RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Arthur Heimbrecht

Compiler Support for the BrainScaleS Plasticity Processor

KIRCHHOFF-INSTITUT FÜR PHYSIK

Department of Physics and Astronomy
University of Heidelberg

Bachelor Thesis
in Physics
submitted by
Arthur Heimbrecht
born in Speyer

Compiler Support for the BrainScaleS Plasticity Processor

This Bachelor Thesis has been carried out by Arthur Heimbrecht at the

Kirchhoff Institute for Physics

Ruprecht-Karls-Universität Heidelberg

under the supervision of

Prof. Dr. Karlheinz Meier

Compiler Support for the BrainScaleS Plasticity Processor

The BrainScaleS wafer-scale system is an approach to accelerated analog neuromorphic computing that supports online plasticity with fixed update rules. For its next generation the current prototype(HICANN-DLS) features a programmable plasticity processor, that is designed for a wide range of plasticity rules. It is based on a Power7 architecture and includes a vector extension for SIMD-parallelization. Together with a custom I/O-interface it is able to access and update synaptic weights, correlation measurements and chip configuration in the system. This allows for fast and flexible application of plasticity rules during experiments. This architecture is also capable of parforming additional tasks in experiments.

This thesis will focus on bringing high-level programming to the architecture by extending the PowerPC back-end of the GNU Compiler Collection (GCC). This allows for programming the vector extension using important features of GCC, such as vector intrinsics, inline assembly and optimization. First tests of the extended back-end have been successful and already pinned down bugs in the current processor, which could be worked around. The extended back-end is already in use for different synaptic plasticity experiments.

Compiler Support für den BrainScaleS Plastizitäts Prozessor

Das Brainscales Wafersystem ermöglicht beschleunigtes analoges neuromorphes Rechnen mit fest eigebauten synaptischen Plastizitätsregeln. Der Prototyp für die nächste Generation des Systems (HICANN-DLS) verwendet einen programmierbaren Plasitzitätsprozessor, der zukünftig verschiedenste Plastitzitätsregeln ermöglichen soll. Dieser Prozessor basiert auf der Power7 Architektur und wurde um eine spezielle Vektoreinheit zur SIMD-Parallelisierung erweitert. Zusammen mit einer eigenen I/O-Schnittstelle, kann der Prozessor auf Synapsengewichte, Korrelationsmessungen und die Chipkonfiguration zugreifen und updaten. Das ermöglicht schnelle und flexible Platizitätsregeln während eines Experiments. Zusätzlich ist der Prozessor in der Lage weiter Aufgaben in Experimenten zu übernehmen.

In dieser Bachelorarbeit wird die Anpassung der GNU Compiler Collection (GCC) Back-Ends für diese Architektur beschrieben. Dies ermöglicht die vollständige Nutzung höherer Programmiersprachen und weitere Vorteile, wie Optimierung, Unterstützung von integriertem Assembler und intrinsische Vektorfunktionen. Mit dem Backend wurden bereits erste Tests der Vektoreinheit durchgeführt, wodurch Bugs im aktuellen Prozessordesign entdeckt werden konnten, die jedoch mit Software zu lösen sind. Mittlerweile wird das erweiterte Backend von Anwendern für verschiedene Experimente genutzt.

Contents

1 Introduction

Neuromorphic computing has developed into a popular scientific field throughout the last years and finds more and more applications and implementations in science and industry, e.g. [**?**]. These systems already show advantages over traditional computer architectures, like the von-Neumann architecture, in specific applications and continue to improve. In its current generation, neural networks abandon discrete time steps and states, but gain more computational power [**?**]. They use spikes and continuous time scales that resemble nature more closely and allow for efficient implementations as analog hardware which offers high performance at low energy consumption [5]. Still, new architectures also require novel styles of programming [2] and users need to adapt to these. This can be a hurdle for many users when developing new experiments, that initially take a significant amount of time.

One example for this is the current way of programming for the plasticity processing unit of the High Input Count Neural Network - Digital Learning System (HICANN-DLS). The HICANN-DLS is a small scale system that features analog emulation of neurons and synapses in networks [8]. The Plasiticty Processing Unit (PPU), as part of HICANN-DLS, can be used for implementing plasticity rules for such networks. It resembles a traditional processor architecture, which was modified for this task. Implementing such plasticity rules differs from conventional high-level programming styles. When creating code for the PPU, users are partially pushed back to the origins of computing; instead of assigning values like `d = a + b`, one must first read the variables from memory, then operate on their values and finally write back the result to memory. Therefore coding for the PPU works on a low level and brings new challenges to users, that are already challenged by neuromorphic programming.

Ultimately, the more a system abandons conventional elements of programming, the more challenges emerge from this. Although experienced programmers can create highly efficient code like this, normal users will not be familiar with this. This can cause fewer users to take the initiative of writing code for such systems, but also can code get confusing, hard to debug and even inefficient.

Compilers usually save users from these problems by offering high-level languages. Over decades compilers have been developed and became a standard tool for programmers. At the same time compilers became more and more of a black box that transforms a program into an executable file. For this reason it may be difficult for some users to abandon their convenience and go back to low-level programming.

Though the PPU is not completely without compiler support, its distinct features are only usable on a low level. As these features are necessary to implement plasticity rules

on the HICANN-DLS, this can easily cause inconveniences for users. Users repeatedly have to mix high-level and low-level code, which is an atypical style of programming. It can cause different problems, as users have to adapt to this and, in the beginning, likely create bugged or inefficient programs. As performance is important for neuromorphic programming, users may need an unreasonable amount of time and work to achieve simple results with this.

Until now full compiler support does not exist for the PPU because of its modified processor architecture, which was developed solely for neuromorphic hardware. It offers a partly customized instruction set that is optimized for its applications.
The HICANN-DLS already is an experimental platform, which is used by several users, even though of PPU-related challenges. Applications like in-the-loop training or Spike Timing Dependent Plasticity (STDP) have been developed and mostly do not involve the PPU . Even when taking the effort of learning to code for the PPU, users are constantly challenged by missing programming features such as creating parameterized functions. This leads to repetitive code or difficulties when integrating calibration into experiment-related code.

Offering more tools for PPU programs could reduce the effort of developing for the PPU, while at the same time increasing capabilities of programs. Besides allowing full high-level programming, compiler support could also offer tools like code optimization and debugging features. At some point compiler support may also facilitate automatic code generation as a prerequisite for implementation of very high-level languages. Users then could create plasticity rules in existing program environments from where code is translated into PPU programs. This creates the need for optimization of PPU code, like those built into virtually every compiler.

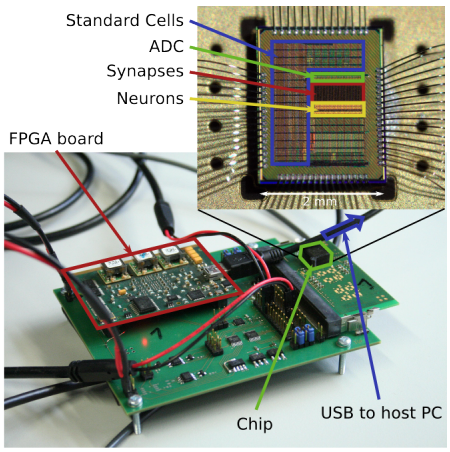

Figure 1.1:
Set-Up of a HICANN-DLS Test System (from *Friedmann et al.*)

This thesis will focus on achieving aforementioned compiler support and briefly explain the process itself. As fundamental knowledge of both, processors and compilers, is needed along the way, the second chapter will start with a very basic introduction to both topics. This involves basic information about the PPU, as well as the GCC, and should explain the basic concepts to an extend which is sufficient. Afterwards, the process of extending the compiler is explained, followed by a presentation of results as well as first test cases. The thesis will conclude in a resume and give an outlook to future applications and development of the compiler and the PPU.

2 Fundamentals and Applications of Computer Architectures and Compiler Design

2.1 Hardware Implementation of Neural Networks

This thesis mainly focuses on a processor that is an essential part of the **High Input Count Neural Network - Digital Learning System (HICANN-DLS)** chip. The following chapter will deal with the HICANN-DLS as a whole and then look into the **Plasiticty Processing Unit (PPU)** in detail while also addressing processor architecture in general.

The HICANN-DLS is a **spike based** system and was built to emulate neural networks at high speeds with low power consumption. This means that neuronal activities do not follow discrete time steps and neurons send out spikes when activated.

Neurons are interconnected through dendrites, synapses and axons where synapses can be of different coupling strength. This means that a neuron is activated only for a short time, called a spike, and sends out this spike through its axon to neurons that are connected via synapses. Between those spikes, the neuron is in a sub-threshold state and not sending any signals, while still receiving input spikes from other neurons. Synapses can work quite differently, but have in common that there is a certain weight associated to them, which will be call **synaptic weight**. The synaptic weight either amplifies or attenuates the pre-synaptic signal. The signal is then passed through the dendrite of the post-synaptic neuron to the soma where all incoming signals are integrated. If the integrated signals reach a certain threshold the neuron spikes and sends this signal to other neurons [**?**].

The HICANN-DLS system implements a simplified neural model in analog electronics, in order to emulate neuronal networks in a biologically plausible parameter range.

At its core HICANN-DLS has a so called **synaptic array** (see figure 2.1) that connects 32 neurons which are located on a single chip to 32 different pre-synaptic inputs). They enclose a 2D field which is the synaptic array as it mainly consists of synapse circuits. All neurons reach into the array through input lines that are organized in columns. The pre-synaptic inputs respectively have wires that resemble rows in the array. At each intersection of those rows and columns a synapse is placed, that thereby connects a neuron and a pre-synaptic input. Overall this gives 1024 synapses, that interconnect neurons with the synaptic input.

A **Field Programmable Gate Array (FPGA)** is connected to all pre-synaptic inputs and routes external spikes to these inputs. Synapses are realized as small repetitive

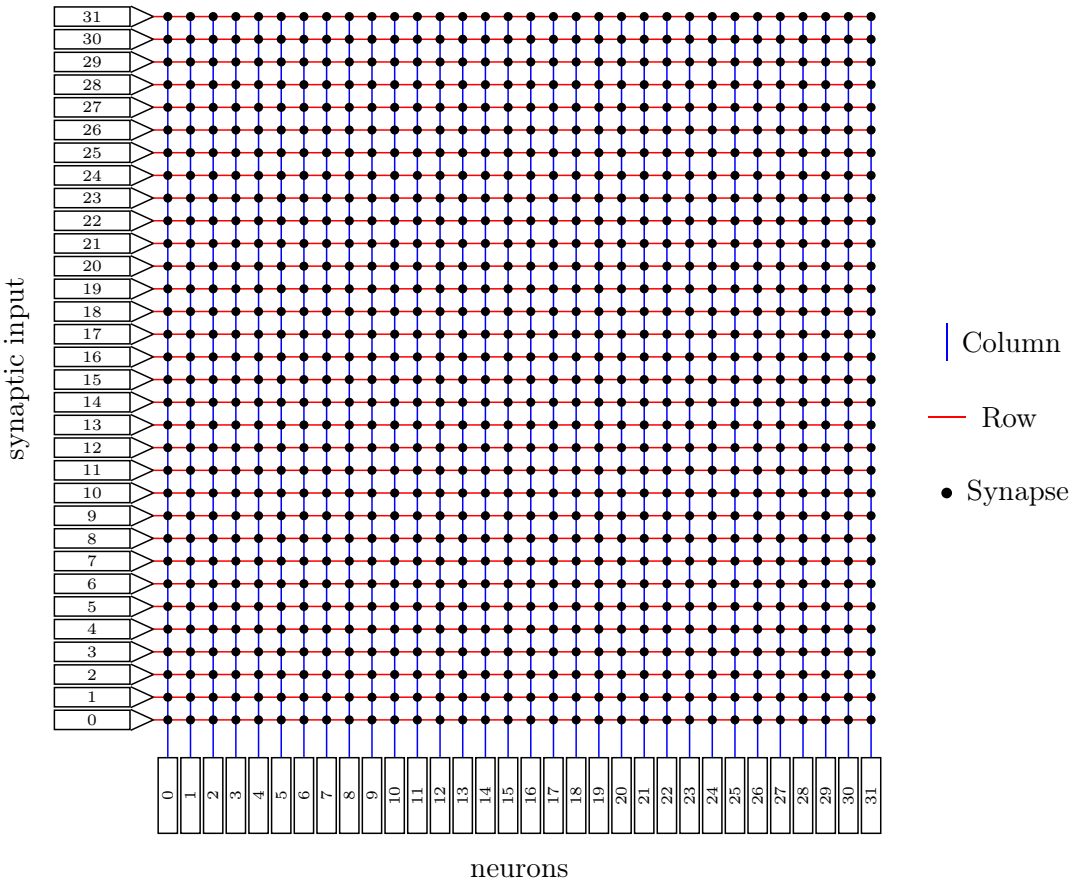

Figure 2.1: The synaptic array consists of pre-synaptic inputs (left), neurons (bottom) and 1024 synapses. All synapses along a column are connected to the respective neuron. Pre-synaptic inputs send their signal to all synapses along their respective row.

circuits that contain 16 bits of data (see figure 2.2). 6 bits of those are used as synaptic weight and the spare two upper bits of that byte are used for calibration. Each synapse also holds a 6-bit wide internal decoder address. Decoder Address and synaptic weight can both be changed from outside.

The synapse array can also be used in 16-bit mode for higher weight accuracy, that combines the weights of two synapses to a 12-bit weight.

The FPGA sends a 6-bit address, whenever it sends a spike to a pre-synaptic input, which then is compared by each synapse to the decoder addresses, they hold themselves. In case the addresses match, each synapse multiplies an output signal with the weight it stores and sends the result along a column where it reaches the neuron.

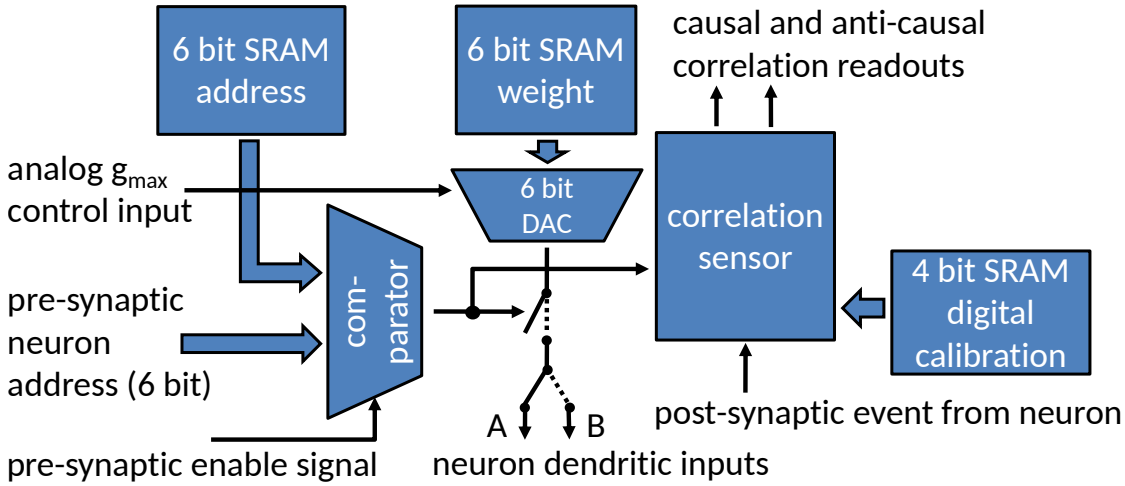

Figure 2.2: Block diagram of a synapse circuit (modified from *Friedmann et al.*).

Along those columns signals from different synapses are collected. Inside the neurons the resulting current input is integrated and if it exceeds a certain threshold, the neuron spikes. If the neuron is spiking, it sends an output signal to the FPGA, which is responsible for spike routing in the first place. All of this is done continuously and does not follow discrete time steps, as mentioned earlier. Along each column sits a **Correlation Analog Digital Converter (CADC)** that measures correlation of post- and pre-synaptic spikes and can be accessed by the PPU, similar to the synaptic array.

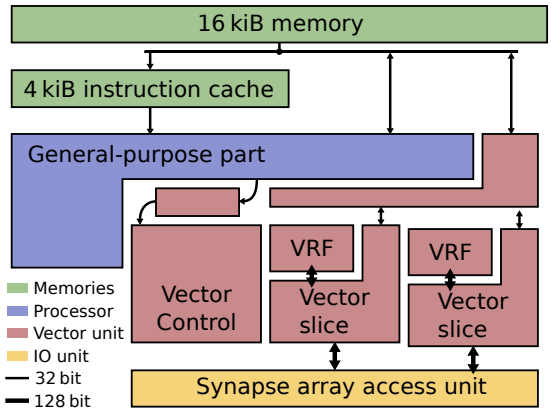

Figure 2.3:
Block diagram of the nux architecture (modified from *Friedmann et al.*).

The PPU is the processing unit of HICANN-DLS and is equipped with a Vector Extension (VE) that is named synaptic plasiticity processor (s2pp). It has also access to the digital information in the synapse array.

The following naming convention will be used throughout this thesis:

PPU is the processor which is part of HICANN-DLS and mainly responsible for plasticity.

nux refers to the architecture of the PPU, see figure 2.3.

s2pp describes the PPU's VE and is part of the nux architecture.

Digital configuration of the synapses and writing PPU programs to the memory is handled by an FPGA, that has access to every interface of a HICANN-DLS chip.

It was developed to handle plasticity and can apply different plasticity rules to synapses during or in between experiments. This is done much faster by the PPU than by the FPGA, which is important for achieving experimental speeds, that are 10^3 times faster than their biological counterparts. In general the PPU is meant to handle plasticity of the synapses during experiments, while the FPGA should be used to initially set up an experiment, manage spike input and record data.

2.2 Processor Architectures and the Plasticity Processing Unit

Although the main goal of HICANN-DLS is to provide an alternative analog architecture, there are advantages to classic computing which are needed for some applications and almost all contemporary processors are built using the so called von-Neumann architecturei [**?**] (figure 2.4).

The main advantage of digital systems over analog systems, such as the human brain, is the ability to do numeric and logical operations at much higher speeds and precision as well as the availability of existing digital interfaces. For this reason "normal" processors are responsible for handling experiment data as well as configuration of an experiment in the HICANN-DLS. This section will explain the basics of such processors and common terms, while referring to the PPU at times when it is convenient.

The PPU, which was designed by *Friedmann et al.*, is a custom processor, that is based on the Power Instruction Set Architecture (PowerISA), which has been developed by IBM since 1990. Specifically the PPU uses POWER7 which was released in 2010 as a successor of the original POWER architecture.

In general, a microprocessor can be seen as a combination of two units which are an operational section and a control section [6, p. 26]. The control section is responsible for fetching instructions and operands, interpreting them, controlling their execution and reading/writing to the main memory or other buses. The operational section, on the other hand, creates results from instructions and operands by performing logic or arithmetic operations on these, as instructed by the control section. Prominent parts of the operational section are the **Arithmetic Logic Unit (ALU)** and the **Register File (RF)**.

