	$insn_6$
	$insn_7$
	$insn_8$

Table 3.6: Incorrect branch delay scheduling.

Instructions 2 through 8 are checked, satisfy all conditions and are moved into a delay slot, leaving the `load ra` instruction too close to the branch instruction. The value in `ra` is therefore not loaded when the branch target is calculated, and the program counter is set to the *old* value of `ra` which is the instruction after the branch delay slots of the last call made. This causes the processor to end up in an endless loop. To solve this we expand the check to include up to 4 more cycles than the number that can be moved into a delay slot. If now the instruction setting the register in question is encountered, we calculate the number of instructions that are required between the two and step forward over that many instructions again until we are at the earliest instruction that can safely be moved into a delay slot.

3.5.2 Post-increment addressing

The EVP supports post-increment addressing, meaning we can do a load or store, and a pointer update instruction in a single instruction. This is a considerable performance improvement[7], because when a pointer walk is used in the code to be compiled, it can be achieved in half the number of instructions. Where normally a `[v]load_ofs` or `[v]store_ofs` and a `[v]ptr_update` instruction are needed, using two `SLSU`, two `VLSU` or one of both functional units, this can now be achieved in a single instruction that uses the same resources as a single `[v]load_ofs`.

Initially, support for post-increment and post-decrement addressing was implemented through definition of peephole patterns, which allow the peephole optimization pass to merge two subsequent instructions that match a certain RTL expression and operands into a single RTL expression with the same functionality. The peephole optimizer has one limitation however. It only merges two operations

[7] depending on the source code, of course.

that are immediately following each other in the instruction stream. This means that two vector load instructions followed by two pointer update instructions only result in one post-increment load. This behaviour is intentional, and allows the peephole optimizer to merge any two consecutive machine instructions. This includes two instructions entirely unrelated from a data-flow point of view.

This limitation prevents full use of the post-increment capabilities of the EVP, so other ways had to be found to implement this feature.

GCC has built-in support for pre-increment, post-increment, pre-decrement and post-decrement addressing that can be enabled by setting the macros

```
#define HAVE_POST_INCREMENT 1
#define HAVE_POST_DECREMENT 1
#define HAVE_POST_MODIFY_DISP 1
#define HAVE_POST_MODIFY_REG 1
```

in `evp.h`. The `HAVE_POST_INCREMENT` and `HAVE_POST_DECREMENT` macros enable the generation of instructions that load a value, and then update the base register with a value that is the size of the register loaded from memory. To enable post-update instructions with different values the macro `HAVE_POST_MODIFY_DISP` is set. This allows any arbitrary value to be used for post-updating the base register. By setting the macro `HAVE_POST_MODIFY_REG` we allow GCC to generate RTL that uses a register to store the value that is added to or subtracted from the base register.

With this enabled GCC will generate these kinds of RTL instructions in the *auto-inc-dec* pass. This pass will try a combine two separate instructions into one, like the following example from the compilation of `fft_btail_vvi.c`. Two instructions after the first combine pass

```
(insn 157 155 158 5 fft_btail_vvi.c:287 (parallel [
            (set (mem:V16HI (reg/v/f:PSI 188 [ outa ]) [2 S32 A256])
                (reg:V16HI 285))
            (clobber (const_int 2 [0x2]))
        ]) 271 {__vec_store_ptr_v16hi_clobber_pattern} (expr_list:REG_DEAD (reg:V16HI 285)
        (nil)))

(insn 158 157 159 5 fft_btail_vvi.c:287 (parallel [
            (set (reg/v/f:PSI 136 [ outa149 ])
                (plus:PSI (reg/v/f:PSI 188 [ outa ])
                    (const_int 32 [0x20])))
            (clobber (const_int 2 [0x2]))
        ]) 444 {addpsi_3333} (expr_list:REG_DEAD (reg/v/f:PSI 188 [ outa ])
        (nil)))
```

become a single instruction during the *auto-inc-dec* pass:

```
(insn 157 256 255 5 fft_btail_vvi.c:287 (parallel [
            (set (mem:V16HI (post_inc:PSI (reg/v/f:PSI 136 [ outa149 ])) [2 S32 A256])
                (reg:V16HI 285))
            (clobber (const_int 2 [0x2]))
        ]) 282 {vec_store_post_inc_v16hi_clobber} (expr_list:REG_INC (reg/v/f:PSI 136 [ outa149 ])
        (expr_list:REG_DEAD (reg:V16HI 285)
            (nil))))
```

The `clobber` side-effect in the instruction is needed by the scheduling pass, for more information on this see 3.3.4. It has no influence on the generation of post-increment or -decrement instructions, as long as we define the `post_inc` instructions (in this case) also with a `clobber` in parallel. If we

did not, GCC would see this as losing information during the combination, and would not generate the combined instruction.

The `define_insn` needed to enable the transformation (for this example) is the following:

```
(define_insn "vec_store_post_inc_v16hi_clobber"
 [(set (mem:V16HI (post_inc:PSI (match_operand:PSI 0 "base_register_operand" "b,b")))
       (match_operand:V16HI 1 "vr_or_ir_register_operand" "Yvr,Yir"))
  (clobber (match_operand:SI 2 "const_int_operand" "=i,i"))
]
 ""
 "vstore_post_update %0, +32, %1%#"
 [(set_attr "vector_type" "vlsu_1_post")
  (set_attr "long_imm" "no")]
)
```

Since these instructions are only generated during the *auto-inc-dec* pass, there is no need for an expansion rule for these post-modify patterns.

A complication resulting from this implementation is the fact that the post-increment instructions have a latency of 3. However, this only goes for the load of the (vector) register. The post-incremented value of the `ptr` register is ready the next cycle. Therefore we had to add another bypass definition (see section 3.3.2) to the machine description file specifying the latency between the post-increment load instructions and all instruction classes that can have the incremented pointer register as an input operand.

