RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Arthur Heimbrecht

Bachelor Thesis

KIRCHHOFF-INSTITUT FÜR PHYSIK

Department of Physics and Astronomy

University of Heidelberg

Bachelor Thesis

in Physics

submitted by

Arthur Heimbrecht

born in Speyer

Bachelor Thesis

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

Bachelor Thesis

As part of the Human Brain Project, BrainScaleS System is a unique project on many levels. This includes a processor solely used by the HICANN-DLS, which manages synaptic weights for every neuron built into one of the many wafers through a special I/O bus. To accelerate the speed at which this so called plasticity processor unit or “nux” computes all synaptic weights of every neuron used, the processor has an extended instruction set architecture that supports vector registers and single input multiple data.

This report deals with the task of adding built-in functions to an existing back-end of GCC, specifically the one already used by nux, in order to extend the already implemented set of functions according to the user’s needs.

Contents

1 Introduction

As neuromorphic computing regularly becomes a popular scientific topic, various applications for such systems emerge and make use of advantages that analogue hardware has over traditional computers . In order to do this, one must get familiar with the system that is used as it often differs in many ways from common computer architectures. Thus programing for such systems can feel odd at times as users need to abandon some familiar techniques and acquire new skills instead. This is a hurdle for many users when developing new experiments and initially takes a significant amount of time.

paper on current neuromorphic computing

An example of this is the current way of programming for the plasticity processing unit of the HICANN-DLS. The HICANN-DLS is a small scale system that features analogue emulation of neurons and synapses in networks. The PPU, as part of HICANN-DLS, can be used for implementing plasticity rules for such networks. It resembles a resembles processor architecture which was modified for this task. Implementing such plasticity rules differs from conventional programming styles. When creating code for the PPU, users are pretty much 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 which brings its own challenges to the user.

Ultimately the more a system abandons conventional elements of programming, which users are accustomed to, the more problems can emerge from this. Not only will fewer users take the initiative of writing for such systems but also can code get confusing, hard to debug and at worst even inefficient.

The main reason, users may feel reluctant to low-level programming, are the advantages of compilers.

Over decades compilers have been developed and became standard tool for programmers. At the same time compilers became more and more 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 mix high-level and low-level programming which is an atypical style of programming. It can lead to different problems as users have to adapt to this and at worst this can even lead to ineffective programs. As performance is important for neuromorphic programming, users may need an unreasonable amount of time and work to achieve simple results with this.

The PPU does not have full compiler support because of its modified processor architecture, which was developed solely neuromorphic hardware. It can be classified as an application specific instruction set architecture which is an intermediate form between general purpose processors and application specific integrated circuits. It offers a partly customized instruction set that is optimized for its applications.

The HICANN-DLS already is an experimental platform for many users even though of PPU-related challenges. Applications like in-the-loop training or stimulated time dependent plasticity have been developed and mostly work around the PPU . Even when taking the effort of learning to code for the PPU, users are constantly challenged by missing capabilities of program code such as creating parameterized functions. This leads to repetitive code or difficulties when involving calibration into code.

Offering more tools for the PPU could increase the capabilities of programs while at the same time reducing the effort it takes to develop for the PPU. Besides allowing full high-level programming, compiler support could also offer tools like code optimization and debugging features. This work therefore aims to achieve compiler support of the PPU's hardware with as many features as possible. This could simplify programming for existing users, advertise the platform to new users and possibly accelerate performance of programs on the PPU. 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. Although this could be realized on its own, it is far easier and likely more efficient to utilize existing optimization techniques that are built into virtually every compiler.

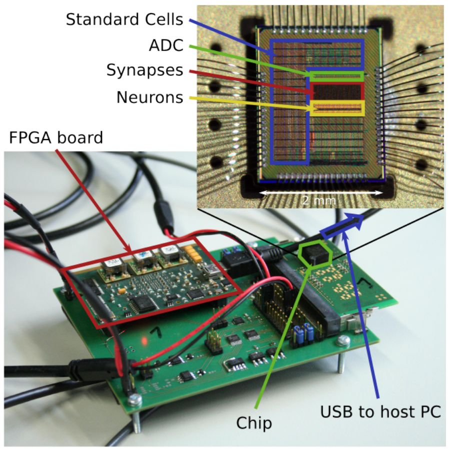


Figure 1.1: set-up of a HICANN-DLS test system

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 and go into detail for the processor and compiler that were used in this thesis. This may not render additional literature obsolete but should explain the basic concepts to an extend which is sufficient. Afterwards the process of extending the compiler is explained into some detail and the result as well as first test cases are presented. The thesis will conclude in a resume and give an outlook to future applications and development of the compiler and the PPU.

Ultimately we want to simplify PPU programming overall and give users tools at hand that allow for many interesting experiments and encourage further development of the whole system.

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). The following chapter will deal with the High Input Count Neural Network - Digital Learning System (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 was built to emulate neural networks at high speeds with low power consumption. In contrast to many other systems it is a **spike based** system. This means that neuronal activities do not follow discrete time steps and neurons send out spikes when activated.

other systems

Neurons are interconnected through dendrites, synapses and axons which 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 resting and not sending any signals. Synapses can work quite differently but have in common that there is a certain weight associated to them, which we will 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 [22].

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

At its core it 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). Pre-synaptic inputs are arranged along one side of the chip which is orthogonal to the arrangement of the neurons. They enclose a 2D field which will be our synaptic array as it mainly consists of synapse circuits. All neurons reach into the array through conductors that are organized in columns. The pre-synaptic inputs respectively have conductors that resemble rows in the array. At each intersection of of those conducting lines, a synapse is placed that thereby connects a neuron and a pre-synaptic input. Overall this gives 1024 synapses that interconnect every neuron and every synapse.