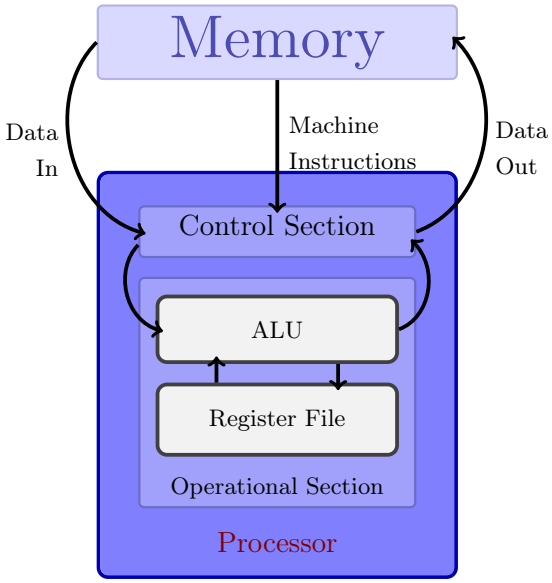

Figure 2.4:
Structure of a Processor in von-Neumann Architecture

The RF can be seen as short-term memory of the processor. It consists of several repeated elements, called **registers**, that save data and share the same size, which is determined by the architecture — the 32-bit architecture of nux for instance has 32-bit wide registers.

Typically the number and purpose of registers varies for different architectures. Common purposes of registers are:

General Purpose Register (GPR) These registers can store values for various causes, but in most cases are soon to be used by the ALU. Most registers on a processor are typically GPRs. Any register that is not a GPR is called a **Special Purpose Register (SPR)**

Linker Register (LR) This register marks the jump point of function calls. After a function completes, the program jumps to the address in the link register.

Conditional Register (CR) This register's value is set by an instruction that compares one or two values in GPRs. Its value can condition some instructions if they are executed or not.

The ALU uses values, which are stored in the RF, to perform the aforementioned logic or arithmetic operations and saves the results there as well.

Some architectures also have an accumulator that is often part of the ALU. Intermediate results can be stored there because access to the accumulator is the fastest possible but it can only holds a single value at a time.

Memory of a von-Neumann machine contains both, the program and data. Usually this memory is displayed as equal-sized blocks of information with addresses as in figure 2.5.

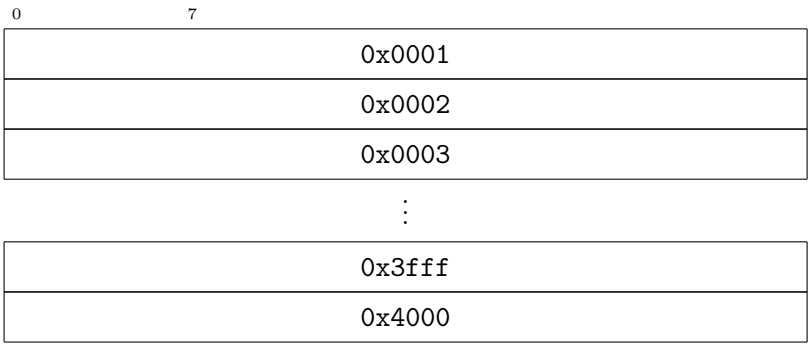

Figure 2.5: Illustration of Word Sizes for 32-bit Words

Each address is equivalent to one byte in memory. The program is normally in a different location in memory than data and the processor goes through the program step by step. Each of these steps is represented by a **machine instruction**, which consists of several elements that occupy a fixed amount of memory (see figure 2.6).

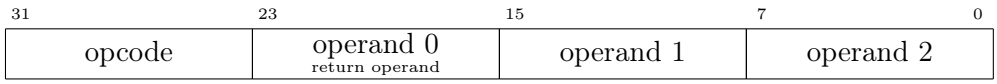

Figure 2.6: Representation of a Machine Instruction in Memory

A machine instruction like in figure 2.6 combines several elements. A program is simply a list of these instructions in memory that belong to the **instruction set**. Each machine instruction requires a fixed amount of memory and consists of an opcode and multiple operands. The **opcode** is the first part of the instruction and is typically an 8-bit number that identifies the operation.

Opcodes are often represented by an alias string like `add`, that is called a **mnemonic**. The opcode is followed by several addresses, that refer to the location of value, or where it should be stored. These addresses are called **operands** and can either be a memory address or a register number. The ALU reads the opcode and operands and performs a set of so called **micro instructions** accordingly [6, p. 23ff.].

During a single clock cycle a chip can perform a single micro instruction. An example for micro instructions in an add instruction (`d = add(a,b)`) would be:

Listing 2.1: Example of Micro Instructions in an `add` instruction

```
1 fetch instruction from memory
2 decode instruction
3 fetch first operand a
4 fetch second operand b
5 perform operation on operands
6 store result
```

The complexity of an instruction set is important for performance. As complex instruction sets feature highly specialized circuits and microinstructions, they are able to complete complex computation in only a few clock cycles. But a complex instruction set often relies on a small set of basic arithmetic instructions and rarely use complex ones. Because every micro instruction must be represented by a circuit in the ALU, a smaller instruction set would safe space and be easier to design.

In general developing a processor architecture involves factors like: available chip space, instruction set and design complexity, energy consumption and maximum clock frequency. Because of this, processors can be classified into two main groups:

Complex Instruction Set Computing (CISC) e.g. x86, MC68000, and i8080

Reduced Instruction Set Computing (RISC) e.g. POWER, ARM and MIPS

The latter usually has an instruction set, that is reduced to simple instructions like `add` or `sub`, and connects these to create more complex instructions. This is similar to how micro instructions work and makes programs on a RISC processor more complicated. This architecture also features more registers than CISC and instruction pipelining, which will be discussed later on.

The PPU is a RISC architecture, therefore this chapter will focus on RISC's key features.

Low-level code, that is written with machine instructions, is called **assembly** code, which is the lowest level of representation of a program that is still is human-readable. Assembly instructions follow the same scheme as machine instructions do:

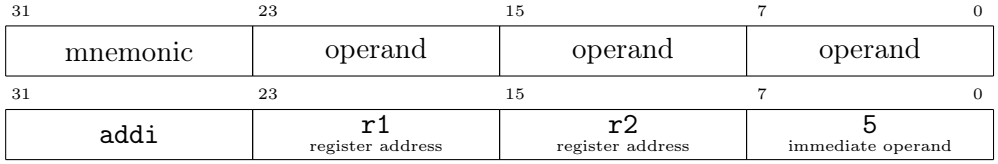

Figure 2.7: Representation of Assembly Instruction `addi` as a Machine Instruction in Memory. The immediate value `5` is added to register `r2` and the result written in `r1`. Table A.1 shows a list of important instructions in the PowerISA.

Listing 2.2: Assembly in Written Form

```
addi      r1, r2, 5
```

In RISC architectures instructions typically consist of 3 operands and are between registers only (except for load/store memory instructions). Instructions, that have less operands, are usually mapped on different instructions. Its operand can be of two different types which are shown in figure 2.7. They either represent a specific register (r1 = register 1) or an immediate value (5 = the integer 5). RISC architectures often support only one **load** (memory to register) and one **store** (register to memory) instruction, which qualifies them as load/store architectures. In order to access memory, operands must be used indirectly:

A memory address is given by an immediate value, that is saved to a register. The registers content is then used by a memory instruction instead of the register address, as shown in listing 2.3.

Listing 2.3: Example Code for Load and Store Instruction. The contents of memory address `0x0000` are loaded into register `r0` and then stored at address `0x1000`. See table A.1 for information on used mnemonics.

```
1 ls        r1, 0x0000
2 lw        r0, 0(r1)
3 li        r1, 0x1000
4 stw       r1, 0(r1)
```

It takes up to several hundred cycles for instructions to access memory, which effectively stalls the processor until the memory instruction has finished.

$$\text{speed(accumulator)} > \text{speed(register)} \gg \text{speed(memory)}$$

Therefore a user should try to avoid memory access as much as possible and use registers instead.

Since instructions on RISC are all very simple, they all follow the scheme in listing 2.1. The processor therefore start **pipelining** instructions, which means starting the next machine instruction as the previous machine instruction just performed the first micro

instruction. Ideally, this will increase the performance by a factor that is equal to the number of micro instructions in a machine instruction.

It must be noted though, that the processor has to implement detection of **hazards**, which are data dependencies between instructions; e.g. one instruction needs the result of another. Such an instruction is then postponed to a delay-slot and other instructions that do not cause hazards are executed instead. The result is reordering of instructions on a processor level [6, p. 54f].

Processors sometimes have so called **co-processors** for complex instructions that are not included in the instruction set, but are still useful. An example would be multiplication on RISC, which would need many cycles, when split into `add` instructions. A co-processor can perform this in just a few cycles. In such a case the control section recognizes the `mult` opcode and passes it to the co-processor instead of the ALU.

This can be extended to whole units similar to the ALU existing in parallel. One example would be a **Floating Point Unit (FPU)**, which is nowadays standard for most processors and handles all instructions on floating point numbers. For this reason has the FPU its own **Floating Point Registers (FPRs)** in a separate register file.

A different kind of extension are **Vector Extensions (VEs)** that do the same as the FPU, but for vectors instead of floats, and allow for **Single Input Multiple Data (SIMD)** processing. This is mostly wanted for highly parallel processes such as graphic rendering or audio and video processing [**?**]. Early supercomputers such as the Cray-1 also made use of vector processing, to gain performance by operating on multiple values simultaneously through a single register [**?**]. This could either be realized through a fully parallel architecture or more easily through pipelining instructions for vector elements. The latter one is possible since there are typically no dependencies, hence no hazards, between single elements in the same vector. Nowadays basically all common architectures support vector processing. A few examples are:

- x86 with SSE-series and AVX
- IA-32 with MMX
- AMD K6-2 with 3DNow!
- PowerPC with AltiVec and SPE
- ARM with NEON

The s2pp VE on the nux architecture is the PPU's distinct feature that allows for SIMD operations on synaptic weights. The VE is weakly coupled to the General Purpose Processor (GPP) of the PPU. Both parts can operate in parallel while interaction is highly limited. To handle the vector unit, the instruction set was extended by 53 new vector instructions. The **Vector Register File (VRF)** contains 32 new vector registers which are each 128-bit wide [**?**]. This allows for either use of vectors with 8 half-word (see figure 2.8) sized elements or 16 byte sized elements, which are 128 bits long as seen in figure 2.9.

This section takes a special interest in the AltiVec vector extension itself which was developed by Apple, IBM and Motorola in the mid 1990's and is also known as Vector Media Extension and Velocity Engine for the POWER architecture. The AltiVec exten-

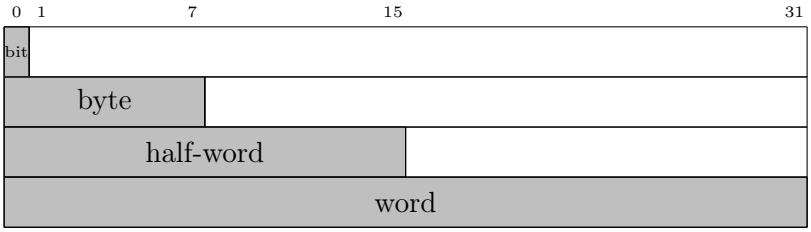

Figure 2.8: Illustration of Word Sizes for 32-bit Words

sion provides a similar single-precision floating point and integer SIMD instruction set. Its vector registers can hold sixteen 8-bit `char` (V16QI), eight 16-bit `short` (V8HI), four 32-bit `int` (V4SI) or single precision `float` (V4SF) — each signed and unsigned [**?**].

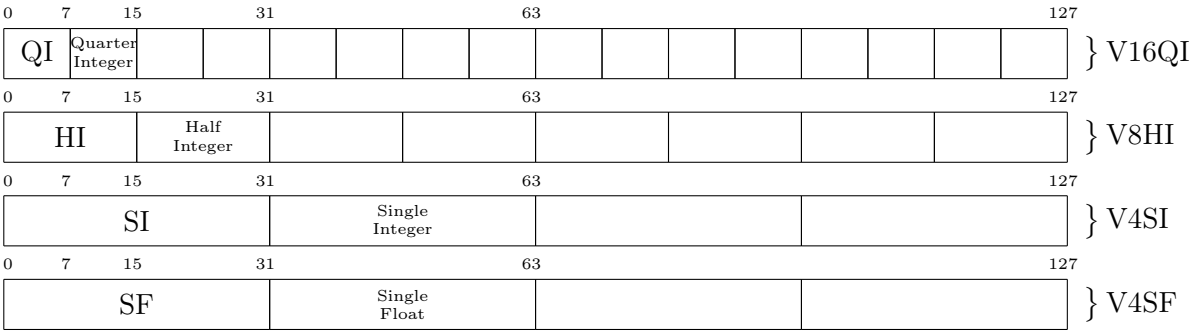

Figure 2.9: Vector structures are 128 bits wide and split into common word sizes.

It resembles most characteristics of the s2pp vector extension, like a similar VRF, and is already implemented in the PowerPC back-end of GNU Compiler Collection (GCC), but both VEs also feature differences.

The s2pp VE features a double precision vector accumulator and a Vector Conditional Register (VCR) which holds 3 bits for each half byte of the vector, making 96 bit in total. The bits represent the result of a previous comparison instruction for vector elements. If the first bit is set, the compared element was larger than `0`, if it was less than `0` the second bit is set and if the element is equal to `0`, the third bit is set. For more information see the nux manual [7, p. 23].

Instructions on the s2pp VE can be specified to operate only on those elements of a vector, that meet the condition in the corresponding bits in the VCR, while the AltiVec VE utilizes the CR of the PowerPC architecture. If element-wise selection is needed, AltiVec offers this through vector masks.

The AltiVec VE has two registers that are not featured on s2pp. The Vector Status and Control Register (VSCR) is responsible for detecting saturation in vector operations and decides which floating point mode is used. The Vector Save/Restore register (VRSAVE) assists applications and operation systems by indicating for each Vector Register (VR) if it is currently used by a process and thus must be restored in case of an interrupt [**?**].

Both of these registers are not available in the s2pp VE but would likely not be needed for simple arithmetic tasks which the PPU is meant to perform.

It was already stated that all instructions of VEs must first pass the control unit, which detects vector instructions and then passes them to the VE. These instructions then go into an instruction cache for vector instructions. On nux the instructions then shortly stay in a reservation station that is specific for each kind of operation and thus allows for little out-of-order operation of instructions in these reservation stations, which is illustrated in figure 2.10. This allows for performing some arithmetic operations on a vector during the process of accessing a different vector in memory. This results in a faster processing speed, as pipelining for each instruction is also supported. Though the limiting factor for this remains the VRF's single port for reading and writing. An even more limiting factor is the shared memory interface of the s2pp and GPP.

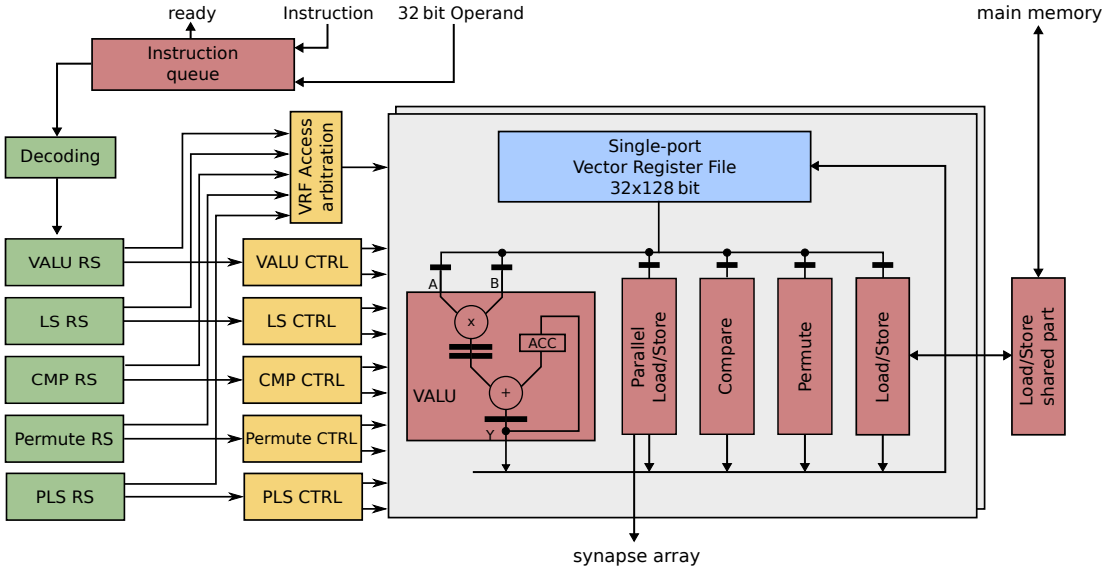

Figure 2.10: Detailed Structure of the s2pp Vector Extension (taken from *Friedmann et al.*)

Normally processors themselves do not keep track of memory directly. This is done by a **Memory Management Unit (MMU)** or a Memory Controller (MC). It handles memory access by the processor and can provide a set of virtual memory addresses which it translates into physical addresses. Most modern MMUs also incorporate a cache that stores memory instructions while another memory instruction is performed. It detects dependencies within this cache and resolves them. Ultimately, this results in faster transfer of data because the MMU can return results from the cache, without accessing the memory [6, p. 435ff.]. Not all MMUs support this though, which might lead to certain problems when handling memory. If instructions are reordered due to pipelining while dependencies on the same memory address are not detected correctly, an instruction may write to memory before a different one could load the previous value, it needed from

there. Another reason could be delayed instructions, which were mentioned earlier. For this reason memory barriers exist.

A **memory barrier** is an instruction that is equal to a guard, placed in code, that waits until all load and store instructions issued to the MMU are finished. It therefore splits the code into instructions issued before the memory barrier and issued after the memory barrier. This prohibits any instruction from being executed on the wrong side of the barrier due to reordering and thereby generally prevents conflicting memory instructions.

One kind of memory barrier is called `sync` which is used in listing 2.4. This and other memory barriers are also described in table A.1.

Listing 2.4: The memory barrier ensures that the first store has been performed before the second store is issued.

```
1 stw     r7,data1(r9)    #store shared data (last)
2 sync                    #export barrier
3 stw     r4,lock(r3)     #release lock
```

Using `sync` can result in up to a few hundred cycles of waiting for memory access to finish and therefore should only be done if necessary.

The PPU's memory is 16 kiB which is accompanied by 4 kiB of instruction cache. The MMU of this system is very simple as it does not cache memory instructions and also has matching virtual and physical addresses, thus memory barriers can become necessary at times.

Another feature of the PPU is the ability to read out spike counts and similar information through a bus which is accessible through the memory interface of the MMU. It uses the upper 16 bits of a memory address for routing These are available because the memory is only 16 kiB large, which is equivalent to 16-bit addresses. A pointer to a virtual memory address allows to read for example spike counts during an experiment. This is possible for the whole chip configuration, such as analog neuron parameters. One of the main feature of the PPU is the access to the synaptic array through an extra bus, which can be seen in figure 2.10.

The memory bus is also accessible by the FPGA. This is needed for writing programs into the memory as well as getting results during, or after experiments. It also allows for communication between a "master" FPGA and a "slave" PPU.

A **bus** is the connection between parts of a processor and used for data transfer.

When using vector instructions for nux, one must always keep in mind that the weights in the synaptic array only consist of the latter 6 or 12 bits which are in a vector register and are right aligned.

Figure 2.11 displays the representation of values in the synapses and in vectors. The weight is the sum of the values where the bits are set to one. A user must shift the vector's elements when reading/writing to the synapse array, as only then do special attributes of instructions work properly.