To give an example where implementing the post-increment addressing mode can generate improvement consider the following code, a pointer walk loading four vectors from an array whose address is contained in register `ptr0`, along with updating four other pointer registers with the values in `ofs1` through `ofs4`.

```
\\ [1]
        vload               vr0, ptr0          // VLSU
||      ptr_update          ptr0, +32          // SLSU
\\ [2]
        vload               vr1, ptr0          // VLSU
||      ptr_update          ptr0, +32          // SLSU
\\ [3]
        vload               vr2, ptr0          // VLSU
||      ptr_update          ptr0, +32          // SLSU
\\ [4]
        vload               vr3, ptr0          // VLSU
||      ptr_update          ptr0, +32          // SLSU
\\ [5]
        ptr_update          ptr1, ofs1         // SLSU
||      vptr_update         ptr2, ofs2         // VLSU
\\ [6]
        ptr_update          ptr3, ofs3         // SLSU
||      vptr_update         ptr4, ofs4         // VLSU
```

Introducing post-increment addressing generates the following code:

```
\\ [1]
        vload_post_update   vr0, ptr0, +32     // VLSU
||      ptr_update          ptr1, ofs1         // SLSU
\\ [2]
        vload_post_update   vr1, ptr0, +32     // VLSU
||      ptr_update          ptr2, ofs2         // SLSU
```

```
\\ [3]
        vload_post_update   vr2, ptr0, +32     // VLSU
||      ptr_update          ptr3, ofs3         // SLSU
\\ [4]
        vload_post_update   vr3, ptr0, +32     // VLSU
||      ptr_update          ptr4, ofs4         // SLSU
```

As we can see, using the `vload_post_update` instruction frees up the SLSU unit to perform the pointer update thereby reducing the total number of instructions from 12 to 8, and reducing the number of cycles from 6 to 4.

For completeness purposes, we will also discuss the implementation of support for hardware loop generation, if-conversion and address-based alias analysis. This work was not part of the thesis project, but was worked on by the other members of the EVP team.

3.5.3 Hardware loops

As mentioned earlier, every branch and jump instruction in the EVP has a number of delay slots that are always executed. Because there are not always enough instructions available to fill all these delay slots, due to data dependencies or a limited number of instructions in the basic block, nops need to be inserted in one or more of these delay slots. This causes a significant decrease in performance. To avoid this, the EVP has several special instructions to make loops more efficient. These instructions store the number of iterations, the start and end address of a loop in registers. This way the 5 cycles it would normally take to calculate the address to jump back to at the end of the loop and whether it is taken or not is all known in advance, eliminating the need for branch delay slots.

GCC has built-in support for the generation of hardware loops. The pass `pass_rtl_doloop` marks the start and end of these hardware loops by generating two instructions called `doloop_begin` and `doloop_end`, which mark the begin and end point of the loop. These patterns are supplied a number of operands, which contain the number of iterations (or the register which holds the value at runtime), the label to jump back to and the estimated number of iterations if it can be determined by the compiler. This information can be used to emit several different preparation instructions if necessary.

GCC generates hardware loops in a *do-while* format. This means the test condition is evaluated before the end of the loop. The EVP expects hardware loops in a *while-do* format, where the condition is evaluated before the first iteration of the loop. This requires some rewriting of the hardware loops generated by GCC. This is done in the *machine dependent optimizations* pass. Here we implement the pass called `evp_hloop_pass`, which checks the validity of the hardware loop through the supporting function `evp_hloop_valid_p`. This function checks the loop body for absence of any jump instructions. The EVP ISA specifies that no jump instructions can be contained in the body of a hardware loop, nor can any jump from outside the loop body jump to an address inside a hardware loop. This function also reorders the control flow in such a way that a valid hardware loop can be generated if previous optimizations have placed the order of basic blocks in such a way that the generation of a hardware loop is not possible.

To give a quick impression of the reordering required to achieve valid control flow for hardware loop generation consider the following example.

A simple loop that uses an internal variable that can easily be expressed in terms of the loop counter, the *loop optimizations* pass (more specifically the *induction variable optimization* performed in that pass) rewrites the loop to the order depicted in the left-hand side of figure 3.2. Because basic block 4

is not empty we cannot directly turn this into a hardware loop by adding the appropriate instructions before and after the loop body.

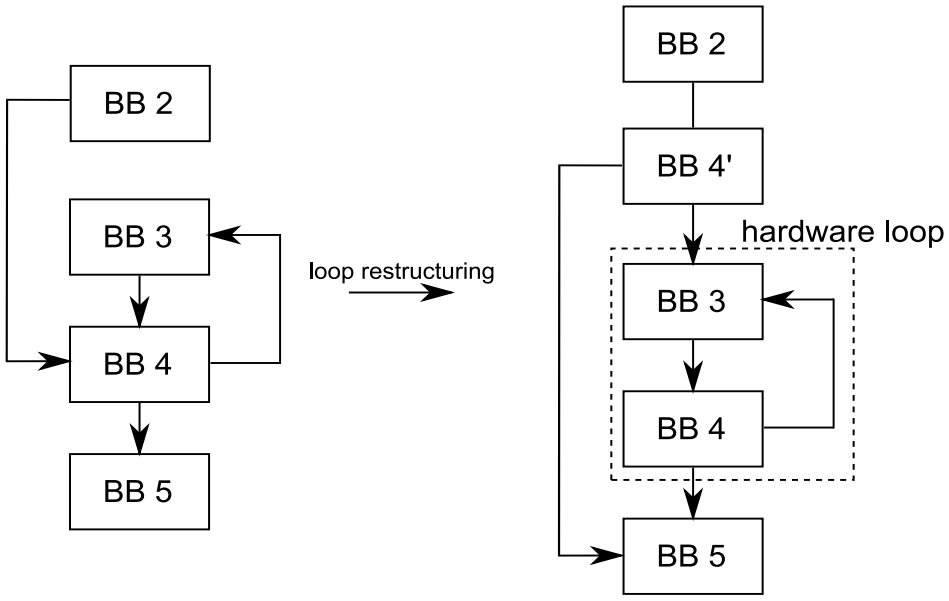

Figure 3.2: Loop restructuring to enable hardware loop generation

After a transformation to the order depicted at the right-hand side of figure 3.2, the hardware loop can be generated by adding instructions before basic block 3 and after basic block 4. To enable this a copy of basic block 4 is made and moved before basic block 3. This basic block 4' has outgoing edges to basic blocks 5 and 3, just like basic block 4 but is executed only once. This transformation allows the generation of a low-overhead hardware loop in the dotted area.