A *Field Programmable Gate Array* (FPGA) is connected to all pre-synaptic inputs and

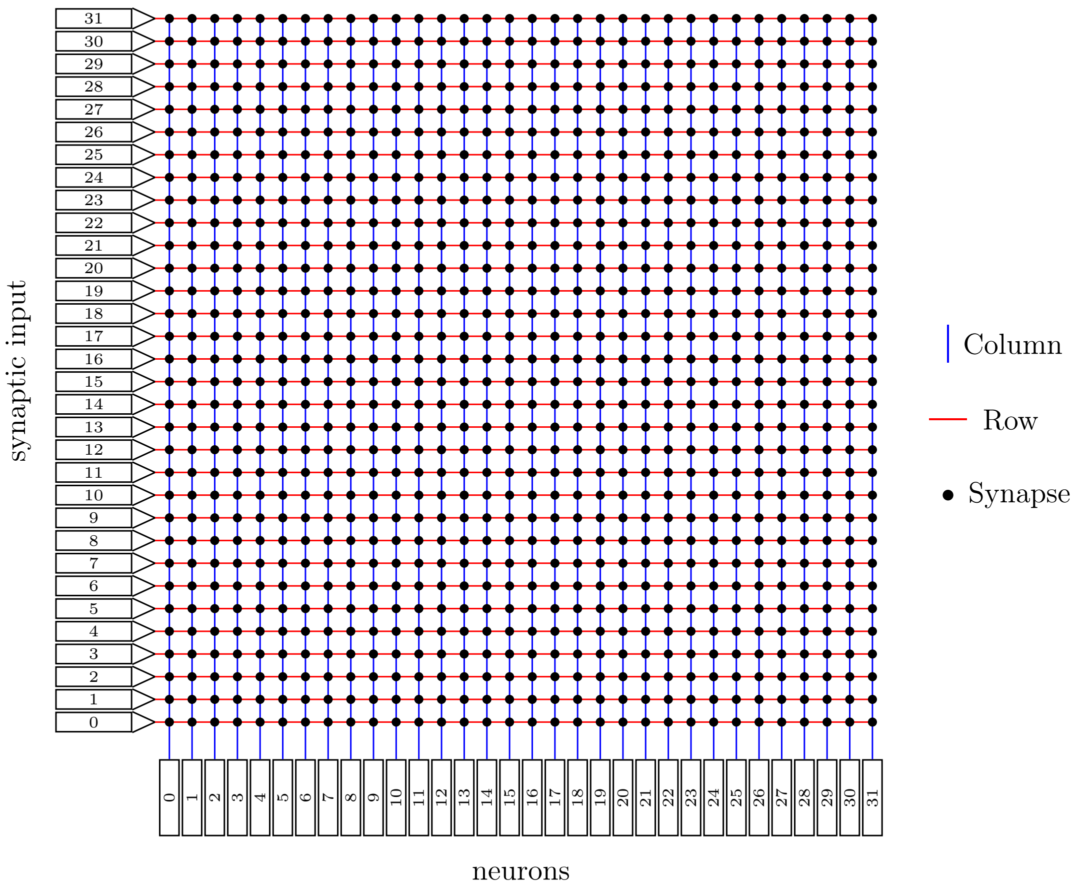


Figure 2.1: The synaptic array consists of pre-synaptic inputs (left), neurons (bottom) and 1024 synapses. All synapses along a column are connected to the respective neuron. Pre-synaptic inputs send their signal to all synapses along their respective row.

routes external spikes to these inputs. Along an input row the signal of one pre-synaptic input reaches all synapses that connected and is processed individually. Synapses in the synapse array are realized as small repetitive circuits that contain 8 bits of accessible information. The lower 6 bits are **synaptic weight** and the spare two upper bits 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 accuracy that combines the weights of two synapses to a 12-bit weight.

The Field Programmable Gate Array (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 addresses

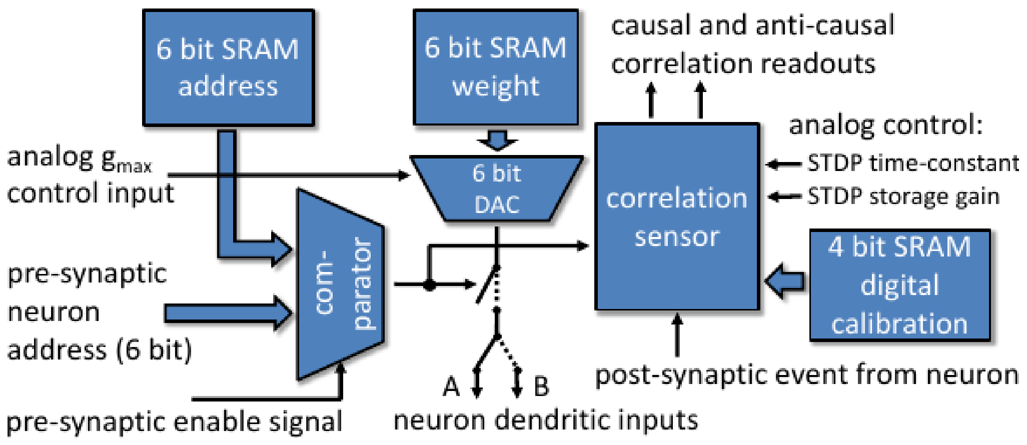

6 bit SRAM address
6 bit SRAM weight
causal and anti-causal correlation readouts
analog control:
STDP time-constant
STDP storage gain
analog g_{max} control input
6 bit DAC
correlation sensor
com-parator
pre-synaptic neuron address (6 bit)
4 bit SRAM digital calibration
A
B
post-synaptic event from neuron
pre-synaptic enable signal
neuron dendritic inputs

Figure 2.2: Block diagram of a synapse circuit (taken from [20]).

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 a neuron. 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 Analogue Digital Converter* (CADC) that converts the signal one neuron receives into digital data for analysis but can also be accessed by the Plasiticty Processing Unit (PPU) similar to the synaptic array.