An example would be instructions that rely on **saturation** which predefines a minimum and maximum value. In case, the result is out-of-range, the instruction will return either the minimum or the maximum (whichever is closer). For this to work properly the

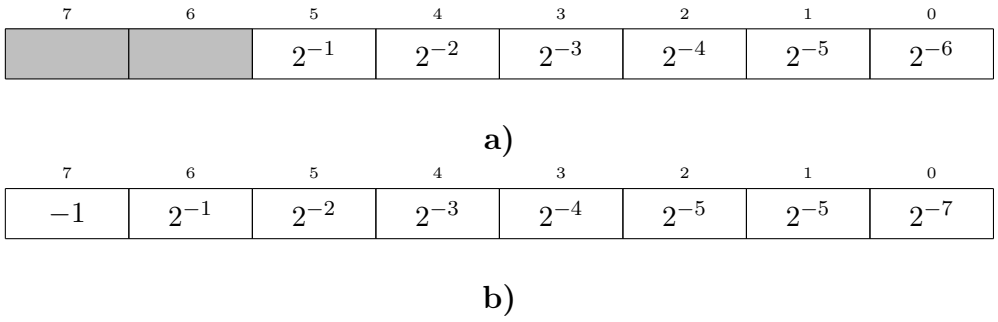

Figure 2.11: Comparison of 2.11a) Representation of Weights in Synapses and 2.11b) Fractional Representation of Vector Components for Fixed-Point Saturational Arithmetic

bit representation must match the intended one, which is the fractional representation, and the values must also be correctly aligned.

An overview of all vector opcodes is provided in the nux manual [7, ch. 5], which is recommended as accompanying literature to this thesis. In general these vector opcodes are divided into groups of instructions:

modulo halfword/byte instructions apply a modulo operation after every instruction which causes wrap around in case of an overflow at the most significant bit position. Each instruction is provided as halfword (modulo 2^{16}) and as byte instruction (modulo 2^8).

saturation fractional halfword/byte instructions allow for the results only to be in the range $[-1, 1 - 2^{-7}]$ for byte elements and $[-1, 1 - 2^{-15}]$ for half-word elements.

permute instructions perform operations on vectors that handle elements of vectors as a series of bits.

load/store instructions move vectors between vector registers and memory or the synapse array.

2.3 Basic Compiler Structure

At its core every compiler translates a source-language into a target-language as figure 2.12 illustrates [1, p. 3]. Most often it translates a high-level, human readable programming language into a machine languages.

What differs compilers from interpreters is the separation of **compile-time** and **run-time**. Interpreters combine these two and translate a program at run-time. A compiler reads the source-language file completely (often several times) and then creates the executable files, which are executed after the process has finished. This has certain advantages to it: While a compiler takes some time at first until the program can be started, the resulting executable is almost always faster and more efficient. This is due

to the possibility of optimizing code during the compilation process and the chance of reading through the source file several times if this is needed (with each time the code is read being called a **pass**). Of course there do exist many different compilers today and what matters to the user is the combination of the amount of time it takes to compile a program and the performance of that program.

A compiler is not the only contributor to translation of a program into an executable program, although it is the most prominent one. Figure 2.13a) illustrates the chain of tools that is involved into this process: First the **preprocessor** modifies the source code, before it is processed by the compiler. It removes comments, substitutes macros and also includes other files into the source before it passes the new program code to the compiler. The compiler then passes its output to the **assembler**. It translates the output of the compiler which is written in **assembly** into actual machine code by substituting the easy-to-read string alternatives with actual opcodes. The **linker** combines the resulting "object-files" that the assembler emitted with standard library functions, that are already compiled, and other resources. The only task which is left for the **loader**, is assigning a base address to the relative memory references of the "relocatable" file. The code is now fully written in machine language and ready for operation [1].

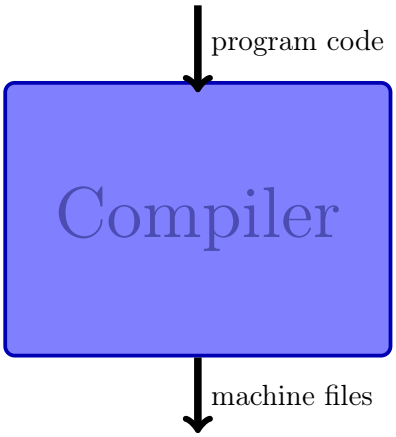

Figure 2.12:
Compiler Representation

Figure 2.13b) shows the separation of a compiler into **front-end**, **back-end** and an optional **middle-end**. This structure makes a compiler portable, which means allowing the compiler to accept different source-languages, which are implemented in the front-end, and produce different target-languages, which must be specified in the back-end. Therefore if one wants to compile two different programs e.g. one in C, the other in FORTRAN, it is necessary to change the front-end, but not the back-end, because the machine or **target** stays the same. The middle-end in this regard is not always needed, but could be responsible for optimizations, that are both source-independent and target-independent.

Of course, the different parts of the compiler have to communicate through a language that different parts can understand or speak. Such a language is called **Intermediate Representation (IR)** and also used during different phases of the compilation process. It may differ in its form but always stays a representation of the original program code [3, p. 8].

The different phases of a compilation process are illustrated in figure 2.13b. First the preprocessed source code is given to the **scanner** that performs lexical analysis, which is combining sequences of characters, like variables, and attributes, such as "number" or "plus-sign", to so called tokens. Next, the **parser** takes the sequence of tokens and builds a syntax tree, that represents the structure of a program and is extended by the **semantic analyzer**, which adds known attributes at compile-time like "integer" or "array of integers" and checks if the resulting combinations of attributes are valid. This already

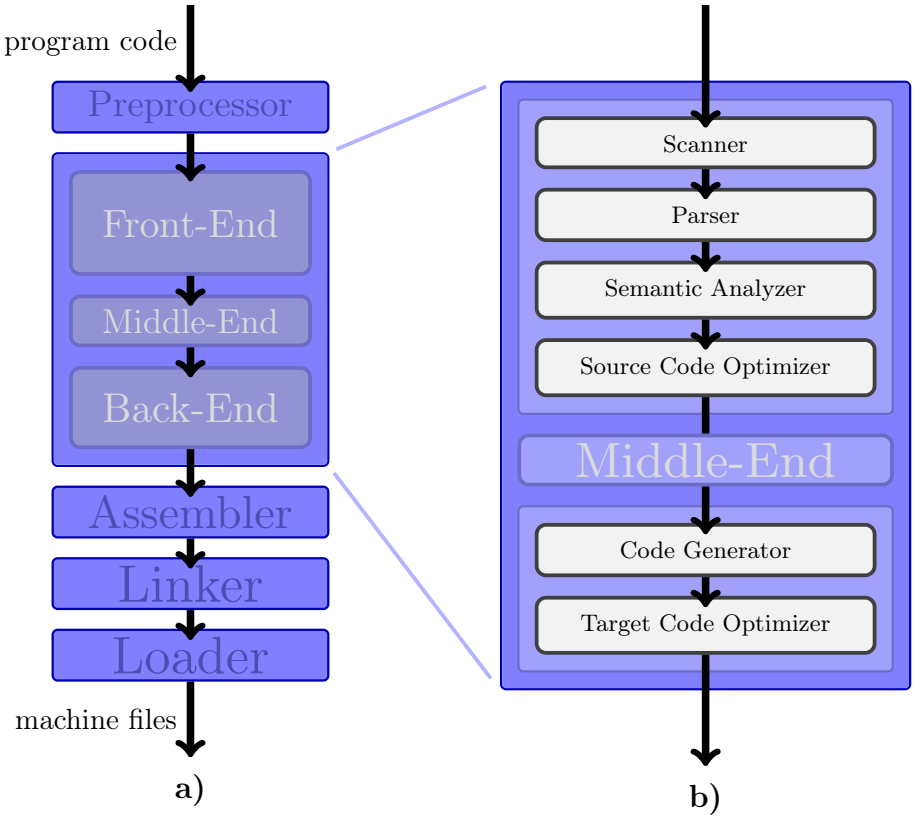

Figure 2.13: Overview of Compilation Stages
2.13a) shows stages of the compilation process. 2.13b) different phases in the compiler.

is the first form of IR. The **source code optimizer** which is the last phase of the front-end takes the syntax tree and tries to optimize the code. Typically only light optimization is possible at this point, such as pre-computing simple arithmetic instructions. After the source code optimizer is done, the syntax tree is converted into a different IR in order to be passed to the back-end.

The **code generator** takes this IR and translates it to machine code that fits the target — typically this is assembly. At last the **target code optimizer** tries to apply target-specific optimization, until the target code can be emitted [3, 1].

2.3.1 Back-End and Code Generation

The last two phases of the compiler, which are part of the back-end, are the most interesting with respect to this thesis. Usually, the processes of code generation and target optimization in the back-end are entangled, as optimization can take place at different phases of code generation. This section will take a look at code generation in the back-end.

The source program reaches the back-end in form of IR. Often the IR is already linearized and thereby again in a form, that can be seen as sequence of instructions. Because of this, the IR may also be referred to as Intermediate Code. The process of generating actual machine code from this is again split into different phases:

- instruction selection
- instruction scheduling
- register allocation

At first the back-end recognizes sets of instructions in intermediate code that can be expressed as an equivalent machine instruction. Depending on the complexity of the instruction set, a single machine instruction can combine several IR instructions. This may involve additional information, that the front-end aggregated and added to the IR as attributes. At the end of this, a compiler typically emits a sequence of assembly instructions, which will be explained later. In order to fulfill this task, the compiler needs the specifications of the target it compiles for. This is called a target description and can contain things like specifications of the register-set, restrictions and alignment in memory and availability of extensions and functionalities. The compiler also needs knowledge of the instruction set of a target, which is determined by the Instruction Set Architecture (ISA) and is a list of instructions, which are available. It also needs to know what functions certain instructions have. The compiler picks instructions according to their functionality from this list and substitutes the IR with this. Ideally a compiler should support different back-ends just by exchanging the machine description and the ISA as the basic methods of generating code are the same for most targets.

After the IR is converted into machine instructions the back-end now rearranges the sequence of instruction. This needs to be done, as different instructions take different amounts of time to be executed. If a subsequent instruction depends on the result of a previous instruction the compiler has two alternative approaches to solve this. First it can stall the programs execution as long as the instruction is executed and feed the next instruction into the processor when the dependency is solved. This means that the compiler adds `nop` before an instruction that needs to wait for an operand. For critical memory usage the compiler can also insert `sync` as memory barriers before hazardous memory instructions. Alternatively, it can stall only the instruction which depends on the result which is currently computed, but perform instructions that do not depend on the result in the mean time. By doing so, the scheduler increases performance noticeably and thus can be seen as part of the optimization process. On RISC architectures this is especially important as load and store instructions take noticeably longer than normal register instructions and pipelining depends mainly on the instruction sequence. Thus the scheduler is also involved in parallelization of code. As a result of this, a compiler would usually accumulate all load instructions at the beginning of a procedure and start computing on registers that already have a value, while the others are still loaded. This is done vice versa at the end of a procedure for storing the results in memory. This process needs the compiler to know the amount of time it takes for an instruction to be

executed. Usually the workload of an instruction is described as **cost**. All of this works hand in hand with hazard detection on processor level.

At last the compiler handles **register allocation**, which also includes memory handling. Typically, the previous processes expects an ideal target machine, which provides an endless amount of registers. As in reality, the processor only has k registers. The register allocator reduces the number of "virtual registers" or "pseudoregisters", that are requested, to the available number of "hard registers" k. For this to be possible the compiler decides whether a value can live throughout a procedure in a register, or must be placed in memory if there are not enough registers available. This results in the allocator adding load and store instructions to the machine code, in order to temporarily save those registers in memory, which is called **spilling**. This can hurt performance and therefore the compiler tries to keep spilling of registers to a minimum and inserts spill code at places where it delays other instructions as little as possible. At the end of register allocation, the compiler assigns hard registers to the virtual registers which are now only k at a time [3, 1].

During and after code generation the compiler also applies **optimizations** to the machine code. Any optimization to the code must take three things into consideration, which are safety, profitability and risk/effort. First of all, safety of optimizations should always be given. Only if the compiler can guarantee that an optimization does not change the result of the transformed code, it may use this optimization. If this applies, the compiler may check for the profit of an optimization, which most often is a gain in performance, but could also be reducing the size of the program. At last the effort or time, it takes for the compiler to perform this optimization, and the risk of generating actually bad or inefficient code must be taken into account as well. If an optimization passes these three aspects, it may be applied to the code. In the end there exist some simple optimizations, that always pass this test like the deletion of unnecessary actions or unreachable code, e.g. functions that are never called. Another example is reordering of code, like the scheduler did before, or the elimination of redundant code, which applies if the same value is computed at different points and thus the first result simply can be saved in a register. If a compiler knows the specific costs of instructions, it can also try to substitute general instructions with more specialized but faster instructions, like substituting a multiplication with 2 by shifting a value one position to the left. There exist many more ways of optimization but one more major type shall be discussed.

In **peephole optimization** the compiler looks at small amounts of code at a time through a "peephole" and tries to find a substitution for the specific sequence of instructions it "sees". If the sequence can be substituted, the peephole optimizer does so, otherwise the peephole is moved one step and a new sequence is evaluated. These substitutions must be specified by hand and are highly target-dependent in contrast to the optimizations which were mentioned before, that are target-independent [3].

2.3.2 Inline Assembly

Some compilers, like GCC, offer the possibility, to include low-level code into high-level programs. This is called **inline assembly** and uses the function `asm`.

Listing 2.5: Exemplatory Assembly Invokation

```
1 int dst, src1, src2;
2 asm volatile (  "add %0, %1, %2"
3                 : "=r" (dst)
4                 : "r" (src1), "r" (src2)
5                 : /*no clobbered regs*/);
6 return dst; /*would return src1 + src2*/
```

Listing 2.5 generates the instruction `add` in the assembly output of the compiler, which is followed by three operands.

First, one must write an assembler template, that is based on assembly. The integer in `%n` indicates the order, in which the operands are specified after the assembler template.

Output operands are specified after the first `:` as a list of comma separated constraints and variables. `"=r"` is such a **constraint**, that determines that the operand must be stored in a register (`r` for register operand) and that the register is written (`=` is called a modifier). The variable in parentheses must be declared before this and must be of matching type (`float` would not be allowed in this case).

The second `:` separates the input operands, which are specified the same way. `r` again represents a register operand and the variable is in parentheses, the different operands are separated by commas. The third `:` separates operands from clobbered (=temporarily used) registers which would also be in quotes, but in this case no registers are clobbered, which are not also operands. `volatile` means that the compiler must not delete the following instructions due to optimization.

2.3.3 Intrinsics

Intrinsics are sometimes also called **built-in functions** and resemble an intermediate form of inline assembly and a high-level programming language. By calling an intrinsic function, the compiler is ordered, to use a certain machine instruction that typically shares its name with the intrinsic. What differs an intrinsic from `asm()` is, that there is no need, to specify constraints, as only argument types must match. One could easily mistake them for normal library functions but they are directly integrated into the back-end of a compiler and thus independent of the programming language. In order to implement intrinsics into a back-end, the compiler needs certain knowledge of what the `asm` instruction does and what kind of operands it needs.

A typical application for intrinsics would be vectorization and parallelization of code through processor extensions. Sometimes this is the sole option of using the machine instructions associated with them.

2.3.4 GNU Compiler Collection

GCC is a compiler suite that supports different programming languages and targets. A single build of GCC can support a variety of front-ends while it was built for a specific target. This target, in most cases, is the processor architecture on which the user runs

the compiler. But GCC also supports the idea of a **cross-compiler**, which is the concept of compiling code on one machine but running the code on another machine that may be based on a different architecture.

One build of GCC does not support different back-ends though and therefore GCC must be built individually for every back-end, it wants to compile code for. This is realized through a modular structure which follows the idea of a front-end, middle-end and back-end which was described in section 2.3. Some information that belongs to a back-end is also needed at the front-end, hence the compiler is built back-end specific but supports a wide variety of back-ends to choose from.

GCC itself is programmed in C++ and is part of the GNU project of the Free Software Foundation. It is wide-spread and one of the most popular compilers especially among academic institutions and small scale developers. Every major UNIX distribution and many minor ones include GCC as a standard compiler [**?**].

There is one major competitor to GCC as an open source compiler suite. This is **Low Level Virtual Machine (LLVM)** together with Clang. Both support running the same source code on different architectures, while LLVM actually runs intermediate code rather than actual machine code and uses GCC to generate this intermediate code for some front-ends. There are ongoing discussions on which compiler is better suited for which application but regarding performance, GCC takes the slight edge [**? ?**]. These results have to be viewed with care though, as they are based on different processor architectures and both compilers provide similar performance.

In this thesis GCC was chosen over LLVM for two main reasons. One is that GCC follows the concept of a traditional compiler that generates machine code. The other is that GCC support for the PPU existed to a minimum with a working cross-compiler when starting with this thesis.

During this thesis, GCC was at stable release version 6.3 and development is ongoing for version 7, but this thesis uses the older version 4.9.2, which has been used internally by the working group. Additionally binutils 2.25 will be used, which was patched by Simon Friedmann and since includes the opcodes and mnemonics for nux.

Because it is the base architecture of nux, the PowerPC back-end of GCC, which is called **RISC system/6000 (rs/6000)** and is equivalent to POWER, will be emphasized throughout the rest of this thesis. According to GCCs Internals manual [11], which will be referred to as the sole source of information in this regard, the back-end of GCC has the following structure:

Each architecture has a directory with its respective name `gcc/config/target/` (i.e. `gcc/config/rs6000/`), that contains a minimum amount of files. These are the **machine description** `target.md`, which is an overview of machine instructions with additional information to each instruction, the **header files** `target.h` and `target-protos.h`, which define mostly macros, and a **source file** `target.c` that implements functions for the target. Every back-end is built from such files. Most back-ends include additional files which simplify a back-ends complex structure.

One of the most prominent functions in a GCC back-end is **reload**. It is specifically meant to do register spilling but is an active part of the whole register allocation phase [9]. Through multiple releases of GCC it became more and more complex and incorporated

Listing 2.6: Definition of a General `add` Insn

```
1 (define_insn "add<mode>3"
2   [(set (match_operand:VI2 0 "register_operand" "=v")
3         (plus:VI2 (match_operand:VI2 1 "register_operand" "v")
4                   (match_operand:VI2 2 "register_operand" "v")))]
5   "<VI_unit>"
6   "vaddu<VI_char>m %0,%1,%2"
7   [(set_attr "type" "vecsimple")])
```

Listing 2.7: Definition of `add` Insn for `float`

```
1 (define_insn "*altivec_addv4sf3"
2   [(set (match_operand:V4SF 0 "register_operand" "=v")
3         (plus:V4SF (match_operand:V4SF 1 "register_operand" "v")
4                    (match_operand:V4SF 2 "register_operand" "v")))]
5   "VECTOR_UNIT_ALTIVEC_P (V4SFmode)"
6   "vaddfp %0,%1,%2"
7   [(set_attr "type" "vecfloat")])
```

more functionalities. This involves for example moving the contents of different registers and memory around. It thus became a main source for errors when constructing a back-end and was replaced in newer releases by the Local Register Allocator (LRA). GCC 4.9.2 is not impacted by this and still features `reload`.

Insn Definition and Register Transfer Language

Register Transfer Language (RTL), which is not to be mixed up with Register Transfer Level, is a form of IR, the back-end uses to generate machine code. Usually GCC uses the IR GIMPLE which resembles stripped down C code with 3 argument expressions, temporary variables and `goto` control structures. The back-end transforms this into a less readable IR, that inherits GIMPLEs structure, but brings it to a machine instruction level. It is inspired by Lisp and for this thesis mainly used as template when defining insns.