3.5.4 Alias analysis

In order to improve performance, several passes that operate on the RTL level use *alias analysis information* which determines whether two different memory accesses might point to the same memory location, thereby generating a data dependency between these instructions. This information most of the time cannot be exactly determined, but several methods exist to determine if two memory accesses definitely do *not* point to the same address.

On the GIMPLE level, GCC performs *type-based alias analysis*. This method disambiguates two memory accesses by using the rule specified in the ANSI C standard that two pointers pointing to different types can never point to the same memory location[36]. Also on the RTL level alias analysis information is calculated. This is flow-sensitive *points-to* alias analysis. This analysis method determines for each variable to which memory space it points. Two pointers that point to different memory spaces cannot alias.

The passes operating on the RTL level could benefit from more powerful alias analysis, and for this address-based alias analysis has been introduced. This method of alias analysis was proposed in [30]. This method of alias analysis was implemented by the other members of the team working on EVP-GCC and discussing it falls outside of the scope of this project.

3.5.5 If-conversion

Another feature supported by the EVP architecture is predicated execution. This is conditional execution of an instruction, implemented by specifying a predicate register to the assembly instruction. This way `if` statements in the code can be translated to a predicates assembly instruction. For example loading a value from memory if a certain expression evaluates to *true* avoids branches in the assembly code and the accompanying overhead. This increasing the performance significantly because of a lot of branch delay slots can be avoided which possibly cannot all be filled with instructions, introducing unnecessary nops. Generating these predicated execution for `if`-statements is called *if-conversion*.

As an example, consider a simple if-statement:

```
if (a == 1)
  b = 2;
a = 3;
```

Without if-conversion, this would result in the following assembly code (`r0` holds `a`, `r1` holds `b`):

```
        cmpneq  p1, r0, 1
        br      .L0, p1
        nop
        nop
        nop
        nop
        nop
        move    r1, 2
.L0
        move    r0, 3
```

When if-conversion is applied, the same code would result in:

```
        cmpeq   p1, r0, 1
        move    r1, 2, p1
        move    r0, 3
```

As we can see, a branch instruction and 5 delay slots can be avoided. This can cause a huge increase in performance. If the if-statement is part of a long piece of code that allows many instructions to be moved into a delay slot this penalty is reduced, but still at least one extra cycle for the branch is needed.

GCC supports if-conversion and its implementation was already in place at the start of this project. Therefore we will not discuss the exact implementation beyond the details mentioned here.

3.6 Summary

In this chapter we detailed the steps taken to specify all machine properties in the back end, the additions needed to enable expansion of Tree-SSA statements into RTL for vector operations, as well as the changes made to the generation of function calls, function argument passing and function prologue and epilogue generation. We presented how the instruction scheduler was extended to schedule vector instructions and how the scheduler manages to schedule instructions that contain operations that can be performed on several different functional units. We presented a new approach to the scheduling of

instructions which enabled us to split the finite state automaton generated by the instruction scheduler even though information from all automata was required for scheduling some instructions. We presented the way assembly is emitted and our additions to the verbose comments in the assembly code, as well as our work on branch delay scheduling and post-increment addressing. We also included the work done by the other members of the team for completeness purposes. s

4 Experimental Results

In this chapter we will discuss the results obtained from all benchmarking done on the EVP-GCC compiler. First we will discuss the correctness of the generated code through application of a modified version of the DejaGnu testsuite in section 4.1.

Due to contractual obligations with the company which produces the current production compiler, we cannot disclose any comparison of the performance of benchmarks run with GCC as compared to the production compiler in this document. This information will be supplied to the members of the graduation committee in a separate document.

We also present the experimental results obtained from a measurement performed by running the vector benchmark with the EVP-GCC compiler in which the new implementation that allows the scheduling of semantically equivalent instructions on different functional units compared to the compiler without this addition. This is presented in section 4.3. Finally we will present the findings of the experiment we performed regarding the possible improvements autovectorization will bring when applied to the EEMBC Telecom benchmark code. These experiments were performed on the AltiVec back end, and are presented in section 4.4.

4.1 Correctness benchmarks

The EVP-GCC compiler needs to be checked for correctness of the generated code. Builds of the GCC compiler are often validated using the open-source DejaGnu[27] testing framework. To validate the EVP-GCC compiler a modified version of the internal GCC testsuite was written and run using DejaGnu. These testcases are supplemented with the benchmarks from the EEMBC-Telecom benchmarking suite, which are also used for scalar performance benchmarking. To run the modified GCC testsuite we assign the compiler executable `cc1`, which is the main compiler file. We used the already available in-house assembler, linker and simulator. For correctness benchmarking of the vector part, 78 additional testcases containing all implemented EVP-C intrinsics were hand-written and added to the EVP-GCC testsuite. To give an example, the testcase for the EVP-C intrinsic `evp_vadd_16` is the following:

```
/* { dg-do run } */
/* { dg-options "-O2 -Iconfig/evp/include" } */

#include "evp.h"

extern void exit (int);

  v_t v1 = evp_v_init_16(1, 0, 0, 0, 0, 0, 0, 0, \
   0, 0, 0, 0, 0, 0, 0, 0);

  v_t v2 = evp_v_init_16(1,  2,  3,  4,  5,  6,  7,  8, \
         9, 10, 11, 12, 13, 14, 15, 16);
```

```
  v_t v3 = evp_v_init_16(61, 60, 59, 58, 57, 56, 55, 54, \
         53, 52, 51, 50, 49, 48, 47, 46);

int main()
{
  int i;

  v1 = evp_vadd_16(v2, v3);

  for (i = 0; i < 16; i++)
  {
    int16_t p_elem = evp_vget_elem_16(v1, i);
    if (p_elem != 62) abort();
  }

  return 0;
}
```

This testcase simply adds two vectors and checks each individual element and aborts (exit with a non-zero return value) when one of the elements is not correct.