The HICANN-DLS is also equipped with a processing unit that includes a vector extension and some memory for it to operate on. This is the plasticity processing unit PPU which is also connected to the synapse array for it to read and write synaptic weights.

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 of synapses.

nux refers to the architecture of the PPU.

s2pp describes the PPU's vector extension 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.

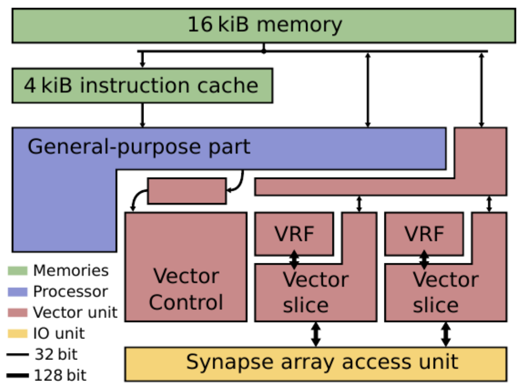


Figure 2.3: Block diagram of the nux architecture (taken from [20]).

It was developed to handle plasticity and as such applies 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 this project is to provide an alternative analogue architecture like the HICANN-DLS does, 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 architecture .

add reference

The main advantage of digital systems over analogue 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. We will now shortly get in touch with basics of such processors and explain common terms while referring to the PPU at times when it is convenient.
The PPU, which was designed by [20], is a custom processor, that is based on the Power Instruction Set Architecture (PowerISA), which was developed by IBM since 1990. Specifically the PPU uses POWER7 which was released in 2010 as a successor of the original POWER architecture and runs at 100 MHz clock frequency.

figure classic architecture

In general a microprocessor can be seen as a combination of two units which are an operational section and a control section. The control section is responsible for fetching instructions and operands, interpreting them, controlling their execution as well as reading and writing to the main memory or other buses. The operational section on the other hand saves operands

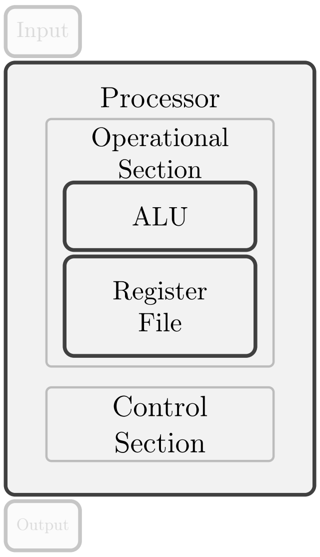


Figure 2.4: Compiler Representation

and results only as long as they are needed and performs logic or arithmetic operation 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).

The Register File (RF) can be seen as short-term memory of the processor. It consists of several repeated elements, called **registers**, that save data and have 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 be a condition for some instructions if they are to be executed or not.

The Arithmetic Logic Unit (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.

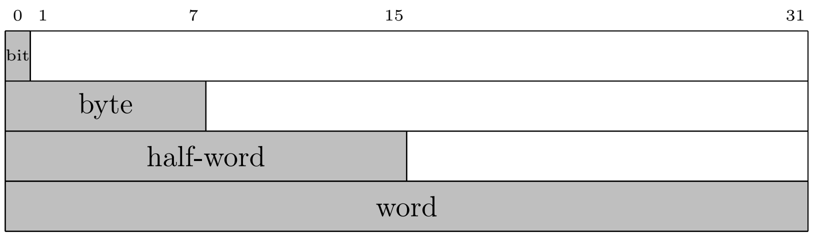


Figure 2.5: Illustration of Word Sizes for 32-bit Words

Memory is often implemented as *Random Access Memory* (RAM) which is both readable and writable. It can be split into two popular types which are *Static RAM* (SRAM) and *Dynamic RAM* (DRAM). They differ in how bits are set in RAM. SRAM uses Flip Flops to switch transistors that indicate which bit is set, while DRAM uses capacities that are charged to do so.Flip Flops allow for short access time, but can only be realized in small scales and are also expensive. Dynamic RAM (DRAM) on the other side is cheaper and usually offers more memory, but also has a longer access time.[**?**] Memory of a von-Neumann machine contains both, the program and data and is often seen as blocks with addresses. Because the smallest amount of information which we are interested in is usually a byte, each address is equivalent to one byte in memory. A program is simply a list of instructions in memory that belong to the **instruction set**. Each instruction thereby is a specific sequence of bits.

should I keep this

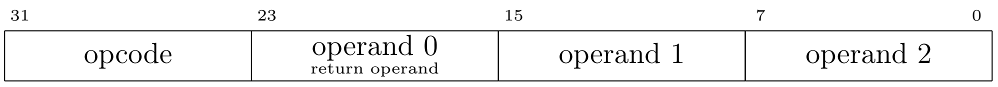


Figure 2.6: representation of machine instruction in memory

An instruction like in figure 2.7 is called a **machine isntruction** and combines several elements. The first part is called an **opcode** and is typically an 8-bit number that identifies an operation. The ALU reads this number and performs a set of so called **micro instructions** accordingly.

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

```
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
```

inhaltlicher zusammenhang

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 a value can be taken from or where it should be stored. These addresses are called **operands** and can either be a memory address or a register number.

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

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

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

As we mentioned the instruction set before, the complexity of an instruction set is also very important for performance. A complex instruction set might seem favorable to increase performance at first but a smaller instruction set also bears some advantages. Because every instruction has to be represented by a circuit in the ALU, a smaller instruction set safes space and is easier to design. In general developing a processor architecture is always a trade-off between factors like: available chip space, instruction set and design complexity, energy consumption and maximum clock frequency. Because of this, processors can be classified in two main groups:

CISC Complex Instruction Set Computer

RISC Reduced Instruction Set Computer

The latter usually has an instruction set that is reduced to simple instructions as `add` or `sub` and connects these to create more complex instructions. It also can be operated at higher clock frequencies, therefore it is the perfect architecture for applications that need to do simple arithmetic as fast as possible.