An **insn** (short for instruction) has several properties like a name, an RTL template, a condition template, an output template and attributes [11, ch. 16.2]. It is used for combining RTL IR with actual machine instructions.

`define_insn` defines an RTL equivalent to a machine instruction as an insn. The name of the insn in listing 2.6 is `add<mode>3` (`3` for three operands), where `<mode>` is to be replaced by a set of values that describe modes [11, ch. 16.9].

A mode is the form of an operand, e.g. `si` for single integer, `qi` for quarter integer (quarter the bits of a single integer), `sf` for single float or `v16qi` for a vector of 16 elements which are quarter integers each [11, ch. 13.6]. There are many more modes that follow the same scheme. In this case the mode is not defined explicitly but uses an iterator that creates a `define_insn` for every valid mode that is specified [11, ch. 16.23]. The insn in

listing 2.7 shows this with a specific mode.

Next follows the **RTL template**, which is in square brackets. All RTL templates need a **side effect** expression as a base, which describes what happens to the operands that follow. In this case `set` means that the value which is specified by the second expression, is stored into the place specified by the first expression [11, ch. 13.15]. The next expression that follows is a specified operand. `match_operand` tells the compiler that this is a new operand. `VI2` belongs to the mode iterator and is to be substituted by the equivalent mode to `<mode>` but in capitals, which can be seen in listing 2.7. After this comes the index of an operand which starts at 0 for every `define_insn`. The following string describes a **predicate**, which tells the compiler more about the operand and which constraints it must fulfill. Operands typically end in `_operand` and a single predicate is meant to group several different operand types. In this case any register would be a valid operand [11, ch. 16.7]. The next string specifies the operands further and is meant to fine tune the predicate. It is called a **constraint** and matches the description in section 2.3.2 (again, `=` means that the register must be writable and `v` stands for an AltiVec vector register) [11, ch. 16.8]. This pattern is repeated for every operand and only changes slightly. The second expression of the `set` side effect has an additional pair of parentheses because of the `plus` statement. This is an **arithmetic expression** and tells the compiler that the following operands are part of an operation that results in a new value. It is also followed by a mode that specifies the result [11, ch. 13.9].

The RTL template is matched by the compiler against the RTL, which is generated from GIMPLE and if the template matches, the RTL is substituted by the output template.

After the RTL template is finished, the **condition** specifies if the insn may be used. It is a C expression and must render `true`, in order to allow the matching RTL pattern to be applied. In this case the condition is also depending on the mode iterator which substitutes `<VI_unit>` for equivalent code to that of 2.7 [11, ch. 16.2], with a matching mode.

The **output template** usually is similar to the assembler template in inline assembly. The string contains the mnemonic of a machine instruction and the operands which are numbered according to the indexes of the RTL template. Again this is depending on the mode iterator and `<VI_char>` will be substituted by a character that belongs to a machine mode [11, ch. 16.2].

At last the insn is completed by its **attributes**, which hold further information about the insn. Attributes are used by the compiler internally to detect effects of an insn on certain registers and similar properties [11, ch. 16.19].

3 Extending the GCC Back-End

The previous chapter dealt with processors, the PPU, compilers and GCC, which was preparation for this chapter. This chapter will now emphasize on the task of extending the GCC back-end. There is a number of files, which will be systematically edited and referenced as they are important parts of the rs/6000 back-end and are changed in the process of extending the back-end.

rs6000.md is the machine description of the back-end in general and contains insn definitions for all scalar functions

rs6000.h is a header file which contains macros and declarations for registers

rs6000.c is the source file which implements the back-end's functions

rs6000.opt lists the options and flags for the target

rs6000-builtins.def contains the definitions of intrinsics

rs6000-cpus.def lists sub-targets that belong to the rs/6000 family

rs6000-c.c links built-ins and overloaded built-ins

rs6000-opts.h contains a set of enumerations that represent option values for the back-end

rs6000-protos.h makes functions in `rs6000.c` globally available

rs6000-tables.opt lists values to a processor enumeration

driver-rs6000.c a collection of driver details for different targets

ppc-asm.h sets macros for the use of `asm`

s2pp.md is a new machine description of s2pp and contains insn definitions

s2pp.h is the header file that defines aliases for built-ins

constraints.md contains definitions of constraints

predicates.md contains definitions of predicates

vector.md defines general vector insns

sysv4.h initializes a variety of option flags and sets default values

t-fprules sets soft-float as default for certain targets

It is recommended to have chapter 5 of the **nux manual** [7, ch. 5] at hand, as it contains an overview of existing s2pp vector instructions.

Before extending the GCC back-end a few things must be stated:

Due to the limited documentation of the back-end itself, one must rely on comments in code and the **GCC internals manual** [11] As a full implementation for a vector extension already exists, the AltiVec extension should be used as a guideline for a new extensions [**?**]. Still, it should be avoided to change exiting code as much as possible. Code is often referred to from different places in the back-end and modifying existing code can therefore easily lead to compiler errors. Especially since the back-end is not extended completely right away but rather step by step. This applies particularly to functions that are implemented for AltiVec only. It is recommended to rather duplicate functions and distinguish them, before they are called. This will make it easier to find bugs, as usually the function that generates an error is indicated in the error message. Also, there do exist enough differences between these two vector extensions, that combining functions would not save work.

It will therefore occasionally be pointed out when functions or other code can be inherited from AltiVec and which modifications are needed.

3.1 Adding the s2pp Option Flag and nux Target

Extending the rs6000 back-end starts by adding the `nux` processor to the list of targets and also including mandatory flags with this. Ideally the user only has to add the flag `-mcpu=nux` when compiling, in order to produce machine code for the nux. The flags which have to be set when using the nux are:

`-msdata=none` disables the use of a **small data section** which is like a data section but has a register constantly referring to it and thus has faster access than the normal data section. Globals, statics and small variables that are often used are preferably stored there. It is turned off because the base pointer is not initialized by the linker and the effect of a small data section would likely be small for the PPU [7**?** , 4].

`-mstrict-align` aligns all variables in memory which means that a variable always starts at a memory address which is a multiple of its size. E.g. a vector has always an address that is dividable by 16 bytes or 128 bits. Memory management is far easier for aligned variables.

`-msoft-float` tells the compiler that there is no FPU and all floating point operations have to simulated by software.

`-mno-relocatable` states that the program code has a fixed memory address that may not be altered. Relocatable code is not needed as the PPU runs only one program that is loaded into memory and uses no environment on top.

But first there should be an **option flag** that activates nux' VE like `-maltivec` does for the AltiVec VE. The name for this new option flag will be `-ms2pp` and it will define an option mask along with it. In `rs6000.opt` and we simply need to add:

```
ms2pp
Target Report Mask(S2PP) Var(rs6000_isa_flags)
Use s2pp instructions
```

This adds `ms2pp` to the list of option flags and the next lines defines a macro, that is associated with it. `Target` means that the option is target specific, therefore only certain architectures support the option flag. `Report` means that the option is printed when `-fverbose-asm` is is set. `Mask(S2PP)` initializes a bitmask, that is available through `OPTION_MASK_S2PP`. That macro is attached to `rs6000_isa_flags`, which is specified by `Var`. It simultaneously specifies a macro `TARGET_S2PP` that is set to `1` [11, ch. 8].

This needs also specification of

```
#define MASK_S2PP OPTION_MASK_S2PP
```

in `rs6000.h` as macros with `MASK_` are a standard from earlier versions of GCC.

Although this option flag shall later enable s2pp support, it needs the aforementioned flags as well, to compile nux programs. For this reason exists a processor type which combines those flags. There exist several lists that contain available targets and nux shall be included. First an inline assembly (see section 2.3.2) flag is created, which tells the assembler which system architecture is used. As nux is based on POWER7, one can copy the flag `-mpower7` in `driver-rs6000.def`:

Listing 3.1: `rs6000.h`

```
    #define ASM_CPU_SPEC \
        ...
        %{mcpu=power7: %(asm_cpu_power7)} \
        ...
        %{mcpu=nux: %(asm_cpu_power7)} \
        ...
```

Listing 3.2: `driver-rs6000.c`

```
    static const struct asm_name asm_names[] = {
        ...
        { "power7",   "%(asm_cpu_power7)" },
        ...
        { "nux",  "%(asm_cpu_power7)" },
        ...
```

This will set the assembler `-mpower7` when using `-mcpu=nux`.

The nux target should also be recognized by preceding phases of the compiler and set option flags accordingly. These options flags can be set in `rs6000-cpus.def`.

```
...
RS6000_CPU ("nux", PROCESSOR_POWER7, MASK_SOFT_FLOAT | MASK_S2PP |
    MASK_STRICT_ALIGN | !MASK_RELOCATABLE)
```

This uses the macro `RS6000_CPU (NAME, CPU, FLAGS)` and adds `nux` to the `processor_target_table[]`. Since option flags usually set masks, one can set the respective masks directly. The masks will tell the compiler that the processor is a POWER7 architecture and uses `soft-float`, `strict-align` and `no-relocatable` (negated relocatable) as well as the new s2pp mask.

It is not possible, to set the `-msdata=none` flag before since the `-msdata` flag is initialized differently. Also since it is not simply set "on" or "off" but accepts several values, it is handled in `sysv4.h`. `rs6000_sdata` will be set according to the string that follows `-msdata=`.

```
1  #define SUBTARGET_OVERRIDE_OPTIONS                          \
2  ...
3    if (rs6000_sdata_name)                                    \
4       {                                                      \
5         if (!strcmp (rs6000_sdata_name, "none"))            \
6       rs6000_sdata = SDATA_NONE;                            \
7       ...
8         else                                                 \
9       error ("bad value for -msdata=%s", rs6000_sdata_name); \
10      }                                                      \
11   else if (OPTION_MASK_S2PP                                 \
12            && OPTION_MASK_SOFT_FLOAT                        \
13            && OPTION_MASK_STRICT_ALIGN                      \
14            && !OPTION_MASK_RELOCATABLE)                     \
15      {                                                      \
16      rs6000_sdata = SDATA_NONE;                            \
17      rs6000_sdata_name = "none";                           \
18      }                                                      \
19   else if (DEFAULT_ABI == ABI_V4)                           \
20 ...
21 )\todo{looose this!!!!!!!!!!!!!!!!!!!!!}
```

It is not possible to detect in this file, if the nux flag is set. It therefore needs a little workaround that helps setting the value of `rs6000_sdata`. If `-msdata` is not set, i.e. only `-mcpu=nux` is set, the compiler will use `if`-clauses that determine which value is assigned to `rs6000_sdata`. It is possible, to add a case that checks for all flags, that are set by `-mcpu=nux` and set `rs6000_sdata` to `SDATA_NONE` if this applies. Hence the target options will set `rs6000_sdata` to `SDATA_NONE`.

There exists a case for which this condition applies even when `nux` is not set as target, but all flags are set by hand. If one chooses an explicit value for `-msdata`, this case does not apply though and the value of `-msdata` is set accordingly.

This is not an ideal solution, but a trade-off with as few side effects as possible.

Already this would allow for the use of `-mcpu=nux` as target and `-ms2pp` as option flag. But since the flags we used are basically mandatory to the s2pp extension, the compiler should check for these flags before starting compilation. First though for each flag needs a macro, which the back-end can identify. This is done in `rs6000-c.c` where **global macros** can be defined:

```
1  rs6000_target_modify_macros (bool define_p, HOST_WIDE_INT flags,
```

```
2                   HOST_WIDE_INT bu_mask)
3 {...
4   if ((flags & OPTION_MASK_S2PP) != 0)
5    rs6000_define_or_undefine_macro (define_p, "__S2PP__");
6   if ((flags & OPTION_MASK_STRICT_ALIGN) != 0)
7    rs6000_define_or_undefine_macro (define_p, "_STRICT_ALIGN");
8   if ((flags & OPTION_MASK_RELOCATABLE) != 0)
9    rs6000_define_or_undefine_macro (define_p, "_RELOCATABLE");
10  if (rs6000_sdata != SDATA_NONE)
11   rs6000_define_or_undefine_macro (define_p, "_SDATA");
12   ...}
```

If `flags` and the respective option masks are set, `rs6000_define_or_undefine_macro` will define a macro that is specified by the second argument. Whether a macro is defined or undefined depends on the boolean `define_p`, which is set by the compiler.

These new macros can be used to check if flags are set. This needs a new file, that will also be needed later on as the **s2pp header file**. `s2pp,h` must be indexed in `gcc/config.gcc` under `extra_headers`.

```
1 ...
2 powerpc*-*-*)
3     cpu_type=rs6000
4     extra_headers="ppc-asm.h altivec.h spe.h ppu_intrinsics.h paired.h
          spu2vmx.h vec_types.h si2vmx.h htmintrin.h htmxlintrin.h s2pp.h"
5     need_64bit_hwint=yes
6     case x$with_cpu in
7     xpowerpc64|xdefault64|x6[23]0|x970|xG5|xpower[345678]|xpower6x|xrs64a
          |xcell|xa2|xe500mc64|xe5500|Xe6500)
8     cpu_is_64bit=yes
9     ;;
10    esac
11    extra_options="${extra_options} g.opt fused-madd.opt rs6000/rs6000-
          tables.opt"
12    ;;
13 ...
```

This is done, so GCC invokes the header file, as it is not referenced elsewhere.

`s2pp.h` can now be used to check the compiler flags.

```
1 /* _S2PP_H */
2 #ifndef _S2PP_H
3 #define _S2PP_H 1
4
5 #if !defined(__S2PP__)
6 #error Use the "-ms2pp" flag to enable s2pp support
7 #endif
8 #if !defined(_SOFT_FLOAT)
9 #error Use the "-msoft-float" flag to enable s2pp support
10 #endif
11 #if !defined(_STRICT_ALIGN)
12 #error Use the "-mstrict-align" flag to enable s2pp support
13 #endif
14 #if defined(_RELOCATABLE)
15 #error Use the "-mno-relocatable" flag to enable s2pp support
```

```
16 #endif
17 #if defined(_SDATA)
18 #error Use the "-msdata=none" flag to enable s2pp support
19 #endif
20 ...
```

If for example `__S2PP__` is not defined but `s2pp.h` included, the compiler will emit an error that tells the user to set the target flag. Since hard floats are not supported on `nux` regardless of `s2pp.h`, nux can be added to the list of soft-float processors in `t-fprules`.

```
1 SOFT_FLOAT_CPUS = e300c2 401 403 405 440 464 476 ec603e 801 821 823 860
    nux
```

3.2 Creating Macros

Since the preliminary requirements are now met, the back-end needs a `vector` **attribute** for specifying vectors in program code. Attributes are used to specify various variables and can be used for example to control alignment [11, ch. 16.19].

First a new vector unit is needed. It will be called `VECTOR_S2PP` and added to the enumeration `rs6000_vector` in `rs6000-opts.h`.

```
1  enum rs6000_vector {
2        VECTOR_NONE,          /* Type is not  a vector or not supported */
3        VECTOR_ALTIVEC,       /* Use altivec for vector processing */
4        VECTOR_VSX,           /* Use VSX for vector processing */
5        VECTOR_P8_VECTOR,     /* Use ISA 2.07 VSX for vector processing */
6        VECTOR_PAIRED,        /* Use paired floating point for vectors */
7        VECTOR_SPE,           /* Use SPE for vector processing */
8        VECTOR_S2PP,          /* Use s2pp for vector processing */ //s2pp-
             mark
9        VECTOR_OTHER          /* Some other vector unit */
10 };
```

To put this to use, it needs macros in `rs6000.h` which compare vector units to the newly created `VECTOR_S2PP`.

```
1 ...
2 #define VECTOR_UNIT_S2PP_P(MODE)                \
3     (rs6000_vector_unit[(MODE)] == VECTOR_S2PP)
4 ...
5 #define VECTOR_MEM_S2PP_P(MODE)                 \
6   (rs6000_vector_mem[(MODE)] == VECTOR_S2PP)
7 ...
```

`VECTOR_UNIT_S2PP_P(MODE)` and `VECTOR_MEM_S2PP_P(MODE)` are identical as identical entries in `rs6000_vector_unit[]` and `rs6000_vector_mem[]` are created. This is a relict from the AltiVec implementation as vector units in memory may differ in certain cases.

Checking for specific **vector modes**, which are supported by s2pp, will also be added. The nux hardware only supports two types of vectors which are vectors with byte elements (V16QI) and vectors with half-word elements (V8HI).

```
#define S2PP_VECTOR_MODE(MODE)        \
          ((MODE) == V16QImode)       \
           ||  (MODE) == V8HImode)
```

Some uses of `TARGET_ALTIVEC` must now be accompanied by `TARGET_S2PP` to handle vectors correctly. There exist five such cases:

`rs6000_builtin_vectorization_cost`, `rs6000_special_adjust_field_align_p` and `expand_block_clear` handle alignment of vectors. **Alignment** refers to the position of data blocks in memory; 16-bit alignment means that variables may only start at addresses that represent multiples of 16 bits. AltiVec vectors and s2pp are aligned the same way and it is desirable to reduce misalignment of 128-bit vectors to a minimum.

`rs6000_common_init_builtins` initializes common built-ins and is needed by all extensions that use built-ins (see section 3.5). In these cases the condition can be extended for `TARGET_S2PP`.

Other conditions that will later be extended for `TARGET_S2PP` need further modification and thus are not mentioned here.

It is necessary, to do the same for `VECTOR_UNIT_S2PP_P` and other macros that have AltiVec counterparts: In `reg_offset_addressing_ok_p` cases for `V16QImode` and `V8HImode` return false if `VECTOR_MEM_S2PP_P` or the AltiVec version apply. In `rs6000_legitimize_reload_address` and `rs6000_legitimate_address_p` offset addresses are handled the same way they are handled for AltiVec. In `rs6000_secondary_reload` indirect addressing is enforced. In `print_operand` operand modifier `y` is validated for s2pp. `rs6000_vector_mode_supported_p` returns true if a mode is supported by s2pp.

All of these cases handle addressing of vectors in memory which is equivalent for AltiVec and s2pp. It is therefore quite simple to support this for s2pp.

Since vector modes and units have been established by now, it is possible to connect these in `rs6000_init_hard_regno_mode_ok`. In case `TARGET_S2PP` is set `VECTOR_S2PP` is assigned to modes `V16QImode` and `V8HImode`.

```
...
  if (TARGET_S2PP)
      {
      rs6000_vector_unit[V8HImode] = VECTOR_S2PP;
      rs6000_vector_mem[V8HImode] = VECTOR_S2PP;
      rs6000_vector_align[V8HImode] = align32;
      rs6000_vector_unit[V16QImode] = VECTOR_S2PP;
      rs6000_vector_mem[V16QImode] = VECTOR_S2PP;
      rs6000_vector_align[V16QImode] = align32;
      }
...
```

Preferred modes when vectorizing a non-vector mode in `rs6000_preferred_simd_mode` can be set.

```
...
  if (TARGET_S2PP)
    switch (mode)
```

```
4         {
5         case HImode:
6       return V8HImode;
7         case QImode:
8       return V16QImode;
9         default:;
10        }
11 ...
```

It is now possible to create vector attributes, as mentioned before. GCC already supports a vector attribute which is also used by AltiVec. Thus s2pp can be added to `rs6000_attribute_table` and `rs6000_opt_masks[]` array with the same values as for AltiVec but changing the keyword.

```
1  static const struct attribute_spec rs6000_attribute_table[] =
2  {
3      /* { name, min_len, max_len, decl_req, type_req, fn_type_req, handler
           ,
4            affects_type_identity } */
5      { "altivec",   1, 1, false, true,  false,
         rs6000_handle_altivec_attribute,
6        false },
7      { "s2pp",   1, 1, false, true,  false, rs6000_handle_s2pp_attribute,
8        false },
9    ...}
10 struct rs6000_opt_mask {
11   const char *name;      /* option name */
12   HOST_WIDE_INT mask;        /* mask to set */
13   bool invert;           /* invert sense of mask */
14   bool valid_target;         /* option is a target option */
15 };
16
17 static struct rs6000_opt_mask const rs6000_opt_masks[] =
18 {
19   { "altivec",          OPTION_MASK_ALTIVEC,        false, true  },
20   ...
21   { "s2pp",          OPTION_MASK_S2PP,       false, true  },
22   ...}
```

The function `rs6000_handle_s2pp_attribute` is also copied from AltiVec, but stripped off unsupported vector modes.