Including the vector testfiles a total of 1039 files are used for testing, running a total of 4546 tests. The results of running the EVP-GCC testsuite are listed below. Please note that all passed tests do not appear in the output. Failed tests are denoted by `FAIL`, and tests that are passed unexpectedly (tests that should fail but pass instead) are marked with `XPASS`.

```
FAIL: gcc.dg-modified3/20020312-2.c execution test
FAIL: gcc.dg-modified3/20031223-1.c (test for excess errors)
FAIL: gcc.dg-modified3/20040311-2.c (internal compiler error)
FAIL: gcc.dg-modified3/20040311-2.c (test for excess errors)
FAIL: gcc.dg-modified3/20041219-1.c (test for excess errors)
FAIL: gcc.dg-modified3/20050121-1.c  (test for warnings, line 8)
FAIL: gcc.dg-modified3/20050121-1.c (test for excess errors)
FAIL: gcc.dg-modified3/940510-1.c (test for excess errors)
FAIL: gcc.dg-modified3/asm-fs-1.c scan-assembler-not \\*_bar
FAIL: gcc.dg-modified3/bitfld-12.c  (test for errors, line 10)
FAIL: gcc.dg-modified3/bitfld-12.c (test for excess errors)
FAIL: gcc.dg-modified3/builtin-apply1.c (internal compiler error)
FAIL: gcc.dg-modified3/builtin-apply1.c (test for excess errors)
FAIL: gcc.dg-modified3/builtin-return-1.c (internal compiler error)
FAIL: gcc.dg-modified3/builtin-return-1.c (test for excess errors)
FAIL: gcc.dg-modified3/builtins-13.c (test for excess errors)
FAIL: gcc.dg-modified3/c99-bool-1.c (test for excess errors)
FAIL: gcc.dg-modified3/cleanup-5.c (internal compiler error)
FAIL: gcc.dg-modified3/cleanup-5.c (test for excess errors)
FAIL: gcc.dg-modified3/ftrapv-1.c (test for excess errors)
FAIL: gcc.dg-modified3/ftrapv-2.c (test for excess errors)
FAIL: gcc.dg-modified3/funcorder.c scan-assembler-not link_error
FAIL: gcc.dg-modified3/init-string-1.c (test for excess errors)
FAIL: gcc.dg-modified3/inline-1.c scan-assembler-not xyzzy
FAIL: gcc.dg-modified3/intmax_t-1.c (test for excess errors)
FAIL: gcc.dg-modified3/nrv1.c execution test
XPASS: gcc.dg-modified3/overflow-warn-3.c constant (test for bogus messages, line 42)
XPASS: gcc.dg-modified3/overflow-warn-4.c constant (test for bogus messages, line 42)
FAIL: gcc.dg-modified3/pr16973.c (internal compiler error)
FAIL: gcc.dg-modified3/pr16973.c (test for excess errors)
FAIL: gcc.dg-modified3/sibcall-3.c execution test
FAIL: gcc.dg-modified3/sibcall-4.c execution test
FAIL: gcc.dg-modified3/simd-3.c (internal compiler error)
FAIL: gcc.dg-modified3/simd-3.c (test for excess errors)
FAIL: gcc.dg-modified3/struct-by-value-1.c execution test
FAIL: gcc.dg-modified3/test1.c execution test
FAIL: gcc.dg-modified3/transparent-union-1.c (test for excess errors)
```

```
FAIL: gcc.dg-modified3/transparent-union-2.c (test for excess errors)
FAIL: gcc.dg-modified3/transparent-union-4.c  (test for warnings, line 9)
FAIL: gcc.dg-modified3/transparent-union-4.c (test for excess errors)
FAIL: gcc.dg-modified3/transparent-union-5.c (test for excess errors)
FAIL: gcc.dg-modified3/ucnid-2.c (test for excess errors)
FAIL: gcc.dg-modified3/uninit-1.c uninitialized variable warning (test for bogus messages, line 16)
FAIL: gcc.dg-modified3/uninit-3.c uninitialized variable warning (test for bogus messages, line 11)
FAIL: gcc.dg-modified3/uninit-9.c uninitialized variable warning (test for bogus messages, line 26)

=== gcc Summary ===

# of expected passes 4481
# of unexpected failures 43
# of unexpected successes 2
# of expected failures 30
# of unresolved testcases 8
# of unsupported tests 151
```

We have a total of 43 unexpected failures in the testsuite. These failures can be attributed to a number of things. Most failures are due to C constructs the target doesn't support, for example the sibling call construct and some move instructions not yet impl[18]emented, for example in the `builtin-apply1` and `builtin-return-1` testcases. These tests use the GCC builtin function `__builtin_apply` and `__builtin_return` which try to save *all* registers on the stack, even the ones for which no load and store instruction patterns have been written. Other failures are due to the fact that not all data types in the EVP back end have the same width as data types in regular C, and macros containing the maximum and minimum values for these data types do not contain the correct values. There are also a number of testcases that cause spurious warnings, and some testcases that fail on execution due to some small bugs still present in the compiler. However, these failures are all non-critical for the correctness of the compiled benchmarks.

4.2 Performance benchmarks

Due to contractual obligations, the sections containing the scalar and vector benchmark results will be supplied to the thesis committee seperate from this report.

4.3 Impact of scheduling semantically equivalent operations on different functional units

In this section we will present the experimental results of a comparison between the EVP-GCC that can schedule semantically equivalent operations on multiple functional units and the EVP-GCC compiler without this feature. For example a move from one `HImode` register to another `HImode` register can be scheduled on the `SALU` or the `SLSU`.

We presented the way the benchmarks are run to obtain these numbers in the section supplied seperate from this report, and because those sections are not publically available we will repeat the methods used here.

To test the performance of the vector support added to GCC, a benchmark was chosen that heavily used vector data. This benchmark is a variable point FFT algorithm, that is run on six data sets of

benchmark	without	with	improvement
FFT_128	594	574	- 3.37 %
FFT_256	1010	986	- 2.38 %
FFT_512	2144	2032	- 5.23 %
FFT_1024	4358	4134	- 5.14 %
FFT_2048	8926	8606	- 3.59 %
FFT_4096	20044	18892	- 5.75 %
Total	37076	35224	- 5.00 %

Table 4.1: Cycle count with and without scheduling semantically equivalent operations

different sizes. The data sets consist of 128, 256, 512, 1024, 2048 and 4096 data points. These FFT's are written in EVP-C, and use several pragmas that are not yet supported by GCC (see section 2.2 of the main report), such as specifying to which register class a variable should belong, and specifying a minimum number of iterations for a loop. The following pragmas are used in this benchmark, and are removed via `#defines` when the code is compiled with GCC:

```
__DataNoInit
__UnitDefaultSectionData( name )
__UnitDefaultSectionConst( name )
__UnitDefaultSectionNoInit( name )
__UnitDefaultSectionFun( name )
__DataSection( name )
__FunSection( name )
__DataAt( ofs )
__FunReturnAt( ofs )
__M1
__M2
__M3
__M4
__M5
__M6
__M7
__M8
__LoopNoHwLoop
__LoopNoSWP
__LoopIterMin( n )
__LoopIterMax( n )
__FunHwLoopLevels( l )
__FunHwUseLoopLevels( l )
```

All FFTs are run with the flags `-DSATURATE`, to enable saturated adds, and `-DSWPS`, which enables manual software pipelining in the `btail_ii` file.