The PPU is a RISC architecture so we will focus more on Reduced Instruction Set Computer (RISC)'s key features throughout this chapter.

The complexity of an instruction set also affects developers directly, if they are working on low-level code. Programs that only consist of machine instructions a called **assembly** 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:

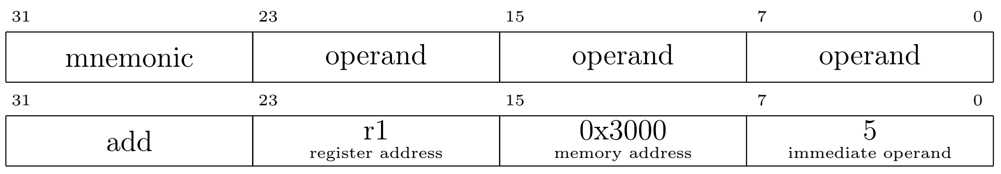


Figure 2.7: Representation of a Machine Instruction in Memory

```
add r1, 0x3000, 5
mnemonic operand/result operand operand
```

In *Reduced Instruction Set Computer* (RISC) architectures instructions typically consist of 3 operands and are usually between registers only (except for load/store memory instructions). The mnemonic in most cases is a named after the first letters of the instructions full name, which is emphasized in the following table. The operand can be of three different types which are all shown above. They either represent a specific register (r1 = register 1), a memory address (0x3000 = the value at memory location 0x3000) or an immediate value (5 = the integer 5). Register operands can also have an indirect use, which means that that content of the register is taken into account. E.g. a memory address can be saved to the register and an operation uses the value at the memory location which the register refers to. This is often the case for RISC architectures as they support only one **load** and one **store** instruction, that either loads a value from memory into a register, or stores a register's value in memory. Therefore RISC architectures often qualify as load/store architectures as the memory address is first stored in a register that afterwards is an operand of a memory instruction. This is called an indirect operand.

As address size is limited by the register, the maximum amount of memory that can be used is:

$$2^n byte \xrightarrow{\text{n = 32 bit}} 2^{32} byte \approx 4 * 10^9 byte = 4GiB \tag{2.1}$$

An overview of different assembly mnemonics for the POWER7 ISA can be seen in the Appendix as table **??**. It is accompanied by tables **??** and **??** which give further information on assembly.

Many RISC architectures have an instruction set that consists exclusively of 3 operand instructions and any instructions that seem to have less than that, are just mapped to more complex instructions that have three operands but still need only the minimum amount of clock cycles. Such simple instructions are similar to micro instructions, which were mentioned earlier, and every simple instruction has the same low number of micro instructions. RISC architectures therefore start “pipelining” instructions, which means starting the next machine instruction as the previous machine instruction just performed the first step in a clock cycle. 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.

Processors sometimes have so called **co-processors** for complex instructions that are not included in the instruction set but are still needed. An example would be multiplication on most RISCs, which would need many cycles when split in `add` instructions but 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 the FPU has its own **floating point registers** (FPRs) in a separate register file on which it performs instructions and which also has access to the memory.

keep this?

A different kind of extension are vector extensions that do the same as the FPU but for vectors instead of floats. 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:

reference

reference

- 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 *Vector Extension* (VE) of nux is the PPU's distinct feature that allows for *Single Input Multiple Data* (SIMD) operations on synaptic weights. The Vector Extension (VE) is only weakly coupled to the *General Purpose Processor* (GPP) of the PPU nand 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 that partly share their opcodes with existing AltiVec instructions. This renders no problem since the nux does not recognize AltiVec opcodes and most likely is not going to in the future. The *Vector Register File* (VRF) 32 new vector registers which are each 128-bit wide [1]. This allows for either use of vectors with 8 halfword sized elements or 16 byte sized elements which are 128 bits long as seen in figure 2.8.

But we take also 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 (VMX) and Velocity Engine for the POWER architecture. The AltiVec extension provides a similar single-precision floating point and integer SIMD instruction set. Its vector registers can either 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.

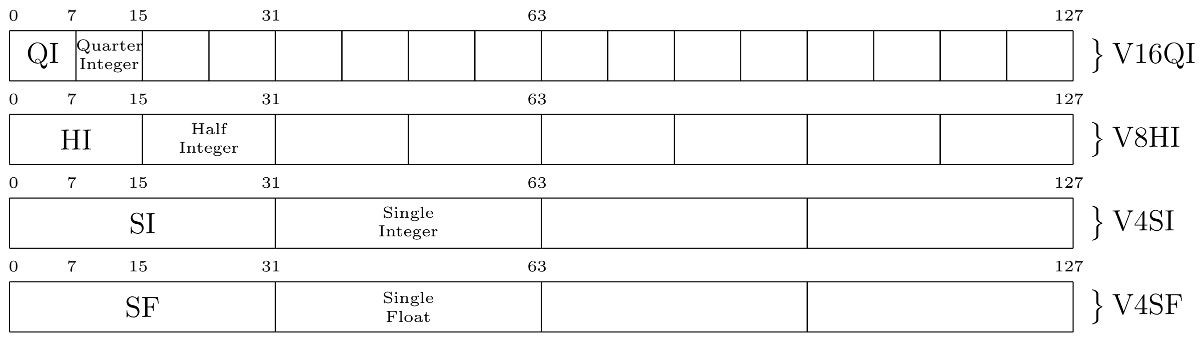


Figure 2.8: Vector structures are mostly 128 bits wide and split into common word sizes that are always powers of two.

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

The s2pp VE features a vector accumulator of 128 bits width and a **VCR!** (**VCR!**) which holds 3 bits for each half byte of the vector, making 96 bit in total. 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

Conditional Register (CR) of the PowerPC architecture. This does not allow for selective operations on individual elements through the CR, but allows for checking if all elements meet the condition in a single instruction. If element-wise selection is needed, AltiVec offers this through vector masks.