This would make these attributes already usable but defining built-ins in `rs6000-c.c` shortens the attribute from `__vector=__attribute__((s2pp(vector__)))` to `__vector` :

```
1  void
2  rs6000_cpu_cpp_builtins (cpp_reader *pfile)
3  {
4    ...
5    if (TARGET_S2PP){
6       builtin_define ("__vector=__attribute__((s2pp(vector__)))");
7       if (!flag_iso){
8          builtin_define ("vector=vector");
9          init_vector_keywords ();
```

```
10          /* Enable context-sensitive macros.  */
11          cpp_get_callbacks (pfile)->macro_to_expand =
                rs6000_macro_to_expand;
12        }
13    }
14  ...
```

`__vector` is then used to define `vector` in `s2pp.h`.

```
1  #define vector __vector
```

At last it must be indicated to the front-end, that special attributes are handled by the back-end.

```
1  static bool
2  rs6000_attribute_takes_identifier_p (const_tree attr_id)
3  {
4    if (TARGET_S2PP)
5      return is_attribute_p ("s2pp", attr_id);
6    else
7      return is_attribute_p ("altivec", attr_id);
8  }
```

3.3 Registers

This section will describe, how s2pp registers are added to the back-end. It will also add constraints and predicates (see section 2.3.4) for these registers.

There are three types of registers in the s2pp VE:

32 vector registers these are normal vector registers that hold vector values

1 accumulator which is used for chaining arithmetic instructions and cannot be accessed directly

1 conditional register which holds conditional bits and also cannot be accessed directly

During extension of the GCC back-end, it becomes apparent that a reserved vector register, that is all zeros the entire time, will be necessary for some implementations of the back-end. This is necessary since the nux instruction set does not include logical vector instructions. Normally the instructions `XOR` and `OR` are used by the back-end to implement simple register features. `OR` is used for moving around the content of a register, as `OR`ing the same first register to a second register will simply copy the contents of the first register. On the other side, does `XOR`ing the same register result in writing all zeros to the return register.

Since these instructions are not available, "nulling" a register becomes a problem. Therefore the first register is reserved and splatted with zeros. Moving this register, will have the same effect as `XOR`ing a register. As `OR` is also not available, an alternative instruction is used, which is `fxvselect`. `fxvselect` selects either elements of the second or the third operand depending on the condition register and its forth operand [7, ch. 5].

Having identical second and third operands thus will simply generate the same vector as return value. By setting the forth operand `0`, `fxvselect` will always choose elements from the second operand. This gives a simple work-around, as `fxvselect` also takes only one clock cycle for execution. An alternative idea would be subtracting the same register from itself with `fxvsubm`, which also nulls the return operand. This would take more clock cycles though and is unfavorable, as it is not clear, how often registers need to be nulled. Ultimately it is a trade-off between having one less register at hand and possibly wasting clock cycles continuously. In this case it is preferable to give up a single register, as the amount of nulling instructions is unknown. At a later point in time, this could be reviewed for performance, which might overturn this decision.

It is not possible to splat zeros constantly because this would require an extra instruction to load a zero into a GPR. This is not possible at late stages of code generation as all registers are already allocated at that time.

When talking about reserved registers, one must also think about saved, call-used and fixed registers:

fixed registers serve only one purpose and are not available for allocation at all.

call-used registers are used for returning results of functions. They are not available to general register allocation but are used when calling functions.

saved registers are available globally and may hold values throughout function calls.

Usually about half of all registers are declared call-used and the other half saved. This is done for AltiVec, as well as FPRs, but might be optimized in the future, depending on requirements of applications (e.g. are many function calls used).

Register indexes are declared in `rs6000.md`:

```
1  (define_constants
2    [(FIRST_GPR_REGNO      0)
3  ...
4     (LAST_GPR_REGNO       31)
5     (FIRST_FPR_REGNO      32)
6     (LAST_FPR_REGNO       63)
7     (FIRST_S2PP_REGNO     33)
8     (LAST_S2PP_REGNO      63)
9     (S2PP_COND_REGNO      32)
10    (S2PP_ACC_REGNO       64)
11    (LR_REGNO             65)
12 ...])
```

Each register index is a unique identifier of registers and is given in incrementing order. Registers which may be available on the same processor must not share and index! s2pp reuses the reserved vector register's index 32 (this register is always null) for the conditional register and uses the free index 64 for the accumulator. As the GPRs need the first 32 registers numbers (0-31) and there is never an FPU on nux, it is possible, to use the 32 registers normally reserved to FPRs.

It then is decided, which registers shall be used for function calls, and therefore reserved for call-use. This is declared by macros that are assigned a register number in `rs6000.md`.

```
/* Minimum and maximum s2pp registers used to hold arguments.  */
#define S2PP_ARG_MIN_REG (FIRST_S2PP_REGNO + 2)
#define S2PP_ARG_MAX_REG (S2PP_ARG_MIN_REG + 12)
#define S2PP_ARG_NUM_REG (S2PP_ARG_MAX_REG - S2PP_ARG_MIN_REG + 1)
...
#define S2PP_ARG_RETURN S2PP_ARG_MIN_REG
...
#define S2PP_ARG_MAX_RETURN (DEFAULT_ABI != ABI_ELFv2 ? S2PP_ARG_RETURN \
                    : (S2PP_ARG_RETURN + AGGR_ARG_NUM_REG - 1))
...

#define FUNCTION_VALUE_REGNO_P(N)                    \
  ((N) == GP_ARG_RETURN                             \
     || ((N) >= FP_ARG_RETURN && (N) <= FP_ARG_MAX_RETURN         \
         && TARGET_HARD_FLOAT && TARGET_FPRS)              \
     || ((N) >= ALTIVEC_ARG_RETURN && (N) <= ALTIVEC_ARG_MAX_RETURN   \
         && TARGET_ALTIVEC && TARGET_ALTIVEC_ABI)             \
     || ((N) >= S2PP_ARG_RETURN && (N) <= S2PP_ARG_MAX_RETURN     \
         && TARGET_S2PP)              \
     )
...
#define FUNCTION_ARG_REGNO_P(N)                      \
  ((unsigned) (N) - GP_ARG_MIN_REG < GP_ARG_NUM_REG         \
     || ((unsigned) (N) - ALTIVEC_ARG_MIN_REG < ALTIVEC_ARG_NUM_REG   \
         && TARGET_ALTIVEC && TARGET_ALTIVEC_ABI)             \
     || ((unsigned) (N) - FP_ARG_MIN_REG < FP_ARG_NUM_REG         \
         && TARGET_HARD_FLOAT && TARGET_FPRS)              \
     || ((unsigned) (N) - S2PP_ARG_MIN_REG < S2PP_ARG_NUM_REG     \
         && TARGET_S2PP)              \
     )
...
```

The only use of these macros is in the prologue and the epilogue, which will be discussed in the next section.

After all register indexes are declared, they can be specified further. Each register type (or **register class**) needs an entry to the enumeration `reg_class` and a definition of identical register names in `REG_CLASS_NAMES`.

```
enum reg_class
{
  ...
  GENERAL_REGS,
  S2PP_C_REG,
  S2PP_REGS,
  FLOAT_REGS,
  S2PP_ACC_REG,
  ALTIVEC_REGS,
  ...
  ALL_REGS}
...
```

```
#define REG_CLASS_NAMES  \
  {                      \
  ...
  "GENERAL_REGS",        \
  "S2PP_C_REG",          \
  "S2PP_REGS",           \
  "FLOAT_REGS",          \
  "S2PP_ACC_REG",        \
  "ALTIVEC_REGS",        \
  ...
  "ALL_REGS"}
```

Then the relation between register classes in specified in `REG_CLASS_CONTENTS`.

```
/* GENERAL_REGS.  */                              \
{ 0xffffffff, 0x00000000, 0x00000008, 0x00020000, 0x00000000 },    \
/* S2PP_C_REG.  */                                                 \
{ 0x00000000, 0x00000001, 0x00000000, 0x00000000, 0x00000000 },    \
/* S2PP_REGS.  */                                                  \
{ 0x00000000, 0xfffffffe, 0x00000000, 0x00000000, 0x00000000 },    \
/* FLOAT_REGS.  */                                                 \
{ 0x00000000, 0xffffffff, 0x00000000, 0x00000000, 0x00000000 },    \
/* S2PP_ACC_REG.  */                                               \
{ 0x00000000, 0x00000000, 0x00000001, 0x00000000, 0x00000000 },    \
/* ALTIVEC_REGS.  */
{ 0x00000000, 0x00000000, 0xffffe000, 0x00001fff, 0x00000000 },    \
...
/* ALL_REGS.  */                                \
{ 0xffffffff, 0xffffffff, 0xffffffff, 0xffe7ffff, 0xffffffff }}
```

Each hexnumber in these arrays can be viewed as a bit mask, with the least significant bit representing the first register, the next higher order bit the second register and so on. As a number is 32-bit wide, it masks 32 registers. Subsequent numbers start where the previous one ended, therefore are registers 32 through 63 (32 is the 33rd register) masked by the second number [11, ch.l 17.8].

Therefore does `0xfffffffe` mask all registers except for the 32nd which is masked by `0x00000001`.

One can see that FPRs are masked completely as `FLOAT_REGS` between definitions of s2pp registers. Subsequent entries must not be subsets of previous masks but may extend these. Also should masks for higher register indexes follow masks for lower indexes. Since a register index which was not masked before, was also added, some subsequent masks like `ALL_REGS` need to be updated accordingly.

There exist macros for register classes as well, which need to be implemented. This is only necessary for general s2pp registers, as other s2pp registers can not be accessed directly.

```
...
#define S2PP_REG_CLASS_P(CLASS)          \
    ((CLASS) == S2PP_REGS)
...
```

As all registers are specified, they can be assigned short names, that are used in assembly. Normally these are the same as the constraints that refer to these registers and an additional integer.

The constraints are:

`kv` for `S2PP_REGS`, the vector registers

`kc` for `S2PP_C_REG`, the conditional register

`ka` for `S2PP_ACC_REG`, the accumulator

`k` was chosen as the first character of s2pp constraints because there are very few letters left which were not used as constraints already and `k` can be somewhat associated with the nux ("nuks"). The second character is the respective first letter of a register type.

Register names are defined in `rs6000.h`.

```
#define ADDITIONAL_REGISTER_NAMES \
{
  ...
  {"kc", 32}, {"kv0",  33}, {"kv1",  34}, {"kv2",  35},   \
  ...
  {"kv27", 60}, {"kv28", 61}, {"kv29", 62}, {"kv30", 63},   \
  {"ka", 64},                \
}
```

The strings are names for registers and the integers represent their indexes.

After these names have been defined, one can also define the according constraints in `constraints.md`

```
(define_register_constraint "kv" "rs6000_constraints[RS6000_CONSTRAINT_kv
    ]"
  "s2pp vector register")

(define_register_constraint "kc" "rs6000_constraints[RS6000_CONSTRAINT_kc
    ]"
  "s2pp conditional register")

(define_register_constraint "ka" "rs6000_constraints[RS6000_CONSTRAINT_ka
    ]"
  "s2pp accumulator")
```

The first string is the register constraint's name and the second string will be assigned a register class later in `rs6000.c`. The last string is only for documentary purposes [11, ch. 16.8].

Before register classes can be assigned, an enumeration in `rs6000.h` must be modified.

```
enum r6000_reg_class_enum {
  ...
  RS6000_CONSTRAINT_v,      /* Altivec registers */
  RS6000_CONSTRAINT_kv,     /* s2pp vector regsiters*/
  RS6000_CONSTRAINT_kc,     /* s2pp conditional register*/
  RS6000_CONSTRAINT_ka,     /* s2pp accumulator*/
  ...
};
```

The last step towards completing the register implementation is assigning register classes and register types to indexes in `rs6000_init_hard_regno_mode_ok`.

Register types are also defined in `rs6000.c` and help assigning register classes. The s2pp registers, which were defined in this section, qualify as standard and vector register type and thus are added to these macros and afterwards used in register initialization.

```
enum rs6000_reg_type {
  ...
  FPR_REG_TYPE,
  S2PP_REG_TYPE,
```

```
5     ...
6     S2PP_C_REG_TYPE,
7     S2PP_ACC_REG_TYPE,
8     ...
9   };
10  ...
11  #define IS_STD_REG_TYPE(RTYPE) IN_RANGE(RTYPE, GPR_REG_TYPE,
        S2PP_REG_TYPE)
12  ...
13  #define IS_FP_VECT_REG_TYPE(RTYPE) IN_RANGE(RTYPE, VSX_REG_TYPE,
        S2PP_REG_TYPE)
14  ...
15  static void
16  rs6000_init_hard_regno_mode_ok (bool global_init_p)
17  {
18    ...
19    for (r = 32; r < 64; ++r)
20      rs6000_regno_regclass[r] = FLOAT_REGS;
21
22    if (TARGET_S2PP){
23      for (r = 32+1; r < 64; ++r)
24        rs6000_regno_regclass[r] = S2PP_REGS;
25      rs6000_regno_regclass[32] = NO_REGS;
26    }
27    ...
28    reg_class_to_reg_type[(int)S2PP_REGS] = S2PP_REG_TYPE;
29    ...
30    if (TARGET_S2PP)
31      {
32      reg_class_to_reg_type[(int)FLOAT_REGS] = NO_REG_TYPE; //S2PP_REG_TYPE
            ;
33      reg_class_to_reg_type[(int)S2PP_REGS] = S2PP_REG_TYPE; //
            S2PP_REG_TYPE;
34      rs6000_regno_regclass[S2PP_COND_REGNO] = S2PP_C_REG;
35      rs6000_regno_regclass[S2PP_ACC_REGNO] = S2PP_ACC_REG;
36      reg_class_to_reg_type[(int)S2PP_C_REG] = S2PP_C_REG_TYPE; //
            S2PP_REG_TYPE;
37      reg_class_to_reg_type[(int)S2PP_ACC_REG] = S2PP_ACC_REG_TYPE; //
            S2PP_REG_TYPE;
38      }
39    ...
40    if (TARGET_S2PP)
41      {
42      rs6000_vector_unit[V8HImode] = VECTOR_S2PP;
43      rs6000_vector_mem[V8HImode] = VECTOR_S2PP;
44      rs6000_vector_align[V8HImode] = align32;
45      rs6000_vector_unit[V16QImode] = VECTOR_S2PP;
46      rs6000_vector_mem[V16QImode] = VECTOR_S2PP;
47      rs6000_vector_align[V16QImode] = align32;
48      }
49    ...
50    if (TARGET_S2PP){
51      rs6000_constraints[RS6000_CONSTRAINT_kv] = S2PP_REGS;
52      rs6000_constraints[RS6000_CONSTRAINT_kc] = S2PP_C_REG;
```

```
53      rs6000_constraints[RS6000_CONSTRAINT_ka] = S2PP_ACC_REG;
54    }
55    ...
56  }
```

Every index in `rs6000_regno_regclass[]` is given a register class which corresponds to a register with the same index and also each register class is assigned a register type in `reg_class_to_reg_type[]`.

What is left to do, is fixing registers:

```
1  static void
2  rs6000_conditional_register_usage (void)
3  {
4    ...
5    if ((TARGET_SOFT_FLOAT || !TARGET_FPRS) && !TARGET_S2PP)
6      for (i = 32; i < 64; i++)
7        fixed_regs[i] = call_used_regs[i]
8                      = call_really_used_regs[i] = 1;
9
10   if (TARGET_S2PP){
11     fixed_regs[32] = call_used_regs[32] = call_really_used_regs[32] = 1;
12     fixed_regs[64] = call_used_regs[64] = call_really_used_regs[64] = 1;
13   }
14   ...
15 }
```

It is necessary to prevent the back-end from fixing the FPRs even though `TARGET_SOFT_FLOAT` is set but still fix registers 32 and 64 (`kc` and `ka`) manually, as these may not automatically be assigned.

One can also add **debugging information** for s2pp registers in `rs6000_debug_reg_global`. Although this is not necessary, it can be helpful at times, when using the `-mdebug` flag.

As all measures of adding the registers to `rs6000.c` are fulfilled one must add those registers to the list of possible `asm` operands in `ppx-asm.c`.

```
1  #ifdef __S2PP__
2  #define k00 0
3  #define k0  1
4  ...
5  #define k29 30
6  #define k30 31
7  #endif
```

This tells the compiler to substitute `k0` for 1 as only integers without constraints are valid assembled machine operands. `kc` and `ka` do not need to be declared here, as they cannot be referenced directly.

All that is missing now, are s2pp specific predicates in `predicates.md`. These can be copied from respective AltiVec predicates and change both the vector specific macros and the predicates' names.

```
1  ...
2  (define_predicate "s2pp_register_operand"
3    (match_operand 0 "register_operand")
```

```
4  {
5    if (GET_CODE (op) == SUBREG)
6      op = SUBREG_REG (op);
7
8    if (!REG_P (op))
9      return 0;
10
11   if (REGNO (op) > LAST_VIRTUAL_REGISTER)
12     return 1;
13
14   return S2PP_REGNO_P (REGNO (op));
15 })
16 ...
17 (define_predicate "easy_vector_constant"
18   (match_code "const_vector")
19 {
20   ...
21   if (VECTOR_MEM_S2PP_P (mode))
22     {
23   if (zero_constant (op, mode))
24     return true;
25
26   return easy_s2pp_constant (op, mode);
27     }
28   ...
29 })
30 ...
31 (define_predicate "indexed_or_indirect_operand"
32   (match_code "mem")
33 {
34   ...
35   if (VECTOR_MEM_S2PP_P (mode)
36       && GET_CODE (op) == AND
37       && GET_CODE (XEXP (op, 1)) == CONST_INT
38       && INTVAL (XEXP (op, 1)) == -16)
39     op = XEXP (op, 0);
40
41   return indexed_or_indirect_address (op, mode);
42 })
43 ...
44 (define_predicate "s2pp_indexed_or_indirect_operand"
45   (match_code "mem")
46 {
47   op = XEXP (op, 0);
48   if (VECTOR_MEM_S2PP_P (mode)
49       && GET_CODE (op) == AND
50       && GET_CODE (XEXP (op, 1)) == CONST_INT
51       && INTVAL (XEXP (op, 1)) == -16)
52     return indexed_or_indirect_address (XEXP (op, 0), mode);
53
54   return 0;
55 })
56 ...
```

If one wants to add more predicates, GCC offers a manuals entry [11, ch. 16.7].