The simulator (`vclsim`) is run while the environment variable `$EVP_NO_CACHE` is defined, which causes the simulator to run the code without checking for pipeline stalls when the scalar cache reports a miss. This is done to examine the results of the compiler in a more isolated environment. Performance hits due to cache misses are a factor that is not taken into account at the time of scheduling, and can therefore sometimes influence the results in unexpected ways. To eliminate this factor the scalar cache is turned off. In some cases it can even occur that inserting a nop at certain point in a program results in a cache hit instead of a miss in an instruction later on, thereby actually *improving* performance instead of decreasing as would be expected.

The performance in terms of cycle count of the EVP-GCC with and without this feature is listed in table 4.1. The column marked 'with' contains the results for the compiler with the ability to schedule semantically equivalent operations on different functional units and the column marked 'without' contains the cycle counts of the compiler without tis feature. We can see that for all FFT sizes in the benchmark, the availability of multiple units to schedule these operations on results in a gain of 2% to 5%. The total execution time (measured by running the entire benchmarkis reduced by 5%. This indicates that implementing our approach to scheduling semantically equivalent instructions on multiple functional units brings a significant advantage.

4.4 Exploration of the possibilities of autovectorization

In this section we will discuss enabling the autovectorizer of GCC. Autovectorization is a process where the compiler transforms a loop that contains scalar operations on consecutive data elements in memory (an array for example) into a loop that performs the same operations on vectors of data. Autovectorization is a feature that was first proposed in [13] by Dorit Naishlos and was worked on in a the *autovectorization* branch [17]. A branch is a copy of a certain version of the compiler (the version at the *branch point*) to which certain additions or changes are made, allowing compiler developers to implement this new feature without causing the main version of the compiler to become unstable. When the features are tested and the branch is stable, a branch can be *merged to mainline*, which means the features are implemented in the GCC mainline and will be included in the next version to be released.

The current progress of the autovectorization branch can be found at [17], and can only vectorize loops which have a single basic block as the loop body. The loops in the EEMBC Telecom benchmark files consist of several algorithms (autocorrelation, convolutional encoder, bit allocation, FFT, inverse FFT and a Viterbi decoder) which operate on large amounts of data in small inner loops. We tested the autovectorizer by compiling the EEMBC Telecom code for the AltiVec [18] back end, as this is a vector architecture and should be able to benefit from autovectorization.

We discovered that the degree of autovectorization for the EEMBC Telecom benchmark code was merely restricted to a few data initialization loops, which constitute only a small part of the total execution time of the code. The loops that take the most time to execute (the *kernels*) contain operations that the autovectorizer cannot effectively transform to a vector representation. This is due to the fact that the kernels contain conditional statements, which result in a loop body that consists of multiple basic blocks. Since the operations do not operate on *all* data in a certain data set, the vectorizer cannot effectively find a way to rewrite this into vector operations. For example an array that updates only all *even* elements cannot be simply replaced by a vector operation with vectors containing a consecutive block of array elements, as all odd elements would be updated as well.

We will give a small example. Consider the following code that operates on two arrays A and B, and results in the array C which contains $Ai + Bi$ for all even elements and just Ai for all odd-numbered elements. A loop performing this operation could consist of the following code:

```
for (i = 0; i < array_length; i++)
{
  if (i & 1)
    C[i] = A[i] + B[i];
  else
    C[i] = A[i];
}
```

When compiled, this results in a loop with three basic blocks, as shown in figure 4.1, in which the loop body consist of the basic blocks 1, 2 and 3.

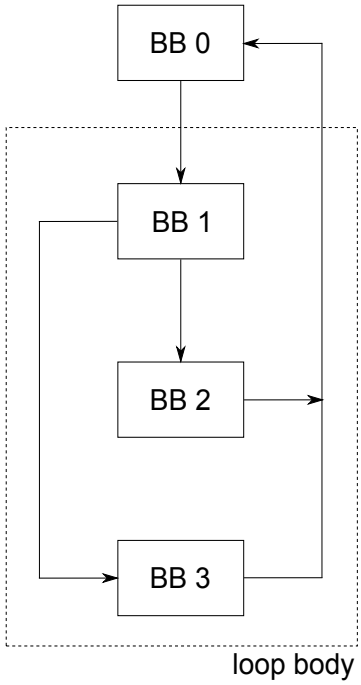

Figure 4.1: Basic block layout of inner loop

As mentioned earlier the autovectorizer cannot vectorize this loop. If this loop could transformed into a loop with just a single basic block as the loop body it could be vectorized (as long as the equivalent vector operations exist for the operations in the loop body, of course). This transformation can be done using *predicated execution*, which is further explained in section 3.5.5, but the AltiVec architecture does not contain these operations[18]. The EVP does, however and we could rewrite the loop to the following code.

```
for (i = 0; i < array_length; i++)
{
  C[i] = A[i];
  if (i & 2) C[i] = C[i] + B[i];
}
```

The line `if (i & 2) C[i] = C[i] + B[i];` denotes a predicated operation. This operation is always executed, but with the added side effect that the result is only stored to the target register if the condition expression evaluates to *true*. Because this line results in a single RTL instruction that is always executed, the loop body becomes a single basic block and can be vectorized. The vector equivalent of predicated execution is *masked execution*, where an instruction is passed a mask register, which contains a 1 in all locations where the result should be written to the target register and a 0 otherwise.