The AltiVec VE has two register that are not featured on s2p. The **VSCR!** (**VSCR!**) is responsible for detecting saturation in vector operations and decides which floating point mode is used. The **VRSAVE!** (**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.

We 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 for instructions in these reservation stations which is illustrated in figure 2.9. This allows for performing some arithmetic operations on a vector during the process of accessing a different vector in memory. The result is faster processing speed as pipelining for each instruction is also supported. Though the limiting factor for this remains the Vector Register File (VRF)'s single port for reading and writing. A more limiting factor in comparison is the shared memory interface of s2pp and General Purpose Processor (GPP).

add instruciton cache?

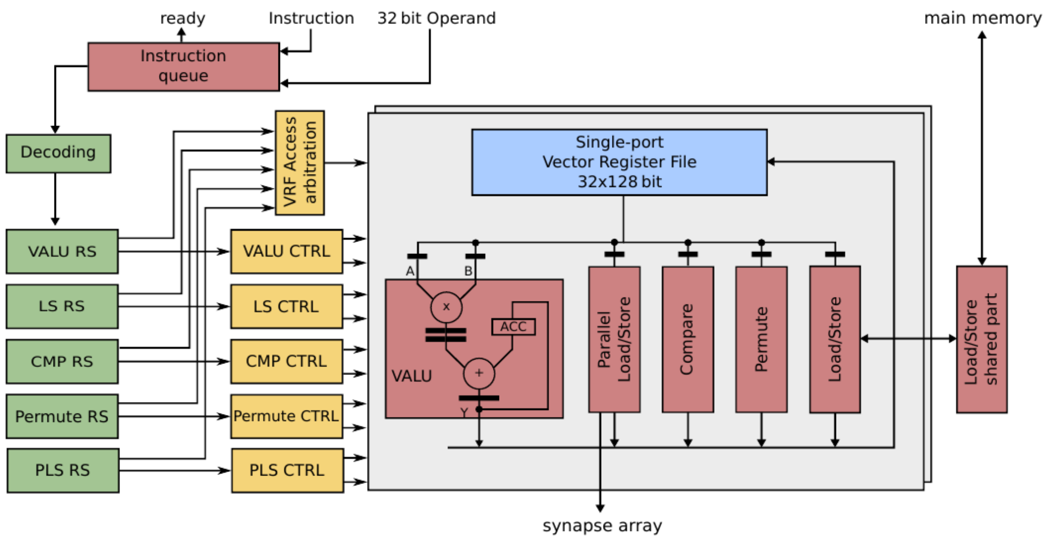


Figure 2.9: Detailed structure of the s2pp vector extension (taken from [20]).

Normally processors themselves do not keep track of memory directly. This is usually done by a *Memory Management Unit* (MMU) or a ***MC!*** (**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 Memory Management Unit (MMU)s also incorporate a cache that stores memory instructions while another memory instruction is performed. It also detects dependencies within this cache, which can be resolved there. Ultimately this results in faster transfer of data as two or more instructions do not need to access the same memory to fetch a value.

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 as it was mentioned above. 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 `sync` which does exactly what we just described. This and other memory barriers are also described in the appendix (**??**).

Listing 2.2: The memory barrier ensures that the first store was performed before the second store is issued.

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

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

add description to instruction cache

The PPU's memory is 16 kiB which is accompanied by 4 kiB of instruction cache.Together with the PPU this is called the plasticity sub-system. 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 done the same way for the whole chip configuration such as analog neuron parameters.

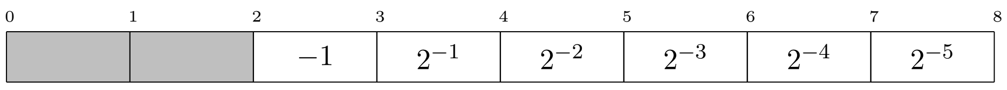
The fact, that the system views the highest as the greatest values, makes this a **big-endian** system.

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.

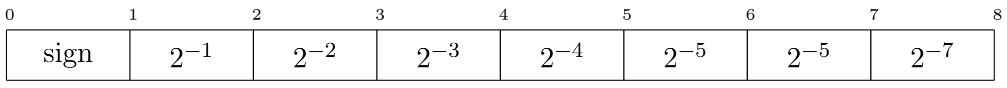
A **bus** is the connection between parts of a processor and used for data transfer. Bus speeds are very high as they transport data in parallel such as the contents of a register. Thus most buses should be as wide as a register of the processor. But as buses of such width need much space, some architectures use narrower buses with fewer bits than a register and instead use two instructions to transfer the contents of a full register. The upper half of the width is called **high order bits** and the bottom half is called **low order bits** for big-endian architectures. Systems of this sort are described as 32/16-bit architecture, which means that registers are 32 bit wide while buses are only 16 bit wide. As the higher order bits of registers are not as often used as the lower order ones, it is often enough to move the lower 16 bits. This results in less performance loss than one would initially expected.

The main feature of the PPU though is access to the synaptic array which is offered through an extra bus. Still the synapse array is mainly accessible through the main memory bus by setting the first bits similar to the spiking rate information. Using this extra bus or the instructions associated with it is nonetheless more comfortable and gives more structure to the program but does not increase performance.

When using vector instruction 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.



a) fractional representation of weights in synapse and in vector unit



b) fractional representation of weights in synapse and in vector unit

Figure 2.10: fractional representation of weights in synapse and in vector unit

Figure 2.10 displays synaptic weights as a fractional number between -1 and $1 - 2^{-5}$. The weight is the sum of the values where the bits are set to one. In contrast to this, the sign-bit changes its meaning when used in the vector unit and becomes a factor instead of a summand. This is the reason why 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 bit representation must match the intended one, which is the above mentioned fractional representation, and the values must also be correctly aligned.