The `easy_s2pp_constant` function, which is referred to in the listing above, checks if an operand is "splittable", ergo that all elements have the same value and the operand can be synthesized by a split instruction. To check this, the operand is analyzed sequentially if either of the two available splat instructions can synthesize the same operand. This function is transferable from an AltiVec equivalent with the exception that there is one less alternative for splatting a vector.

```
1  bool
2  easy_s2pp_constant (rtx op, enum machine_mode mode)
3  {
4    unsigned step, copies;
5
6    if (mode == VOIDmode)
7      mode = GET_MODE (op);
8    else if (mode != GET_MODE (op))
9      return false;
10
11   step = GET_MODE_NUNITS (mode) / 4;
12     copies = 1;
13
14   /* Try with a fxvsplath  */
15   if (step == 1)
16     copies <<= 1;
17   else
18     step >>= 1;
19
20   if (vspltis_constant (op, step, copies))
21     return true;
22
23   /* Try with a fxvsplatb  */
24   if (step == 1)
25     copies <<= 1;
26   else
27     step >>= 1;
28
29   if (vspltis_constant (op, step, copies))
30     return true;
31
32   return false;
33 }
```

This is one of two times, an AltiVec function (`vspltis_constant`) will be used instead of defining a new function, as this function has not to be altered. A similar function `gen_easy_s2pp_constant` can be transfered as it is basically the same but generates RTL code that will create a constant vector operand from a different operand.

As all registers are now fully implemented, it must be taken care of conflicts with FPRs. s2pp registers and FPRs share the same indexes and since they are not fixed could be identified as FPRs. For this reason any use of `FP_REGNO_P(N)` must be checked and extended with an exception `&& !TARGET_S2PP` when it is necessary. This is especially the case when dealing with hard registers and having the compiler emit register moves.

3.4 Reload

As hinted in section 2.3.4, `reload` mainly performs register allocation. Obviously it requires special handling for vector registers to `reload` because register allocation is an important part of the compilation process. Thus support for `S2PP_REGS` must be added.

As `reload` is capable of moving the contents of registers, it must be specified that s2pp registers are not directly compatible with GPRs or any other registers. At the same time do s2pp memory instructions need two indirect operands which are GPRs.

```
1  static reg_class_t
2  rs6000_secondary_reload (bool in_p,
3                           rtx x,
4                           reg_class_t rclass_i,
5                           enum machine_mode mode,
6                           secondary_reload_info *sri)
7  {...
8    /* Handle vector moves with reload helper functions.  */
9    if (ret == ALL_REGS && icode != CODE_FOR_nothing)
10     {...
11       if (GET_CODE (x) == MEM)
12       {...
13         if (rclass == GENERAL_REGS || rclass == BASE_REGS)
14           {...
15             /* Loads to and stores from vector registers can only do reg+
                   reg
16             addressing.  Altivec registers can also do (reg+reg)&(-16).
                   Allow
17             scalar modes loading up the traditional floating point
                   registers
18             to use offset addresses.  */
19             else if (rclass == VSX_REGS || rclass == ALTIVEC_REGS
20                      || rclass == FLOAT_REGS || rclass == NO_REGS
21                      || rclass == S2PP_REGS)
22               {...
23 ...
```

Because registers are quite different in their specifications and reload could possibly ask for any combination of source/destination register, s2pp register moves are restricted to other s2pp registers or memory. This creates the need for checking the register class of an RTL expression and eventually correcting it.

```
1  static enum reg_class
2  rs6000_preferred_reload_class (rtx x, enum reg_class rclass)
3  {...
4    if ((rclass == S2PP_REGS)
5        && VECTOR_UNIT_S2PP_P (mode)
6        && easy_vector_constant (x, mode)){
7      return rclass;
8    }
9  }
10 ...
11 static enum reg_class
```

```
12 rs6000_secondary_reload_class (enum reg_class rclass, enum machine_mode
       mode,
13                    rtx in)
14 {...
15   if ((regno == -1 || S2PP_REGNO_P (regno))
16       && rclass == S2PP_REGS)
17     return NO_REGS;
18 ...
19 }
20 ...
```

As `reload` does not initially support the addressing mode which AltiVec and s2pp both use, indirect addresses for s2pp must be handled like AltiVec.

```
1
2 void
3 rs6000_secondary_reload_inner (rtx reg, rtx mem, rtx scratch, bool
      store_p)
4 {...
5   switch (rclass)
6     {...
7     case S2PP_REGS:
8     ...
9     }
10 }
```

At last the mode, which is used, needs to be validated.

```
1 static bool
2 rs6000_cannot_change_mode_class (enum machine_mode from,
3                   enum machine_mode to,
4                   enum reg_class rclass)
5 {
6   if (TARGET_S2PP && rclass == S2PP_REGS
7     && (S2PP_VECTOR_MODE (from) + S2PP_VECTOR_MODE (to)) == 1)
8    return true;
9   ...
10 }
```

3.5 Built-ins, Insns and Machine Instructions

Basically the back-end is now capable of handling s2pp vector instructions. The only thing that is left, is specifying machine instructions that move registers or access memory. For this reason a machine description file `s2pp.md` is added to the back-end, which will contain all available vector instructions.

This new file must be declared in `rs6000.md` and `t-rs6000`.

```
1 ...
2 $(srcdir)/config/rs6000/s2pp.md
3 $
4 ...
```

```
...
(include "s2pp.md")
...
```

The most important insn is `*s2pp_mov<mode>`. It is generally used by `emit_move_insn` to allocate registers and memory.

```
(define_insn "*s2pp_mov<mode>"
  [(set (match_operand:FXVI 0 "nonimmediate_operand" "=Z,kv,kv,*Y,*r,*r,
      kv,kv")
   (match_operand:FXVI 1 "input_operand" "kv,Z,kv,r,Y,r,j,W"))]
  "VECTOR_MEM_S2PP_P (<MODE>mode)
   && (register_operand (operands[0], <MODE>mode)
   || register_operand (operands[1], <MODE>mode))"
  {
   switch (which_alternative)
    {
    case 0: return "fxvstax %1,%y0";
    case 1: return "fxvlax %0,%y1";
    case 2: return "fxvsel %0,%1,%1";
    case 3: return "#";
    case 4: return "#";
    case 5: return "#";
    case 6: return "fxvsel %0,0,0";
    case 7: return output_vec_const_move (operands);
    default: gcc_unreachable ();
    }
  }
  [(set_attr "type" "vecstore,vecload,vecsimple,store,load,*,vecsimple,*"
      )])
```

Insn definitions have been described earlier in section 2.3.4, therefore this insn will only be explained briefly. The name is preceded by an asterisk that renders the name not accessible because this insn shall only be referred to by the RTL sequence. Names starting with an asterisk are in general equal to no name.

The RTL template is fairly simple and states that operand `0` is set by operand `1`. Still, this insn applies for a number of cases which are specified by its constraints. The constraints of each operand build pairs. The first pair for example (`Z` and `kv`) tells the compiler that memory, which is accessed by an indirect operand `Z`, will be set by the contents of a vector register `kv`. Which machine instruction is used for each pair of constraints, is stated in the output template by `switch(which_alternative)`. The template can also be written in C code.

Each case is indexed according to the list of constraints so `case 0`'s constraints are the first pair. This case returns a machine instruction for storing a vector in memory `fxvstax`. The second operand of this instruction also contains a character besides its index, which is an operand modifier [11, ch. 6.45.2] that will cause the operand `0` to be split into an offset and an address in respective GPRs.

Other alternatives include instructions `fxvlax`, for loading a vector from memory, and `fxvsel`, for moving vector registers. `case 6` moves the contents of vector register `0` (this register is all `0`s) and thereby nulls the second operand (`j` is the respective constraint for

a zero vector).

Returning `#` as an output template is equivalent to stating, that there is no machine instruction which can perform the RTL template. This causes the compiler to look for different RTL templates, that have the same effect but also feature a machine instruction.

This process is called **insn splitting** and can also be used for optimization. A developer may define which RTL templates are equivalent by using `define_split` [11, 16.16].

As there exist non-AltiVec-specific splits for cases 3 through 5, it is not necessary to define those splits.

Insn splitting is quite common for GCC back-ends because defining insns with unspecific constraints and then splitting them allows for easier generation of code. The back-end will automatically search for code that fits the operands of the instruction. The same applies for `*s2pp_mov` which cannot be referenced by name, but by an RTL template of `mov` in `vector.md`. In order to fulfill the condition, the macro `S2PP_VECTOR_MODE_P` must be added to the insn.

```
1 (define_expand "mov<mode>"
2   [(set (match_operand:VEC_M 0 "nonimmediate_operand" "")
3         (match_operand:VEC_M 1 "any_operand" ""))]
4   "VECTOR_MEM_ALTIVEC_OR_VSX_P (<MODE>mode) || VECTOR_MEM_S2PP_P (<MODE>
      mode)"
5   {...})
```

`define_expand` is an **insn expansion** which is similar to insn splitting, but may combine several RTL templates and cannot specify any machine instruction directly. Usually this is used to compose built-ins from an instruction sequence [11, ch. 16.15].

`define_insn_and_split` is a combination of insn definition and insn splitting and is also described in the GCC manual [11, ch. 16.16].

Besides `mov` there is number of insn expansions in `vector.md` that must also apply to s2pp, which are mostly general instructions that rely on memory, or are standard names.

```
1  (define_expand "vector_load_<mode>"
2    [(set (match_operand:VEC_M 0 "vfloat_operand" "")
3          (match_operand:VEC_M 1 "memory_operand" ""))]
4    "VECTOR_MEM_ALTIVEC_OR_VSX_P (<MODE>mode) || VECTOR_MEM_S2PP_P (<MODE>
       mode)"
5    "")
6
7  (define_expand "vector_store_<mode>"
8    [(set (match_operand:VEC_M 0 "memory_operand" "")
9          (match_operand:VEC_M 1 "vfloat_operand" ""))]
10   "VECTOR_MEM_ALTIVEC_OR_VSX_P (<MODE>mode) || VECTOR_MEM_S2PP_P (<MODE>
       mode)"
11   "")
12
13 ;; Splits if a GPR register was chosen for the move
14 (define_split
15   [(set (match_operand:VEC_L 0 "nonimmediate_operand" "")
16         (match_operand:VEC_L 1 "input_operand" ""))]
17   "(VECTOR_MEM_ALTIVEC_OR_VSX_P (<MODE>mode)  || VECTOR_MEM_S2PP_P (<MODE
       >mode))
18    && reload_completed
```

```
19     && gpr_or_gpr_p (operands[0], operands[1])
20     && !direct_move_p (operands[0], operands[1])
21     && !quad_load_store_p (operands[0], operands[1])"
22   [(pc)]
23 {
24   rs6000_split_multireg_move (operands[0], operands[1]);
25   DONE;
26 })
27 ...
28
29 (define_expand "vector_s2pp_load_<mode>"
30   [(set (match_operand:VEC_X 0 "vfloat_operand" "")
31         (match_operand:VEC_X 1 "memory_operand" ""))]
32   "VECTOR_MEM_S2PP_P (<MODE>mode)"
33   "
34 {
35   gcc_assert (VECTOR_MEM_S2PP_P (<MODE>mode));
36 }")
37
38 (define_expand "vector_s2pp_store_<mode>"
39   [(set (match_operand:VEC_X 0 "memory_operand" "")
40         (match_operand:VEC_X 1 "vfloat_operand" ""))]
41   "VECTOR_MEM_S2PP_P (<MODE>mode)"
42   "
43 {
44   gcc_assert (VECTOR_MEM_S2PP_P (<MODE>mode));
45 }")
46 ...
47 (define_insn_and_split "*vec_reload_and_plus_<mptrsize>"
48   [(set (match_operand:P 0 "gpc_reg_operand" "=b")
49         (and:P (plus:P (match_operand:P 1 "gpc_reg_operand" "r")
50                        (match_operand:P 2 "reg_or_cint_operand" "rI"))
51                (const_int -16)))]
52   "(TARGET_ALTIVEC || TARGET_VSX || TARGET_S2PP) && (reload_in_progress
       || reload_completed)"
53   "#"
54   "&& reload_completed"
55   [(set (match_dup 0)
56         (plus:P (match_dup 1)
57                 (match_dup 2)))
58         (parallel [(set (match_dup 0)
59                         (and:P (match_dup 0)
60                                (const_int -16)))
61                    (clobber:CC (scratch:CC))])])
62 ...
63 (define_insn_and_split "*vec_reload_and_reg_<mptrsize>"
64   [(set (match_operand:P 0 "gpc_reg_operand" "=b")
65         (and:P (match_operand:P 1 "gpc_reg_operand" "r")
66                (const_int -16)))]
67   "(TARGET_ALTIVEC || TARGET_VSX || TARGET_S2PP) && (reload_in_progress
       || reload_completed)"
68   "#"
69   "&& reload_completed"
70   [(parallel [(set (match_dup 0)
```

```
                    (and:P (match_dup 1)
                           (const_int -16)))
               (clobber:CC (scratch:CC))])])
...
(define_expand "add<mode>3"
  [(set (match_operand:VEC_F 0 "vfloat_operand" "")
        (plus:VEC_F (match_operand:VEC_F 1 "vfloat_operand" "")
                    (match_operand:VEC_F 2 "vfloat_operand" "")))]
  "VECTOR_UNIT_ALTIVEC_OR_VSX_P (<MODE>mode) || VECTOR_UNIT_S2PP_P (<MODE
      >mode)"
  "")

(define_expand "sub<mode>3"
  [(set (match_operand:VEC_F 0 "vfloat_operand" "")
        (minus:VEC_F (match_operand:VEC_F 1 "vfloat_operand" "")
                     (match_operand:VEC_F 2 "vfloat_operand" "")))]
  "VECTOR_UNIT_ALTIVEC_OR_VSX_P (<MODE>mode) || VECTOR_UNIT_S2PP_P (<MODE
      >mode)"
  "")
...
```

Before other insns can be implemented, `case 7` of `*s2pp_mov`, which applies for constant vectors, should be emphasized.

Constant vectors have a constant value that is known at compile time. GCC will look for such vectors and check if these vectors can be splatted. This is mainly used by AltiVec, which offers a special splat instruction that takes a constant immediate value as operand. Splatting an immediate value saves a GPR and thus boosts performance. This function can achieve a similar effect for constant zero vectors by moving the zero register and is therefore used. To distinguish between cases where this applies, the function `output_vec_const_move()` in `rs6000.c` is used. This function is also used by AltiVec and the second function, that can be used by both VEs, but this time a separate case in defined inside the function. Only if the vector is a zero vector, this functions returns a machine instruction for s2pp, otherwise it will return `#`.

```
const char *
output_vec_const_move (rtx *operands)
{...
  if (TARGET_S2PP)
  {
    rtx splat_vec;
    if (zero_constant (vec, mode))
  return "fxvsel %0,0,0,0";

    splat_vec = gen_easy_s2pp_constant (vec);
    gcc_assert (GET_CODE (splat_vec) == VEC_DUPLICATE);
    operands[1] = XEXP (splat_vec, 0);
    if (!EASY_VECTOR_15 (INTVAL (operands[1]))){
  return "#";
    }
    mode = GET_MODE (splat_vec);
    if (mode == V8HImode){
  return "#";
```

```
19      }
20      else if (mode == V16QImode){
21    return "#";
22      }
23      else
24    gcc_unreachable ();
25    }
```

Since the instruction set missing an instruction to use this split, one must define a split which converts the immediate splat into a normal splat:

```
1  (define_split
2    [(set (match_operand:FXVI 0 "s2pp_register_operand" "")
3     (match_operand:FXVI 1 "easy_vector_constant" ""))]
4    "TARGET_S2PP && can_create_pseudo_p()"
5    [(set (match_dup 2) (match_dup 3))
6     (set (match_dup 0) (unspec:FXVI [(match_dup 2)] UNSPEC_FXVSPLAT))]
7    "{
8     operands[2] = gen_reg_rtx (SImode);
9     operands[3] = CONST_VECTOR_ELT(operands[1], 1);
10    }")
11 ...
12 (define_insn "*s2pp_fxvsplat<FXVI_char>"
13   [(set (match_operand:FXVI 0 "register_operand" "=kv")
14    (unspec:FXVI
15          [(match_operand:SI 1 "register_operand" "r")] UNSPEC_FXVSPLAT))]
16   "TARGET_S2PP"
17   "fxvsplat<FXVI_char> %0,%1"
18   [(set_attr "type" "vecperm")])
```

The upper RTL template, which moves a constant vector to a vector register, is split into the bottom RTL template, which inserts an intermediate step. `match_dup n` means that the operand should match the operand with the same index, that is specified somewhere else. For this reason are operands 2 and 3 specified in the following C template, which also converts operand 1 into a single integer element because all values are the same for splattable vectors. The second RTL template uses the newly created integer and moves it to a GPR, which is then splatted into a vector register.

An `unspec` operator together with an `UNSPEC_...` macro tells the compiler that the operation is not specified but has a name to distinguish it from other unspecified operations.

Now that memory insns (`fxvstax`, `fxvlax`) for s2pp exist, these can be implement in `rs6000.c` as well. In `rs6000_init_hard_regno_mode_ok` a code for store and load instructions must be assigned to supported modes in case the target flag is set. As this is also done for AltiVec, it is possible to differentiate between those option flags.

By now the compiler would already support `asm` usage as shown in 2.3.2.

This also completes the prerequisites for intrinsic functions or **built-ins**. The different steps of adding built-ins were also described in a previous internship report [12]. Therefore this section will only describe this briefly and refer to the report at times.

First, one must define insns for each vector instruction that is listed in the nux user guide [7]. Insns that work similar to `fxvstax` and `fxvlax` are implemented, in order to

support synram access. These are called `fxvoutx` and `fxvinx`.

```
;;store
(define_insn "s2pp_fxvstax<fxvstax_char><mode>"
  [(parallel
    [(set (match_operand:FXVI 0 "memory_operand" "=Z")
          (match_operand:FXVI 1 "register_operand" "kv"))
     (unspec [(const_int 0)] FXVSTAX)])]
  "TARGET_S2PP"
  "fxvstax %1,%y0,<fxvstax_int>"
  [(set_attr "type" "vecstore")])

;;load
(define_insn "s2pp_fxvlax<fxvlax_char><mode>"
  [(parallel
    [(set (match_operand:FXVI 0 "register_operand" "=kv")
          (match_operand:FXVI 1 "memory_operand" "Z"))
     (unspec [(const_int 0)] FXVLAX)])]
  "TARGET_S2PP"
  "fxvlax %0,%y1,<fxvlax_int>"
  [(set_attr "type" "vecload")])

;;synram
(define_insn "s2pp_fxvoutx<fxvoutx_char><mode>"
  [(parallel
    [(set (match_operand:FXVI 0 "memory_operand" "=Z")
          (match_operand:FXVI 1 "register_operand" "kv"))
     (unspec [(const_int 0)] FXVOUTX)])]
  "TARGET_S2PP"
  "fxvoutx %1,%y0,<fxvoutx_int>"
  [(set_attr "type" "vecstore")])

(define_insn "s2pp_fxvinx<fxvinx_char><mode>"
  [(parallel
    [(set (match_operand:FXVI 0 "register_operand" "=kv")
          (match_operand:FXVI 1 "memory_operand" "Z"))
     (unspec [(const_int 0)] FXVINX)])]
  "TARGET_S2PP"
  "fxvinx %0,%y1,<fxvinx_int>"
  [(set_attr "type" "vecload")])
```

All of these insns exist with different conditionals and are named accordingly because load and store insns are difficult to implement with an additional argument that is the conditional.