This means the autovectorizer can bring significant improvement when compiling out-of-the-box code, but this requires the addition of support for vector registers and masked execution. Since this

was all not in place at the start of the project and had to be implemented, this was deemed too large a project to complete in the scope of a MSc. thesis project. Therefore we did not continue implementing this and it will be listed as future work (see section 5.3).

Conclusions and Recommendations

5

In this report we presented the implementation of adding vector support to the EVP port for the GCC compiler framework. In this chapter we will give an summary of the issues addressed in this thesis.

In Chapter 1 we gave an overview of the background of the VLIW architectures, vector processors and the GCC compiler framework. We also discussed previous work done is this field. We listed a number of challenges addressed in this report. We also gave an overview of which contributions were part of the MSc. thesis project and which contributions were done by other members of the team working on the EVP-GCC compiler.

Chapter 2 detailed the inner workings of the EVP, GCC and the EVP-C language extension. We presented an overview of all functional units in the EVP, the available registers and the way vector data is organized. In addition we briefly discussed the VLIW aspect of the architecture as well as other features of the processor, predicated execution and hardware loops. We briefly discussed the basics of the EVP-C extensions to the C language and we gave an overview of the inner workings of GCC and discussed some of the more important passes of the compiler.

In Chapter 3 we presented the steps taken to reach all the goals we presented in Chapter 1.We presented how the basic machine properties of the EVP were defined in the back end. This included the implementation of the vector data types, register classes and machine modes. We presented how register classes were matched to the registers of the EVP. We presented how Tree-SSA statements are expanded to RTL instructions. We described the expansions rules and instruction patterns that had to be defined in the EVP back end. We also addressed the expansion of vector instructions and the way intrinsics are handled. This resulted in a compiler that was capable of translating all EVP-C intrinsic operations in the FFT benchmark code to the appropriate RTL instructions. We presented the way predicates and constraints are defined. This includes the new approach taken to implement the constraints, where we replaced the macros used previously by a construct that directly allows us to define the constraints. We presented the implementation of passing arguments to a function and placing a function's return value in the correct register. The generation of assembly code was extended to include verbose comments and the generation of post-increment and -decrement loads and stores was implemented. We also presented extending the GCC's scheduling pass to generate VLIW instructions and we implemented a novel approach to schedule some semantically equivalent instructions on different functional units. This was necessary because using the method provided by GCC resulted in a finite state machine for the scheduler that grew too large. Work done by other members of the team working on EVP-GCC was presented also for completeness purposes. This consists of adding support for if-conversion, the generation of hardware loops and implementing a different method of alias analysis, which is address-based alias analysis.

Chapter 4 presented the experimental results obtained from running the DejaGnu testsuite extended with some new testcases in order to test for correctness of the generated code, and the performance of the scalar EEMBC Telecom benchmark and an FFT benchmark for the vector part in terms of cycle count. The latter two results are not included in this thesis, as they are supplied to the graduation committee seperately, due to the fact that we cannot disclose this information because of contractual

obligations. We also presented the results of a comparison between the EVP-GCC compiler capable of scheduling semantically equivalent operations on diffent functional units and the EVP-GCC compiler without this feature. We measured an improvement of 2-5% in cycle counts for the vector benchmark. We also outlined the experiments done with the AltiVec back end to see the effect of autovectorization of out-of-the-box code of the EEMBC Telecom benchmark suite. We found that the autovectorizer could not bring much improvement without also improving the if-conversion capabilities of the compiler. Improving if-conversion to a level where a large number of inner loops of the benchmark code could be vectorized was estimated to be too large a project for this thesis.

5.1 Impact of the Open Source nature of GCC

In this section we will discuss the aspects of open source or free software and the impact they have had over the course of this project.

Using a compiler licensed under the GNU GPL has several advantages:

- The compiler developer has access to the complete source code, and this brings a number of advantages.
  - It allows the compiler developer to understand the inner workings of the compiler. The fact that the compiler developer has full insight into the inner workings of the compiler allows him to understand exactly what is happening and why certain results are achieved. This aspect has been very important when the new approach of scheduling semantically equivalent operations on different functional units was implemented. The insight in the functions that performed the state transitions for the instruction scheduler allowed us to see the reasons why implementing this using the regular method as described in [28] resulted in a state machine that grew too large. It also allowed us to use some of the functions in the mainline to change the instruction code of instructions as detailed in section 3.3.4. If we would not have had access to the complete GCC source code, implementation of this new scheduling approach would not have been possible.
  - The compiler developer has the ability to change the inner workings of the compiler. The GPL allows a compiler developer not only to read the source code, but also modify it where desired. This aspect was important when the address-based alias analysis was added. This method of alias analysis was not part of GCC, but being able to add this increased the performance of the EVP-GCC compiler.
  - It allows the compiler to be quickly adapted to changes in the architecture. During the course of this project, we worked with a single version of the EVP architecture. Therefore there were no changes to the architecture during the project, so there was no need to adapt the back end to any changes.
  - It allows easy design space exploration. As our main focus was to provide proof that GCC could be used as a compiler framework for the EVP, we did no design space exploration for future versions of the EVP.
- GCC has a large supporting community. The large number of people working on GCC and the quick response time of questions asked to the community through the GCC mailing list allowed us to discuss the problems with the scheduler not only with its author but with anyone

in the community that wants to join in [37]. This has helped with the insights in the instruction scheduling process and discovering its strengths and limitations.

- We can benefit from research work done by others. The autovectorizer is one example of a research project started by IBM at the Haifa Labs that found its way into the GCC mainline. This research and the knowledge gained there will be of use to us when the EVP back end can be adapted to allow for autovectorization in the future.

There is also an aspect of the open source nature of GCC that might cause problems, which is the requirement under the GNU GPL that any free software modified must also be available under the GPL when published. Although over the course of this project we did not encounter any problems, there might be a point were code in the back end will reveal or hint at some of the details of the architecture that ST-Ericsson might not want to reveal.

5.2 Contributions

In Chapter 1 we listed the goals for this project. In this section we will discuss these goals and their solutions. All the issues addressed below are part of our work on this project. Work done by the other members of the team is listed later.

The main goal that we set was listed as:

- The goal of this project is to extend the current version of the EVP-GCC compiler to include support for vector instructions and EVP-C intrinsic operations, and to evaluate the performance relative to the current EVP production compiler.

In order to achieve this we defined a number of subgoals:

- Extend the compiler to support vector data types and vector registers.