An overview of all opcodes is provided by [19], which is recommended as accompanying literature besides this thesis. In general these opcodes are divided into 6 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 between 2^{-1} and 2^{-15} for halfword and 2^{-7} for byte instructions.

permute instructions perform operations on vectors that handle elements of vectors only 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 2.11 illustrates. Most often it translates a high-level, human readable programming language into a machine language that consists of basic instructions that build complicated structures. In doing so, compilers may be the essential part in everyday lives of programmers everywhere. And since the point in time when the first compilers emerged, their development played a big role in making computers such an important part of everyday life as they are today.

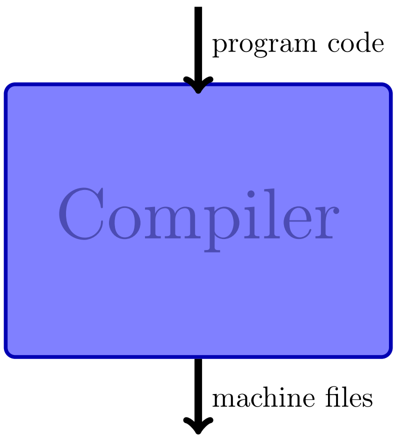
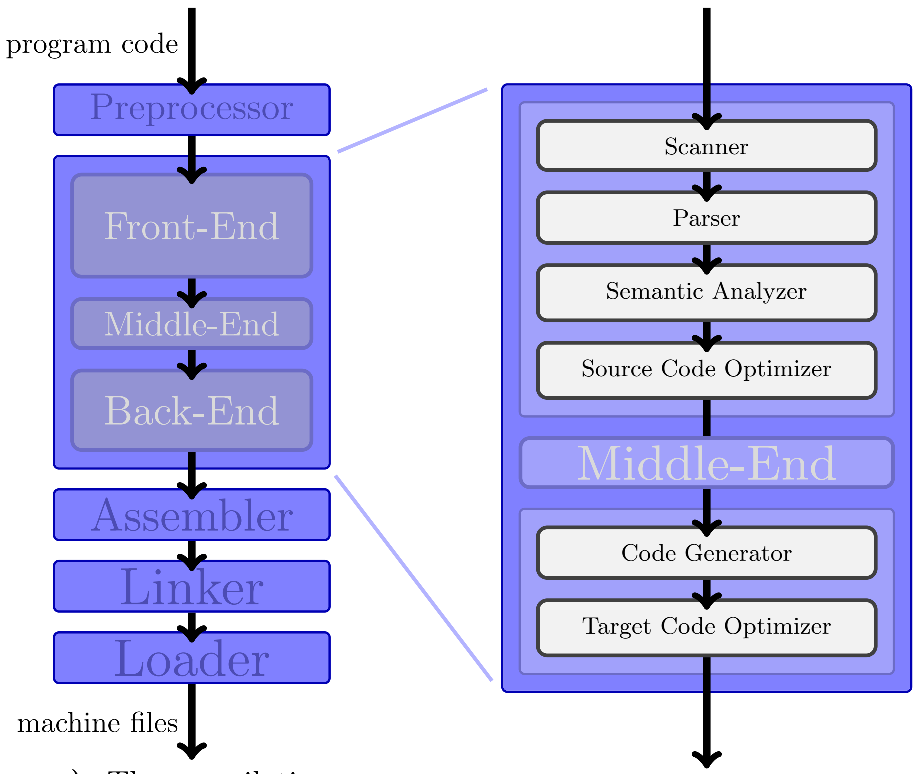


Figure 2.11: Compiler Representation

What differs compilers from the competing concept of interpreters is the separation of **compile-time** and **run-time**. As interpreters combine these two and translate a program at run-time, a compiler takes the time to read the source-language file completely (often several times) and only then creates the executable files which are executed after the process has finished. The advantages of this are simple: While a compiler takes some time at first until the program can be started, the resulting executable is next to 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 typically the combination of the amount of time it takes to compile a program and the performance of that program.



a) The compilation tree has different contributors besides the compiler.

b) The compiler goes through different phases, that belong to the front- or back-end.

Figure 2.12: Overviews of compilation stages.

A compiler is not the only contributor to translation of a program into an executable program although it is the most prominent one. Figure 2.11 illustrates the chain of tools that is involved into this process: As one can see the **preprocessor** modifies the source before it is processed by the compiler and 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** which translates the output of the compiler which is written in a language called **assembly** into actual machine code by substituting the easy-to-read string alternatives with actual opcodes. At last the **linker** combines the resulting "object-files" that the assembler emitted with standard library functions, that are already compiled, and other resources. The result is a single file that is almost executable. The only task which is left for the **loader** is assigning a base address to the relative memory references of the "relocatable" file which were used until now. The code is now fully written in machine language and ready for operation.

Since we are more interested in compilers than other components, we will take a better look at the compiler itself. Figure 2.11 shows the common separation of a compiler into **front-end**, **back-end** and an optional **middle-end**. This is done to make a compiler portable, which means allowing the compiler to work for different source-languages which are implemented in the front-end and 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 all 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.

The different phases of a compilation process are illustrated to the far right of figure 2.11. First the preprocessed source code is fed into the **scanner** that performs lexical analysis, which is combining sequences of characters to symbols and so called tokens that get associated with an attribute such as "number" or "plus-sign" and the symbol. 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 is the first form of IR. The **source code optimizer** which is the last phase of the front-end takes the syntax tree and takes a first shot at optimizing 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 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.

During these phases the compiler also generates a symbol and literal table. A **symbol table** is, as the name states, an overview of all symbols that are used in the program, it contains the symbols name and the attribute of the semantic analyzer. A **literal table** in contrast holds constants (i.e. strings) and makes them globally available by reference, as does the symbol table. This information is used by the code generator and various optimization processes.

2.3.1 Back-End and Code Generation

We now want to focus a little more on the last two phases of a compiler, which are also part of the back-end. We already stated that the back-end is responsible for code generation and target optimization and since we will keep focus on the back-end later on, we need to get used to a few other terms that are common when talking about compiler back-ends.