Simple arithmetic instructions exist in multiple versions that either support conditional execution or do not. Due to an issue with nux regarding conditionals and arithmetic instructions, there exist two different ways of how arithmetic instructions are implemented:

As tests revealed, the conditional execution of arithmetic instructions gives wrong results in case a condition does not apply. In this case, the result of a previous instruction is written to the return operand. Normally the operation should leave the contents of

the operand untouched instead. For this reason a workaround through insn splitting is implemented, that utilizes `fxvselect` and its conditional execution, as this instruction does work as intended [13]. This was shown by another series of tests. Still, we should offer arithmetic operations without splits and thus without conditionals, as this saves a clock cycle in comparison to having an additional `fxvselect`.

As of now, this was only done for simple arithmetic operations, `fxvadd...`, `fxvsub...` and `fxvmul...`. There also exist more complex operations that make use of the accumulator, which should not be used with conditionals as further testing is imminent and it is not clear whether those conditionals would work. Additionally would an extra instruction render the advantages of an accumulator meaningless. Accumulator insns still offer a conditional operand which will be set to `0` when assigning intrinsic names. Also does the additional operand not influence the performance of the instruction in a bad manner when set to `0`.

We will refer back to this when discussing the results.

Since this completes the insns we can go on and create built-ins for these in `rs6000-builtins.def`. First we specify macros that simplify adding intrinsics. The exact definition of those macros is described for AltiVec built-ins in the internship report [12] and we can easily transfer those for s2pp by adding the `RS6000_BTM_S2PP` built-in mask in `rs6000.h`.

```
1 #define RS6000_BTM_S2PP      MASK_S2PP    /* s2pp-mark/s2pp vectors.  */
2
3 #define RS6000_BTM_COMMON    (RS6000_BTM_ALTIVEC          \
4 ...
5                   | RS6000_BTM_S2PP)
```

We then use those new s2pp built-in macros to create built-in definitions for each insn and also each mode (halfword or byte). Most built-ins follow the scheme of a normal function that has a result and a certain number of arguments. But there is a number of insns that do not produce an output as they set the accumulator or the conditional register. These instructions need special handling and thus are defined as special built-ins.

Besides defining built-ins we also define overloads. These are used to differ intrinsics through arguments types, simplify using them and prevent false usage. Overloads are also further explained in the internship report [12].

To make overloads usable one must link them to existing built-ins as overloads are not directly connected to insns. This is done through structures that combine the built-in and overload names with a return type and up to three arguments.

To use these structures, we will add a function to `rs6000-c.c` that resolves the overloaded built-ins and is again built upon an AltiVec function that does the very same. These functions are `s2pp_build_resolved_builtin` and `s2pp_resolve_overloaded_builtin`

Which resolving function is used by the back-end, is decided in `rs6000.h`

```
1 #define REGISTER_TARGET_PRAGMAS() do {              \
2   c_register_pragma (0, "longcall", rs6000_pragma_longcall);   \
```

```
3     targetm.target_option.pragma_parse = rs6000_pragma_target_parse; \
4     if(OPTION_MASK_S2PP) \
5       targetm.resolve_overloaded_builtin = s2pp_resolve_overloaded_builtin
          ; \
6     else \
7       targetm.resolve_overloaded_builtin =
          altivec_resolve_overloaded_builtin; \
8     rs6000_target_modify_macros_ptr = rs6000_target_modify_macros; \
9   } while (0)
```

Also we need functions in `rs6000.c` that handle built-ins in general. A new function `s2pp_expand_builtin`, which is invoked by `rs6000_expand_builtin`, handles all special built-ins that belong to s2pp and picks expander functions according to the built-in's name. We therefore must create expander functions that take care of s2pp built-ins.

One group of built-ins are memory intrinsics that handle explicit memory addressing and synram intrinsics (`fxvinx`, `fxvoutx`). Expanders are needed because of the special way memory is accessed through indirect register referencing. Since AltiVec uses the same way of addressing we can reuse its implementation but use a different function names `s2pp_expand_stv_builtin` and `s2pp_expand_lv_builtin`.

Besides memory expander functions we add a new kind of expander function for s2pp. These expect no kind of return operand since a great number of instructions do not return any values because they write the accumulator or the conditional register. These functions are called `s2pp_expand_unaryx_builtin`, `s2pp_expand_binaryx_builtin` and `s2pp_expand_ternaryx_builtin` and are used in regard to the number of arguments in a built-in.

```
1   static rtx
2   s2pp_expand_unaryx_builtin (enum insn_code icode, tree exp)
3   {
4       tree arg0;
5       rtx op0, pat;
6       enum machine_mode mode0; //, tmode;
7       arg0 = CALL_EXPR_ARG (exp, 0);
8       op0 = expand_normal (arg0);
9       mode0 = insn_data[icode].operand[0].mode;
10
11      /* If we got invalid arguments bail out before generating bad rtl.
            */
12      if (arg0 == error_mark_node)
13        return const0_rtx;
14
15      if (! (*insn_data[icode].operand[0].predicate) (op0, mode0))
16        op0 = copy_to_mode_reg (mode0, op0);
17
18      pat = GEN_FCN (icode) (op0);
19      if (pat)
20        emit_insn (pat);
21      return NULL_RTX;
22  }
```

As we are already handling special built-ins we need to define those built-ins in `rs6000.c` as well. `s2pp_init_bultins` takes care of this, as it is a series of `define_builtin`

functions which must be written explicitly.

Besides built-ins that are connected to vector instructions, it is also possible to create intrinsics, that rely solely on GPRs. `vec_ext`, `vec_init` and `vec_promote` are built-ins, which are compiler-constructed sequences of machine instructions that mainly use memory. These functions are identical to AltiVec's implementation since they do not need a vector extension.

Finally we conclude on built-ins by defining alternative names for built-in functions in `s2pp.h`. A complete list of all intrinsic functions that the compiler supports at the time this thesis was written is available in table A.3.

3.6 Prologue and Epilogue

Another problem, that only emerges, when using many registers at the same time, is missing prologue and epilogue support.

There is a limited number of registers, which the compiler can use for saving values, and this number is also reduced by the amount of fixed registers Thus the compiler occasionally must store values in memory before calling a function that needs some of the available registers by calling a so called **prologue** [11, ch. 17.9.11]. The compiler then restores the registers after the function has finished through an **epilogue**.

Before registers can be saved, it must be checked which registers need to be saved. This is done by `seve_reg_p()`. In `rs6000_emit_prologue` the compiler checks each register that exists and saves them according to this function. Since the compiler checks register numbers instead of types, we need to alter the case for FPRs and add a target flag:

```
1  ...
2    if (!WORLD_SAVE_P (info) && (strategy & SAVE_INLINE_FPRS))
3      {
4        int i;
5        if (!TARGET_S2PP){
6          for (i = 0; i < 64 - info->first_fp_reg_save; i++)
7            ...
8        }
9        else{
10         for (i = 0; info->first_s2pp_reg_save + i <= LAST_S2PP_REGNO; i
               ++)
11           if (save_reg_p (info->first_s2pp_reg_save + i)){
12
13             int offset = info->s2pp_save_offset + frame_off + 16 * i;
14             rtx savereg = gen_rtx_REG (V8HImode, i+info->
                   first_s2pp_reg_save);
15             rtx areg = gen_rtx_REG (Pmode, 0);
16             emit_move_insn (areg, GEN_INT (offset));
17             rtx mem = gen_frame_mem (V8HImode,gen_rtx_PLUS (Pmode,
                   frame_reg_rtx, areg));
18             insn = emit_move_insn (mem, savereg);
19             rs6000_frame_related (insn, frame_reg_rtx, sp_off-frame_off,
                   areg,
20             GEN_INT(offset), NULL_RTX);
21         }
```

```
22        }
23        ...
24      }
25  ...
```

An `rtx`, which was a type often used in the previous listing, is short for RTL expression and can be used by `insn`s as an argument. RTL expressions can also contain information on how the operand is to be constructed as this allows chaining of operations.

If a register needs to be saved, the compiler computes an offset, that takes the current frame offset and adds 16 bytes(s2pp register size) for every register. `savereg` and `areg` are two kinds of registers that will later serve as operands for a memory instruction. `savereg` is the vector register we want to save and `areg` will be the GPR operand that holds an offset. `emit_move_insn` creates IR that stores the offset in `areg`. Now we can create a complete memory operand `mem` that takes `offset` in `areg` and `frame_reg_rtx`, which is the register holding the frame pointer, and combines them to a single operand. As both, register and memory, are specified, the compiler can emit a move insn that stores `savereg` to `mem`. Finally `rs6000_frame_related` handles the insn which was just created and performs additional customizations which are needed as the IR belongs to a function call.

After the prologue has finished, the function is compiled. Next, `rs6000_emit_epilogue` is called and works similar to prologue but restores the registers from memory. This function can not work on its but needs other functions that set parameters and prepare statements as well.

```
1       ...
2       int first_s2pp_reg_save;
3       ...
4       int s2pp_save_offset;
5       ...
6       int s2pp_size;
7       ...}
8   ...
9   int
10  direct_return (void)
11  {
12    if (reload_completed)
13      {
14        rs6000_stack_t *info = rs6000_stack_info ();
15
16        if (info->first_gp_reg_save == 32
17        ...
18        && info->first_s2pp_reg_save == LAST_S2PP_REGNO + 1
19        ...)
20      return 1;
21      }
22
23    return 0;
24  }
25  ...
26
27  static int
```

```
28 first_s2pp_reg_to_save (void)
29 {
30   int i;
31
32   /* Find lowest numbered live register.  */
33   for (i = FIRST_SAVED_S2PP_REGNO; i <= LAST_S2PP_REGNO; ++i)
34     if (save_reg_p (i))
35       break;
36
37   return i;
38 }
39 ...
40 static rs6000_stack_t *
41 rs6000_stack_info (void)
42 {...
43     info_ptr->first_s2pp_reg_save = first_s2pp_reg_to_save ();
44     info_ptr->s2pp_size = 16 * (LAST_S2PP_REGNO + 1
45                     - info_ptr->first_s2pp_reg_save);
46 ...
47 }
```

4 Results and Applications

In the course of the third chapter, the GCC back-end was extended for the s2pp vector extension. This chapter will examine features and early applications of the back-end.

So far the compiler back-end can be integrated into a compiler which creates machine code for the PPU. This enhanced compiler also supports the use of `-mcpu=nux` and `-ms2pp` target flags and a header file named `s2pp.h` which can be included the same way GCC standard header files are included. The back-end features a `vector` variable attribute which enables vector variables of different types.

These vector variables can serve as arguments for implemented s2pp intrinsic functions that cover every vector instruction which is available on nux. Additionally the new intrinsics support type detection and map automatically to the according machine instruction for each vector type if this is demanded by the user.

When using this, assigning registers as well as memory is done automatically and does not require user interaction. Despite a special bus to the synapse array, the back-end can also accesses the synapse array through special intrinsics. Specifically the intrinsics `fxv_inx` and `fxv_outx` allow to access the synapse array and load/store variables from/to there.

As many of the mentioned intrinsics as possible were designed similar to existing intrinsics for AltiVec and use the `vec_` prefix besides th `fxv_` prefix. This was done to include both, users that are accustomed to the existing vector macros and new users that are somewhat familiar with AltiVec.

In addition to intrinsics, the compiler also supports inline assembly coding in `asm` with vector instructions (as described in section 2.3.2, see listing 4.1). The user does not need to choose hard registers or implement store and load instructions by himself, as this is done by the compiler. This is possible through the addition of the `kv` constraint that marks vector registers as `r` does for GRPs. Overall inline assembly for nux became more intuitive than it had been before.

This will ultimately make it easier to combine low-level coding in high-level programs.

The new back-end also supports the use of global functions that support simple function calls.

First tests were conducted during extension development and made use of intrinsics instead of macros. Ideas for more complex tests will be discussed in chapter 5.

David Stöckel further implemented a small series of tests using a newly developed unit testing framework as part of `libnux` in order to conduct high-level software tests (example in listing 4.1).

Through these early tests, it was possible to find a bug in nux for conditional execution of arithmetic instructions [**?**]. It was possible to implement a workaround for this which was already mentioned in chapter 3. The workaround was tested and passed David

Listing 4.1: Example of nux Test.
This test directly loads two values as immediates into registers and splats them into vector registers. Those vector registers are added and the result saved in a variable. The result is then tested and also written into the mailbox for analysis.

```
1  libnux_testcase_begin("fxvpckbu");
2  vector uint8_t vec1, vec2, vec3;
3  asm volatile (  "li %3, 0x1258\n\t" /*load value into gpr*/ \
4                  "fxvsplath %0, %3\n\t" /*splat value of gpr*/ \
5                  "li %3, 0x00ff\n\t"\
6                  "fxvsplath %1, %3\n\t" /*splat value of gpr*/ \
7                  "fxvaddbm %2, %0, %1, 0 \n\t"
8                  : "=kv" (vec1), "=kv" (vec2), "=kv" (vec3) ,"=r" (value)
                       :/*handle output operands*/\
9                  : "r1"); /*reserve clobbered registers*/
10
11 libnux_mailbox_write_string("fxvpckbu\n");
12 for (uint32_t index = 0; index < 16/sizeof(vec_extract(vec3,0)); index++)
       {
13         libnux_test_equal(vec3[index], 0x1337);
14         libnux_mailbox_write_string("Index is ");
15         libnux_mailbox_write_hex(index);
16         libnux_mailbox_write_string("\tvalue is ");
17         libnux_mailbox_write_hex(vec3[index]);
18         libnux_mailbox_write_string("\n");
19         }
```

Stöckel's test, while performance was no criteria. At the same time the code size was minimally affected as only one machine instruction was added.

He also used the nux back-end for first experiments that made use of different functionalities of nux. One experiment used the PPU to increase the synaptic weights of all synapses in small steps and then measure the network activity. The same procedure was repeated for decreasing weights and the difference between activities was evaluated. Another experiment updated all synaptic weights depending on spike counts to create homeostatic behavior and yet a different experiment implemented simple Spike Timing Dependent Plasticity (STDP) that relies on accessing the hardware correlation data of HICANN-DLS [**?**]. All these experiments still used inline assembly instead of intrinsics but may be transferred to intrinsics in the future. Nonetheless did the nux back-end simplify usage of inline assembly as described earlier and allowed the user to focus on tests rather than low-level operand management.

All tests used a version of GCC that was patched and then integrated to the waf build system of the working group. Hence a patched cross-compiler is already available at the time of this thesis.

As pointed out in the beginning of this thesis we also wanted to support optimization of vector specific machine code. Until the end of this thesis could be achieved for basic optimization (with flag `-O1`) which mainly reduces memory accesses to a minimum but keeps execution order. This still gives readable assembly code and should achieve similar

performance to code written with the former standard macros.

Since the compiler did not recognize s2pp instructions before, `asm` statements had to be `volatile` to prevent vector instructions from being removed by optimization. `volatile` code can still be moved around by optimization. Other optimizations, like loop optimization or optimizations within a volatile statement are not possible. Efficient machine code was therefore relying more on the users code than usual. This changed with support of simple optimization (`-O1`) by the extended back-end. Optimization going beyond `-O1` is yet to be tested for reliability with the new back-end which will be discussed later on.

Besides these internal improvements to s2pp usage we want to point out the main advantage of the new back-end for users of nux. The main goal of this thesis was to simplify programming for nux and listing 4.2 shows this for an exemplary program.

Listing 4.2: Code with Intrinsics

```
void start() {
  vector uint8_t vec1, vec2,
      vec3;
  vec1 = fxv_splatb(8);
  vec2 = fxv_splatb(11);
  vec3 = fxv_splatb(2703);

  vec1 = fxv_mul(vec1, vec2);
  vec1 = fxv_add(vec1, vec3);
  return;
}
```

Listing 4.3: Code With Macros

```
void start() {

  fxv_splatb(0, 8);

  fxv_splatb(1, 11);

  fxv_splatb(2, 2703);
  fxv_mulbm(0, 0, 1);
  fxv_addbm(0, 0, 2);
  return;
}
```

Listing 4.4: Assembly Output for 4.2

```
start:
  li %r9,8
  fxvsplatb %f11,%r9
  li %r9,11
  fxvsplatb %f12,%r9
  li %r9,2703
  fxvsplatb %f10,%r9
  fxvmulbm %f12,%f11,%f12
  fxvaddbm %f12,%f12,%f10
  blr
```

One can see that listing 4.2 resembles standard C code and while listing 4.3 still uses the old macros for nux programming. Listing 4.4 shows the assembly output by the compiler for `-O1` optimization. Comparing 4.2 and 4.3 shows that intrinsics give more structure to the program than macros do, especially since variables are supported. Dependencies between variables are also obvious right away.

When using the macros in listing 4.3, the code is very close to the assembly output in 4.4, which would almost be identical to the assembly output of 4.3. The only differences would be register numbers, as GCC does not assign the lowest index registers first.

Another example is listing 4.5, that combines function calls and the new intrinsic `vec_extract`. This intrinsic extracts the element, which is indexed by the second argument, from a vector which is the first argument. It also presents the possibility of vector intrinsics as function arguments and return types.

Listing 4.6 shows `-O1` optimized assembly output, which illustrates the capabilities of optimization.

Overall the compiler shows promising abilities that could help users create future software for the PPU.

Listing 4.5:
Code Example with Function and Complex Intrinsic

```
vector uint8_t splat_elem(
    vector uint8_t vec,
    uint8_t elem_no) {
  uint8_t elem = vec_extract(
      vec, elem_no);
  return fxv_splatb(elem);
}

void start() {
  vector uint8_t vec1;
  volatile vector uint8_t
      vec1;

  vec1 = (vector uint8_t)
        {0, 1, 2, 3, 4, 5, 6,
             7, 8, 9, 10, 11,
             12, 13, 14, 15};

  vec2 = splat_elem(vec1, 11)
      ;

  return;
}
```

Listing 4.6:
Assembly Output for 4.5

```
start:
        stwu %r1,-48(%r1)
        li %r9,11
        fxvsplatb %f12,%r9
        li %r9,16
        fxvstax %f12,%r1,%r9
            ,0
        addi %r1,%r1,48
        blr
```

5 Discussion and Outlook

The motivation of this thesis was, to simplify programming for the PPU and provide tools to users by establishing compiler support for the nux architecture. This should help the development of new applications for the HICANN-DLS and make the system accessible to more users.

Already there exist many experiments on HICANN-DLS the addition of compiler support could help increase their number and complexity. Easy experiments can now be generated in a short amount of time and need little expertise by the user. There exist tutorials and examples on using vector types in C [**?**] that can be used as introductory reading.

At the same time can experienced users create complex experiments more easily than before, as functions are available for different purposes and optimization will help improving performance. This will become even more important, as higher levels of optimization are yet to be verified for use.

Testing the compiler in general is still a major task for the near future. The back-end needs high-level tests similar to listing 4.1 that can be run on the PPU and give comparable results. This should be extended to more complex testing scenarios that involve various combinations of intrinsics with different arguments and dependencies as well as conditional branching and looping. Running these tests with various optimization stages, would proof reliability of optimization as well. The same could be done for inline assembly code.

Also the number of low level tests should be increased to compare results, especially if tests fail. These would rather test the nux architecture than the compiler, but only testing on both sides gives meaningful results. These tests should be similar to existing tests that are written in assembly.