  As we presented in Chapter 3 a number of new data types were defined in the back end. The vector registers were added to the macros defining the registers, and three new machine modes were introduced, `V32QImode`, `V16HImode` and `V8SImode`. These modes are mapped to different sets of vector register classes in such a way that if data can freely be moved from one register class to another via a simple move instruction without losing data they have the same mode. This method also allows future addition of extra modes without any significant rewriting.

- Extend the compiler to support the EVP-C extension to the C language by means of implementing support for a number of EVP-C intrinsic operations. The specific instructions that will be implemented will be all EVP-C intrinsics used in the vector benchmark code.

  For all EVP-C intrinsic operations that occur in the FFT benchmark expansion rules and instruction patterns were written, which allows full compilation of the FFT benchmark. This is detailed in section 3.2.1.3.

- The EVP supports post-increment and post-decrement load and store operations. Support for generation of these instructions must be implemented in the compiler.

Using the `auto-inc-dec` optimization pass we implemented support to generate post-increment and post-decrement addressing instructions. This required adding some extra by-passes to correctly define the latency of the instructions for the different registers (i.e. loading the value in the vector register has a latency of 3, while the pointer update has a latency of only 1).

- Some operations can be issued on several different functional units. Support for all alternatives of these instructions must be enabled in the GCC's instruction scheduler.

  Implementing support for this aspect was the major part of this project. The resource usage patterns for all different instructions resulted in a state machine for the scheduler that was too large. We were unable to split the automaton into several smaller ones due to the large number of interdependencies between all resource usage patterns. We presented a new approach to split the state machine without losing the ability to schedule these semantically equivalent instructions on all of the functional units that could perform the operation in the instruction. We showed how we the automaton could be split by building a supporting function that attempted to schedule the alternative that resides in the other automaton and, when successful, altered the information in the scheduler to reflect this scheduling choice. This allowed us to make full use of all available alternatives while at the same time retaining two relatively small automata. We measured a gain of 2-5% on the vector benchmarks when this feature is enabled.

- Enable the autovectorization in GCC to vectorize loops in out-of-the-box benchmark code and measure the performance increase in terms of cycle count compared to non-vectorized code.

  A preliminary research was performed on the EEMBC Telecom code, using the AltiVec back end (the EVP back end did not support vector instructions at the time). This research indicated that enabling autovectorization did not bring much in terms of performance for compiling out-of-the-box EEMBC Telecom benchmarks, at least not until the compiler would be able to perform masked vector execution. Bringing masked vector execution up to a level that would allow the vectorizer to bring significant improvement when compiling out-of-the-box EEMBC Telecom benchmarks required more time than could be handled in the scope of this thesis project, and this remains as future work.

- Measure the performance of the generated code for the chosen benchmark.

  We executed the code from the EEMBC Telecom benchmark and the FFT benchmark and this resulted in the numbers presented in chapter 4. From these numbers we can conclude that the improvements made to the EVP-GCC compiler result in well scheduled code and efficient code.

- The generated code for the benchmarks should be correct.

  We have shown in chapter 4 that the number of failures in the DejaGnu testsuite extended with the extra testcases for the EVP-C intrinsics is at 43. This is approximately equal to the number of failures of the compiler as it was at the beginning of this project. As explained in chapter 4 the remaining failing tests are non-critical for the correct execution of the benchmarks.

5.3 Recommendations for future work

While we have reached our goal of adding support for vector instructions and EVP-C intrinsics to the EVP-GCC compiler there are a number of issues that must be resolved before EVP-GCC is a mature

compiler. These issues, related to feature-completeness of the compiler, are the following:

- Support for all EVP-C intrinsics. So far only the intrinsics used in the FFT benchmark are implemented in the compiler. This was done by hand. To add support for all intrinsics this process should be automated. The machine description files for the intrinsics should be generated from the database containing all EVP-C intrinsics and instructions.

- Adding support for other vector sizes. At the moment, EVP-GCC only supports vectors that are 256 bits wide. In order to support all vector sizes also vector types should be defined for all other vector widths.

- Adding support for circular pointers. The EVP supports an addressing mode where a pointer that is pointing to the last element of an array automatically points to the first element when incremented. Support must also be added to the GCC compiler to recognize such pointers and generate the correct assembly code for it.

- Implementing support for the EVP-C pragmas in GCC. At the moment all EVP-C pragmas are ignored in GCC. Implementing the functionality of some of these pragmas could bring further improvement to the performance of EVP-GCC. One example is the memory space qualifiers `__M1`, `__M2` and so on. Allowing these memory qualifiers to aid the alias analysis could result in less data dependencies, which in turn should improve the performance of the resulting code. For example, at the moment all load operations that load data that resides in a global variable and is passed to a function are still dependent on all context-save/restore instructions. This causes the instruction scheduler to generate code that does not achieve the performance that could be achieved if this information was available. The programmer knows these pointers do not alias but has no effective way of passing this information to the compiler.

- Adding support for floating point data types. Because the EVP only supports fixed point arithmetic at the moment, EVP-GCC does not support floating point operations. Future versions of the EVP may well include floating point support, in which case support should be added to the compiler.

We also have a number of suggestions for future research work. These are listed below:

- Measure the performance on larger applications. The benchmark used to measure the performance of the vector part is relatively small as opposed to applications that commonly run on the EVP. It would be interesting to see how GCC matches up for a large application such as a 3G application. This would provide a better indication of the performance of GCC than the benchmarks used in this project.

- Explore scheduling priorities. At the moment, instructions which use the return address register are given priority in order to maximize the distance between the load instruction and the jump instruction so as many instructions as possible can be moved into delay slots. An exploration could be done to look at the effect on performance for granting different instruction classes a scheduling priority over other classes (e.g. schedule a move from one scalar register to another on the `SLSU` instead of the `SALU` if both options are available).