Usually the processes of code generation and target optimization are entangled as optimization can take place at different phases of code generation. Thus we first take a look at code generation in the back-end.

As we learned already, 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 single machine instruction can combine several IR instructions. At the end of this is a compiler typically emits a sequence of assembly instructions which we will explain later on. 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 sometimes referred to as the ISA which is in essence a list of instructions which are available and also their functionality. The compiler picks instructions according to their functionality from this list and substitutes the IR with this. Ideally a back-end thus could 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 now has two alternative approaches to solve this. First it can simply stall the programs execution as long as the instruction is executed and feed the next instruction into the processor only when the dependency is solved. This means that the compiler adds `nop`s before every instruction that needs to wait for an operand as `nop` tells the processor to wait until the previous instruction has finished. For critical memory usage the compiler can also insert `sync`s 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 partly be seen as part of the optimization process. On RISC architectures this is especially important as load and store instructions can take a few hundred times more clock cycles than normal register instructions and pipelining depends mainly on the instruction sequence. Thus the scheduler is also involved parallelization of code. As a result of this a compiler would usually accumulate all load instructions at he 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 of course needs the compiler to know the amount of time it takes for an instruction to be executed and works hand in hand with hazard detection on processor level.

At last he compiler handles register allocation which also includes memory handling. Typically the previous processes expect an ideal target machine which provides and 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 into memory because 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". It is obvious that this can hurt performance and therefore the compiler tries to keep spilling of registers to a minimum and also insert spill code at places where it delays other instructions as little as possible. At he end of register allocation the compiler assigns hard registers to the virtual registers which are now only k at a time.

During and after code generation the compiler also applies optimizations to the machine code. Any optimization to the code though must take three things into consideration, which are safety, profitability and risk/effort. The first thing which always must be fulfilled, is safety. Only if the compiler can guarantee that an optimization does not change the result of the transformed code compared to the original code it may use this optimization! Only 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 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 should be taken into account as well. If 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 would be the 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 instruction 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 we only want to explain one more kind of optimization which is called peephole optimization.

In peephole optimization the compiler only looks at a small amount of code through a "peephole" and tries to find a substitution for this specific sequence of instructions. 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. If the sequence can be substituted the peephole optimizer does so, otherwise the peephole is moved one step further and the new sequence is evaluated.

2.3.2 Assembly Basics

Following a short introduction in the previous section, we want to spend a little time on assembly as it is useful to know how to program in assembly while using C. This is done as follows:

Listing 2.3: Exemplatory Assembly invkation

```
asm volatile (  "add %0, %1, %2"
                : "=r" (dst)
                : "r" (src1), "r" (src2):);
```

The code above tells the compiler to generate the instruction `add` in assembly which is followed by three operands. The number `n` in `%n` indicates that the operand is specified by the `n+1`th description of an operand that follows. The description that follows after `:` describes the output operands. `"=r"` means that the output is to be stored in a register (letter `r` for register operand) and that the register is to be written (`=` this is called constraint). The variable in parentheses must be declared before its occurrence and of matching type (`float` would not be allowed in this case). The following description is that of the input operands and those must not be written! `r` again stands for a register operand and the variable is in parentheses, the arguments are separated by commas. After the third `:` follow clobbered (=temporarily used) registers which would also be in quotes, but these are optional arguments. `volatile` means that the compiler must not delete the following instructions due to optimization.

As a special command `asm (:::memory);` would indicate a memory barrier to the compiler, ergo the machine instructions previous to this line and the one following may not be interchanged.

keep this?

2.3.3 Intrinsics

Something that will also occur quite often later in this thesis are intrinsics. Intrinsics are sometimes also called built-in functions and resemble an intermediate form of `asm` and a high-level programming language. This means that by calling an intrinsic function, we tell the compiler to use a certain machine instruction that typically shares its name with the intrinsic. What differs an intrinsic from `asm()` is that we do not need to specify constraints or registers classes but only need to keep an eye on the type of arguments. One could easily mistake them for normal functions of library 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 need a certain knowledge of what the `asm` instruction does and what kind of operands it needs.

A typical field of 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

The GNU Compiler Collection (GCC) is a compiler suite that supports different programming languages and targets. Though normally it is seen as a build of GCC supports 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 a different machine that is also based on a different architecture. One must though build a version of GCC locally for every back-end one 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 as it was described in section 2.3 although 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 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.[23] As an open source project there is a constant development to the compiler and there exist many threads that support known bugs.

There is one major competitor though which stands besides GCC as an open source compiler suite which is Clang that is part of the LLVM (low level virtual machine). Both support running the same source code on multiple machines while LLVM actually runs intermediate code rather than actual machine code and uses GCC to generate this intermediate code for some front-ends. However while one can argue in favor for either one, GCC seems a little better suited for our application. These results have to be viewed with care, as they are based on different processor architectures but it seems like both compilers provide similar performance. Ultimately it is the personal preference of the programmer that decides which compiler one is more comfortable with and often enough he chooses that compiler. In our case I have chosen GCC over LLVM for two main reasons. One is that after all GCC follows more the traditional concept of a compiler that generates machine code at the end and also I was far more familiar with GCC than with LLVM when this decision had to be made. The other is that GCC support existed to a minimum before I started this thesis and thus there was a point to start from. This topic will be referred to later on in the discussion but for now a short motivation seems to be sufficient.

reference LLVM vs. GCC on ARM, LLVM vs. Gcc in EISC

By now GCC is a stable release version of 6.3 with version 7 in the works but we will use an older version which is used internally at the BrainScaleS project which is version 4.9.2. Additionally we will use binutils 2.25 which was patched by Simon Friedmann and since includes the opcodes and mnemonics which are supported by the nux. A complete specification of the libraries used and a handy script that builds a cross-compiler for the nux on PowerPC systems can be found there as well.