All tests should be conducted in simulation as well as on hardware, to create a robust testing environment. It is planned to emulate the nux architecture on hardware in the future, which would accompany existing software simulation. This would make parallel testing of hardware faster and allow for continuous testing of future PPU modifications.

Such a testing environment would allow to validate if optimization beyond `-O1` is reliable with the new back-end. This could also involve existing AltiVec tests in GCC, that are transferable to nux.

Results form these tests as well as new insights from this thesis should be featured in the nux manual, in order to provide sufficient documentation of the hardware.

As the current back-end requires GCC 4.9.2, the future development of GCC should also be considered. The most recent version of the GCC 4.9 release series dates back to August 2016 and maintenance is officially discontinued [10]. There exist newer bugs

but these were fixed through patches thanks to an active community behind GCC. It is highly unlikely that the GCC compiler itself will cause problems in the future but it might be reasonable to move to the latest 4.9 release 4.9.4 and test the back-end there as well.

Radical changes in the GCC environment are untypical for this project and the 4.9 release series is likely to be sufficient for a long time. Nonetheless exists an experimental build of GCC 7 with an early version of the s2pp back-end and also the latest binutils version by David Stöckel which has not been tested yet and only demonstrates the possibility of porting the back-end to different GCC releases if it ever became necessary. Also will the POWER architecture likely be supported by GCC to a great extend as there still exist back-ends for deprecated architectures and very minor target architectures.

Thus the most crucial development is that of the nux architecture. If it is ever decided that the PPU is to be completely redesigned, the current back-end would likely not support the new architecture. Smaller changes however, i.e. adding logical vector instructions to the instruction set, may be supported by adding these to the machine description and creating intrinsics from this as described in section 3.5 and the already mentioned internship report [12]. The back-ends structure would also allow for adding custom intrinsics that can be composed from existing machine instructions through the machine description and RTL code.

Eventually the s2pp extended GCC back-end may be usable for a reasonable amount of time and even longer if the back-end is maintained.

The PPU will play a key role in future experiments on HICANN-DLS. Experiments on this system that do not utilize the PPU are usually slower or less flexible and must be controlled from outside of the HICANN-DLS. The performance of vector processing on the PPU and direct access to the analog system offer various possibilities for future experiments. Although the processing power of the PPU is limited when compared to larger experimental setups, it fulfills the requirements for optimization and algorithms, simplistic virtual environments, interacting with analog neural networks at minimum latency and managing calibration of the system. This would allow for experiments that run solely on the PPU and do not need external supervision, which makes the HICANN-DLS a standalone system. As such, it would be able to run long-term experiments on its own and create many new testing scenarios.

A limiting factor to this is the small memory of the PPU. As the PPU should gain access to the FPGAs memory in future HICANN-DLS releases, this limitation will be resolved.

Future set-ups might feature the PPU in wafer-scale implementation, that allows for multiple experiments running in parallel or large network experiments, involving plasticity on major parts of the system.

All of this needs code that is at the same efficient and favorably of small size. Software should be easy to write, as more complex systems will cause programs to become more complicated and users should be encouraged to work on this platform.

The s2pp compiler support offers this and could open the way to additional features.

One such feature could be the GNU Project Debugger (GDB) which offers code debugging, as this is currently not possible for PPU software. GDB support might also be possible in the future but it is likely that more work on the back-end is necessary for this. Until the end of this thesis there have been no tests with GDB and the new back-end and other features such as optimization and testing would be more important. Over time this will help realizing experiments with large simulated networks or multiple standalone experiments in parallel.

Ultimately though, the future of the PPU is welded by users, developers and the applications they create for HICANN-DLS and other systems. Giving them the right tools at hand will accelerate its development.

A Appendix

A.1 Acronyms

ALU Arithmetic Logic Unit

CADC Correlation Analog Digital Converter

CISC Complex Instruction Set Computing

CPU Central Processing Unit

CR Conditional Register

DRAM Dynamic RAM

IR Intermediate Representation

FPGA Field Programmable Gate Array

FPR Floating Point Register

FPU Floating Point Unit

GCC GNU Compiler Collection

GDB GNU Project Debugger

GPP General Purpose Processor

GPR General Purpose Register

HICANN-DLS High Input Count Neural Network - Digital Learning System

ISA Instruction Set Architecture

LLVM Low Level Virtual Machine

LR Linker Register

LRA Local Register Allocator

MC Memory Controller

MMU Memory Management Unit

MSB Most Significant Bit

nux alternative name for PPU

POWER Performance Optimization With Enhanced RISC

PPU Plasiticty Processing Unit

RAM Random Access Memory

RF Register File

RTL Register Transfer Language

RISC Reduced Instruction Set Computing

rs/6000 RISC system/6000

s2pp synaptic plasiticity processor

SIMD Single Input Multiple Data

STDP Spike Timing Dependent Plasticity

SPR Special Purpose Register

SRAM Static RAM

VE Vector Extension

VCR Vector Conditional Register

VR Vector Register

VSCR Vector Status and Control Register

VRSAVE Vector Save/Restore register

VRF Vector Register File

A.2 Assembly Mnemonics

Mnemonics follow a certain pattern that has letters which can be interchanged to alter the meaning of the mnemonic, some of these characters are:

`i` indicates that the instructions uses an immediate value

`b` stands for byte and references the size of the operand

`h` stands for halfword and references the size of the operand

`w` stands for word and references the size of the operand

`s` indicates that one of the operands is shifted

`g, ge, l, le, e` stand for greater, greater or equal, less, less or equal and equal which is the possible content of the conditional register

There are also special operands which might occur in inline assembly, which behave like pointers, while others contain debugginf information.

`@l(C)` is equivalent to the lower order 16 bits of of `C` in the symbol table

`@ha(C)` is equivalent to the higher order 16 bits of of `C` in the symbol table and minds the sign extension

`.loc # # #` marks a line of code (file, line, column) in the source file

`.LVL` is a local label which can be discarded

`.LFB` marks the begin of a function

`.LFE` marks the end of a function

`.LC0` is a constant of the literals table at position `0`

mnemonic	operands	description
`add`	`RT, RA, RB`	**add** `RB` to `RA` and store the result in `RT`
`addi`	`RT, RA, SI`	**add** `SI` to `RA` and store the result in `RT`
`addis`	`RT, RA, SI`	**add** `SI` **s**hifted left by 16 bit to `RA` and store the result in `RT`
`and`	`RA, RS, RB`	`RS` and `RB` are **and**ed and the result is stored in `RT`
`b`	`target_addr`	**b**ranch to the code at `target_addr`
`ble`	`BF, target_addr`	**b**ranch to the code at `target_addr` if `BF` is **l**ess or **e**qual
`blr`		**b**ranch to the code at address in the **l**inker **r**egister
`cmp`	`BF, L, RA, RB`	`RA` and `RB` are **comp**ared and the result (`gt,lt,eq`) is stored in `BF`, `L` depicts if 32-bit or 64-bit are compared
`cmplwi`	`BF, RA, SI`	`RA` **comp**ared **l**ogically **w**ordwise with **i**mmediate `SI` and the result is stored in `BF`
`and`	`RA, RS, RB`	`RS` and `RB` are **and**ed and the result is stored in `RT`
`eieio`		enforce in-order execution of I/O
`isync`		instruction cache synchronize
`la`	`RT, D(RA)`	**l**oad **a**ggregate `D + RA` into `RT`
`li`	`RT, SI`	**l**oad **i**mmediate value `SI` into `RT`
`lis`	`RT, SI`	**l**oad **i**mmediate value `SI` **s**hifted left by 16 bit into `RT`
`lbz`	`RT, D(RA)`	**l**oad **b**yte at address `D+RA` into `RT`, fill the other bits with **z**eros
`lwz`	`RT, D(RA)`	**l**oad **w**ord at address `D+RA` into `RT`, fill the other bits with **z**eros
`mflr`	`RT`	**m**ove **f**rom **l**inker **r**egister to `RT`
`mr`	`RT, RA`	**m**ove **r**egister `RA` to `RT`
`nop`		**no-op**eration or an instruction is performed that has no effect
`rlwinm`	`RA, RS, SH, MB, ME`	**r**otate **l**eft **w**ord in `RS` by **i**mmediate `SH` bits then a**n**d with **m**ask which is 1 from `MB+32` to `ME+32` and 0 else, store to `RA`
`stw`	`RS, D(RA)`	**st**ore **w**ord from `RS` to address `D+RA`
`stwu`	`RS, D(RA)`	**st**ore **w**ord from `RS` to address `D+RA` and **u**pdate RA to `D+RA`
`sync`		synchronize data cache

Table A.1: Overview of Common Assembly Mnemonics [**?** **?**].

A.3 List of nux Intrinsics

intrinsic name	use	data types				effect
		d	a	b	c	
fxv_add vec_add	d = fxv_add(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		add a and b modulo element-wise and write the result in d
fxv_sub vec_sub	d = fxv_sub(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		subtract b from a modulo element-wise and write the result in d
fxv_mul vec_mul	d = fxv_mul(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		multiply a and b modulo element-wise and write the result in d
fxv_addfs	d = fxv_addfs(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		add a and b saturational element-wise and write the result in d
fxv_subfs	d = fxv_subfs(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		subtract b from a saturational element-wise and write the result in d
fxv_mulfs	d = fxv_mulfs(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		multiply a and b saturational element-wise and write the result in d
fxv_stax vec_st	fxv_stax(a,b,c)		vector signed char vector unsigned char vector signed short vector unsigned short	int	vector signed char* signed char* vector unsigned char* unsigned char* vector signed short* signed short* vector unsigned short* unsigned short	a is stored to memory address c + b
fxv_outx	fxv_outx(a,b,c)		vector signed char vector unsigned char vector signed short vector unsigned short	int	vector signed char* signed char* vector unsigned char* unsigned char* vector signed short* signed short* vector unsigned short* unsigned short	a is stored to synaptic address c + b
fxv_lax vec_ld	d = fxv_lax(a,b)	vector signed char vector unsigned char vector signed short vector unsigned short	int	vector signed char* signed char* vector unsigned char* unsigned char* vector signed short* signed short* vector unsigned short* unsigned short		d is read from memory address a + b
fxv_inx	d = fxv_inx(a,b)	vector signed char vector unsigned char vector signed short vector unsigned short	int	vector signed char* signed char* vector unsigned char* unsigned char* vector signed short* signed short* vector unsigned short* unsigned short		d is read from synaptic address a + b

Table A.2: List of all implemented built-ins and how they are used.

intrinsic name	use	data types				effect
		d	a	b	c	
fxv_sel	d = fxv_sel(a,b,c)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a	2-bit int	select element from a if c applies for that index otherwise select b, store the result in d
vec_extract	d = vec_exctract(a,b)	signed char unsigned char signed short unsigned short	vector signed char vector unsigned char vector signed short vector unsigned short	int		d is read from synaptic address a + b
vec_insert	d = vec_insert(a,b,c)	vector signed char vector unsigned char vector signed short vector unsigned short	signed char unsigned char signed short unsigned short	vector signed char vector unsigned char vector signed short vector unsigned short	int	d is a copy of b with element c replaced by a
vec_promote	d = vec_promote(a,b)	vector signed char vector unsigned char vector signed short vector unsigned short	signed char unsigned char signed short unsigned short	int		d is an empty vector with a at element b
vec_sh fxv_sh	d = fxv_sh(a,b)	vector signed char vector unsigned char vector signed short vector unsigned short	vector signed char vector unsigned char vector signed short vector unsigned short	int		d is a with each element shifted by b to the left
vec_splat_s16 vec_splat_u16 fxv_splatb	d = fxv_splatb(a)	vector signed char vector unsigned char	int			a is splatted into vector d , vec_splat_u16 returns an unsigned vector
vec_splat_s8 vec_splat_u8 fxv_splath	d = fxv_splath(a)	vector signed short vector unsigned short	int			a is splatted into vector d , vec_splat_u8 returns an unsigned vector
fxv_cmp	fxv_cmp(a)		vector signed char vector unsigned char vector signed short vector unsigned short			each element of a is compared to 0 and the VCR set accordingly
fxv_mtac fxv_mtacfs	fxv_mtac(a)		vector signed char vector unsigned char vector signed short vector unsigned short			moves the contents of a to the accumulator
fxv_addactacm fxv_addactacf	fxv_addactac(a)		vector signed char vector unsigned char vector signed short vector unsigned short			adds a to the accumulator and stores the value in the accumulator
fxv_addacm fxv_addacfs	d = fxv_addacm(a)	vector signed char vector unsigned char vector signed short vector unsigned short	vector signed char vector unsigned char vector signed short vector unsigned short			adds a to the accumulator and returns d
fxv_mam fxv_mafs	d = fxv_mam(a,b)	vector signed char vector unsigned char vector signed short vector unsigned short	vector signed char vector unsigned char vector signed short vector unsigned short	vector signed char vector unsigned char vector signed short vector unsigned short		multiplies a and b and adds this to the accumulator, the result is returned as d
fxv_matacm fxv_matacfs	fxv_matacm(a,b)		vector signed char vector unsigned char vector signed short vector unsigned short	vector signed char vector unsigned char vector signed short vector unsigned short		multiplies a and b and adds this to the accumulator. the result is stores in the accumulator
fxv_multacm fxv_multacfs	fxv_multacm(a,b)		vector signed char vector unsigned char vector signed short vector unsigned short	vector signed char vector unsigned char vector signed short vector unsigned short		multiplies a and b and stores the to the accumulator
fxv_addtac	fxv_addtacm(a,b)		vector signed char vector unsigned char vector signed short vector unsigned short	vector signed char vector unsigned char vector signed short vector unsigned short		adds a and b and saves the result in the accumulator
fxv_pckbu fxv_pckbl	d = fxv_pckbu(a,b)	vector signed short vector unsigned short	vector signed char vector unsigned char	vector signed char vector unsigned char		packs the upper/lower 8 bits of each element in a and b into single elements in d
fxv_upckbl fxv_upckbr	d = fxv_upckbl(a,b)	vector signed char vector unsigned char	vector signed short vector unsigned short	vector signed short vector unsigned short		unpacks the leftmost/rightmost elements of a and b into d

Table A.3: List of all implemented built-ins and how they are used.

Notes

Bibliography

[1] Aho, A. V., *Compiler*, it Informatik, 2., aktualisierte aufl. ed., XXXVI, 1253 S. pp., Pearson Studium, München [u.a.], index S. 1227-1253, 2008.

[2] Amir, A., et al., Cognitive computing programming paradigm: A corelet language for composing networks of neurosynaptic cores, in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, 2013.

[3] Cooper, K. D., and L. Torczon, *Engineering a compiler*, 2. ed. ed., XXIII, 800 S. pp., Elsevier, Amsterdam ; Heidelberg [u.a.], previous ed.: 2004 ; Hier auch später erschienene, unveränderte Nachdrucke, 2012.

[4] Davis, G., Back to the basics: Compiler optimization for smaller, faster embedded application code, `http://www.embedded.com/design/debug-and-optimization/4025587/Back-to-the-Basics-Compiler-optimization-for-smaller-faster-embedded-application-code`, accessed 2017.03.01, 2005.

[5] Esser, S. K., R. Appuswamy, P. Merolla, J. V. Arthur, and D. S. Modha, Backpropagation for energy-efficient neuromorphic computing, in *Advances in Neural Information Processing Systems 28*, edited by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, pp. 1117–1125, Curran Associates, Inc., 2015.

[6] Flik, T., *Mikroprozessortechnik und Rechnerstrukturen*, 7., neu bearb. aufl. ed., XIV, 649 S. pp., Springer, Berlin ; Heidelberg [u.a.], 2005.

[7] Friedmann, S., *nux Manual*, 2016.

[8] Friedmann, S., J. Schemmel, A. Grübl, A. Hartel, M. Hock, and K. Meier, Demonstrating hybrid learning in a flexible neuromorphic hardware system, 2016.

[9] FSF, Gcc wiki, `https://gcc.gnu.org/wiki/reload`, accessed 2017.02.26, 2013.

[10] FSF, Gcc 4.9 release series, `https://gcc.gnu.org/gcc-4.9/`, accessed 2017.02.26, 2016.

[11] FSF, *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., `https://gcc.gnu.org/onlinedocs/gccint/index.html`, 2017.

[12] Heimbrecht, A., Internship report — implementations of new altivec intrinsics, 2017.

[13] Heimbrecht, A., Bug no.2359 — conditional not working properly for arithmetic operations, `https://brainscales-r.kip.uni-heidelberg.de/issues/2359`, accessed 2017.03.06, 2017.

[] Hsu, J., Ibm's new brain [news], *IEEE Spectrum*, *51*(10), 17–19, doi:10.1109/MSPEC.2014.6905473, 2014.

[] IBM, *Developing PowerPC Embedded Application Binary Interface (EABI) Compliant Programs*, IBM Microelectronics, 1998.

[] Kim, J.-J., S.-Y. Lee, S.-M. Moon, and S. Kim, Comparison of llvm and gcc on the arm platform, *IEEE*, pp. 1–6, 2010.

[] Kozyrakis, C. E., and D. A. Patterson, Scalable, vector processors for embedded systems, *IEEE Micro*, *23*(6), 36–45, doi:10.1109/MM.2003.1261385, 2003.

[] Maass, W., Networks of spiking neurons: The third generation of neural network models, *Neural Networks*, *10*(9), 1659 – 1671, doi:http://dx.doi.org/10.1016/S0893-6080(97)00011-7, 1997.

[] Matlis, J., A brief history of supercomputers, `http://www.computerworld.com.au/article/132504/brief_history_supercomputers/`, accessed 2017.03.03, 2005.

[] NXP, *AltiVec^{TM} Technology Programming Environments Manual*, Freescale Semiconductor, 2006.

[] Ollmann, I., Altivec, `http://web-docs.gsi.de/~ikisel/reco/Systems/Altivec.pdf`, accessed 2017.03.06, 2003.

[] Park, C., M. Han, H. Lee, and S. W. Kim, Performance comparison of gcc and llvm on the eisc processor, *IEEE*, pp. 1–2, 2014.

[] Silbernagl, S., and A. Despopoulos, *Color Atlas of Physiology*, Basic sciences, Thieme, 2009.

[] Stöckel, D., Gerrit repository libnux, `https://brainscales-r.kip.uni-heidelberg.de:9443/gitweb?p=libnux.git;a=summary`, accessed 2017.03.06, 2017.

[] Stöckel, D., Gerrit repository ppu-software, `https://brainscales-r.kip.uni-heidelberg.de:9443/gitweb?p=ppu-software.git;a=summary`, accessed 2017.03.06, 2017.

[] Tanenbaum, A. S., *Computerarchitektur*, i - informatik : rechnerarchitektur, 5. aufl. ed., 829 S. pp., Pearson Studium, München [u.a.], 2006.

[] von Hagen, W., *The Definitive Guide to GCC*, section Introduction, pp. xxiii–xxix, second ed., Apress, 2006.

[] Wetzel, J., E. Silha, C. May, J. Furukawa, and G. Frazier, *PowerPC User Instruction Set Architecture Book I*, IBM Microelectronics, version 2.02 ed., 2005.

[] Wetzel, J., E. Silha, C. May, J. Furukawa, and G. Frazier, *PowerPC User Virtual Environment Architecture Book II*, IBM Microelectronics, version 2.02 ed., 2005.

Statement of Originality (Erklärung):

I certify that this thesis, and the research to which it refers, are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, March 6, 2017

.....
(signature)