- Enabling the Integrated Register Allocator. From version 4.4.0 onward, GCC includes a new register allocator next to the one included in previous versions. It would be interesting to measure the performance of the Integrated Register Allocator as opposed to the one currently used.
- Enabling the autovectorizer. GCC includes support for autovectorization of scalar code. If the EVP back end would enable the autovectorizer to vectorize code containing conditional statements the performance could be measured on out-of-the-box EEMBC Telecom benchmark code.
- Adding customizability for architecture exploration. From a design point of view, it is very valuable to be able to change parameters of the architecture. Think of changing the number of registers of certain classes, adding extra move instructions, changing vector widths, adding functional units, changing time shapes or adding support for pre-increment addressing could give interesting data aiding in the design of future generations of the EVP. For this a number of macros that are now hard-coded could easily be rewritten to allow a research team to easily tweak these settings.
- Enable the DFA scheduler to generate a non-deterministic automaton. At the moment the scheduler uses a deterministic scheduling approach. This means that the first alternative (of operations that can be scheduled on multiple different functional units) that can be scheduled is scheduled. By allowing the automaton to be non-deterministic, greater freedom is given to the scheduler. Additional measures have to be taken to later detect which alternative is the one selected by the scheduler. The production compiler currently uses a non-deterministic scheduler. The code to detect the choices made by the scheduler is available in-house and not part of the (licensed) compiler framework. Therefore this could be adapted to work with GCC and improve the performance of the scheduler.

Bibliography

[1] J.P. Grossman, Compiler and Architectural Techniques for Improving the Effectiveness of VLIW Compilation, International Conference on Computer Design, 2000

[2] International Telecommunication Union, International Mobile Telecommunications-2000 Standard, http://www.itu.int/

[3] International Telecommunications Union, International Mobile Telecommunications-Advanced (IMT Advanced), http://www.itu.int/

[4] M.V. Wilkes and J.B. Stringer, "Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer," Proceedings of the Cambridge Philosophical Society, v. 49, 1953, pp. 230-238

[5] A.E. Charlesworth, An Approach to Scientific Array Processing: The Architectural design of the AP-lZOB/FPS- 164 Family. Computer (September 1981), pp. 18-57

[6] J.A. Fisher, Very Long Instruction Word Architectures and the ELI-512, Proceedings of the 10th Symposium on Computer Architectures, June 1983, pp. 140-150

[7] J.A. Fisher, Trace Scheduling: A Technique for Global Microcode Compaction. IEEE Transactions on Computers Vol. C-30, July 1981, pp. 478-490

[8] Intel Corporation, Intel Itanium 2 Processor Hardware Developer's Manual, http://developer.intel.com/design/itanium2/manuals/251109.htm

[9] TMS320C6000 CPU and Instruction Set Reference Guide. Texas Instruments. http://www.ti.com/

[10] J. Parthey, Porting the GCC-Backend to a VLIW-Architecture. Chemnitz University of Technology, March 2004

[11] A. Strätling, Optimizing the GCC Suite for a VLIW Architecture. Chemnitz University of Technology, November 2004

[12] D. Melnik, S. Gaissaryan, A. Monakov, D. Zhurikhin. An Approach for Data Propagation from Tree SSA to RTL, Proceedings of the International Workshop on GCC for Research in Embedded and Parallel Systems (GREPS), Brasov, Romania, 2007

[13] D. Naishlos, Autovectorization in GCC, GCC Summit, June 2004. ftp://gcc.gnu.org/pub/gcc/summit/2004/Autovectorization.pdf

[14] D. Nuzman and R. Henderson, Multi-platform Auto-vectorization, The 4th Annual International Symposium on Code Generation and Optimization, March 2006

[15] D. Nuzman, I. Rosen, A. Zaks, Auto-Vectorization of Interleaved Data for SIMD, ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI), June 2006

[16] D. Nuzman and A. Zaks, Autovectorization in GCC - Two Years Later, GCC Summit, June 2006 http://www.gccsummit.org/2006/2006-GCC-Summit-Proceedings.pdf

[17] Auto-vectorization in GCC, the autovect-branch, http://gcc.gnu.org/projects/tree-ssa/vectorization.html

[18] Power ISA v.2.03, http://www.power.org/resources/downloads/PowerISA_203.Public.pdf

[19] S.K. Gupta and N. Sharma. Alias Analysis for Intermediate Code, Proceedings of the GCC DeveloperŠs Summit, 2003, pp. 71-78

[20] Free Software Foundation, http://www.fsf.com/

[21] GNU General Public License v3, Free Software Foundation, http://www.gnu.org/copyleft/gpl.html

[22] GNU Compiler Collection (v4.3.1), Free Software Foundation, June 2008, http://gcc.gnu.org/

[23] The GNU Operating System, http://www.gnu.org/

[24] GCC Internals Manual (v4.4.0), Free Software Foundation, http://gcc.gnu.org/onlinedocs/

[25] John Hennessy and David Patterson, Computer Architecture, Fourth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[26] EEMBC, http://www.eembc.com/

[27] DejaGnu Framework, http://www.gnu.org/software/dejagnu/

[28] V. Makarov. The Finite State Automaton Based Pipeline Hazard Recognizer and Instruction Scheduler in GCC. Proceedings of the GCC Developer's Summit, 2003.

[29] D. Novillo, Tree SSA - A New Optimization Infrastructure for GCC, 2003 GCC Developer's Summit, pages 181-193, Ottawa, Canada, May 2003.

[30] Sanjiv K. Gupta and Naveen Sharma. Alias Analysis for Intermediate Code. Proceedings of the GCC Developer's Summit, 2003, pages 71-78.

[31] J. Smeets and K. Moerman, Philips Semiconductors Adelante VD32040 Architecture: An Embedded Vector Processor for Low Power DSP Applications. Proceedings of the GSPx, 2005

[32] C.H. van Berkel, F. Heinle, P.P.E. Meuwissen, K. Moerman, and M. Weiss. Vector Processing as an Enabler for Software-Defined Radio in Handheld Devices. EURASIP Journal on Applied Signal Processing 2005:16, 2613.2625

[33] NXP Adelante 32-bit embedded vector processor VD32040, NXP Semiconductors, http://my.semiconductors.philips.com/acrobat/literature/9397/75015342.pdf

[34] Adelante SDK for VD3204x, EVP-C Language Reference Manual Rev. 01.06, July 2008.

[35] Adelante SDK for VD3204x, C-compiler User Manual Rev. 01.03, May 2008.

[36] The ANSI C Standard, American National Standard Institute, 1990.

[37] GCC Mailing List Archives, http://gcc.gnu.org/ml/gcc/