We take a special interest in the PowerPC back-end of GCC which is called rs/6000 for IBMs RISC system/6000 architecture that is equivalent to POWER. According to GCCs Internals manual [3], which we will refer 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 in gcc/config e.g. gcc/config/rs6000 that contains a minimum amount of files. These are the machine description rs6000.md which is an overview of machine instructions with additional information to each instruction and the header files rs6000.h and rs6000-protos.h and source file rs6000.c that handle the target description through macros and functions. Every back-end needs these files in the GCC source and the final back-end is build from these files through the macros and functions just mentioned. To notice a back-end in the first place the back-ends directory -here "rs6000"- must be added to the file config.gcc which also includes a list of all files in the aforementioned directory. Most back-ends include additional files which makes a back-ends complex structure clearer but these are not mandatory and we will address these later.

Instead we address one of the most important functions which unfortunately is also one of the least documented though most complicated ones. The function/process is called "reload" and is used as part of the register allocation process. [2] Specifically reload is meant to do register spilling but over almost 25 years that GCC existed until 4.9.2 it became more and more complex and basically does everything associated with register allocation (mainly moving the contents of different registers and memory around, and finding the right registers in the first place). Over the years it thus became one of the main sources of errors when constructing a back-end and was meant to be replaced several times. As of now reload is being replaced by LRA (local register allocator) but GCC 4.9.2 is not impacted by this therefore we are stuck with reload indefinitely.

To address possible errors in reload later on we now get to know one form of IR in GCC that is Register Transfer Language (RTL).

Register Transfer Language Basics

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 looks like 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 lists and thus we will need to take a look at those at last in before we take on the task of extending a GCC back-end.

We do so in looking at one of the most fundamental RTL statements first while explaining the each part at a time.

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

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

There exist manuals to basically everything which is written here and the more extensive manual will be referenced at he end of each paragraph.

`define_insn` is an RTL expression that generates an RTL equivalent to a machine instruction. One such instruction is called an insn (short for instruction) hand has several properties like a name, an RTL template, a condition template, an output template and attributes. [9] The name in this case is `add<mode>3` (`3` for three operands) where `<mode>` is to replaced by a set of values that describe a modes. [18] A mode is the form of an operand and can be something like `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. [13] There are many more modes that follow the same scheme. In this case we do not specify the mode explicitly but use an iterator that creates a `define_insn` for every valid mode we specify. [12] The second `define_insn` 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 our case `set` means that the value which is specified by the second expression is stored into the place specified by the first expression. [17] The first expression that follows is a specified operand. `match_operand` tells the compiler that what follows is a new operand. `VI2` belongs to the mode iterator we saw earlier and is to be substituted by the equivalent mode to <mode> in caps, which can be seen for the following `define_insn`. All modes `VI2` are to be substituted by the same real mode. After the mode 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. [15] The next string specifies the operands further and is meant to fine tune the predicate. It is called a constraint and matches the description which was taken in section 2.3.2. `=` again means that the register must be writable and `v` stands for an AltiVec vector register.[7] This pattern is repeated for every operand and only changes slightly. Though 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 succeeded by a mode that specifies the mode of the result.[4]

The RTL template is matched by the compiler against the RTL it generated from GIMPLE and if the template matches the RTL is substituted by the output template that follows.

After the RTL template is finished, the condition specifies if the insn may be used. It is a C expression an 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 the next `define_insn` with a matching mode. [9]

The output template is usually is similar to the `asm` template from `asm`. 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. [9]

At last the insn is completed by its attributes which hold further information about the insn that is used by the compiler internally like which effect an insn has on certain register etc.. We are less interested in this, as attributes are optional and we do not add attributes to the back-end. [6]

The attentive reader might have noticed that only RTL template is written in RTL. This is true but still do insn patterns belong into this section. The RTL not only is the most important part of an insn but we will hardly see RTL outside from RTL templates. Still should RTL be mentioned in its other form here as it is used for debugging purposes.

RTL can be split into two phases which are non-strict RTL and strict RTL. Non-strict occurs only before `reload` and is very deliberate in specifying its operands. Operands usually are virtual registers that have a unique number. `match_operand` then is replaced by `(reg:SI 1)` which tells the compiler the type of operand, the mode and the register number.[16]

Strict RTL has passed `reload` and no longer contains virtual registers but only references existing hard registers or memory.

An example of non-strict RTL and strict RTL of the same code can be seen in figure

examples of RTL code

MMU ?= memory controller PPC must handle syncing in compiler when I/O is added explain stack and frame pointer PPU instruction set

3 Extending the GCC Back-End

Since we now are familiar with the basics of processors, the PPU, compilers and GCC we can put this knowledge to use and start extending the GCC back-end. There is a number of files we will systematically edit and keep referencing to as they are important parts of the rs6000 back-end and were 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 declaration 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 rs6000 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 CPU 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 [?] at hand, as it contains an overview of existing s2pp vector instructions.

During the course of extending the GCC back-end a few things became clear:

Due to the limited documentation of the back-end itself, one must rely on comments in code and the GCC internals manual [3] As a full implementation for a vector extension already exists, the AltiVec extension should be used as a guideline for a new extension. 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, than doing so inside a function call. This will make it to find bugs as usually the function that generates an error is indicated. Also there do exist enough differences between these two vector extensions, that combining functions would not save work in the end.

Therefore we will occasionally point out when functions or other code can be inherited from AltiVec and which modifications are needed.

3.1 Adding Option Flag and nux Target

Extending the rs6000 back-end starts by adding the `nux` processor to the list of targets and also want to include mandatory flags with this. Ideally the user only has to add the option 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.

`-mstrict-align` aligns all variables in memory which means that a variable always starts at a memory address without offset. Every variable requests at least 1 byte of memory when strictly aligned

`-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.

But first there should be an option flag that activates nux' VE like `-maltivec` does for the AltiVec VE. We chose the name `-ms2pp` for this new option flag and 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 to be 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 specified by `Var`. It simultaneously specifies a macro `TARGET_S2PP` that is set to `1`. [14]

When this is done we need to specify

```
#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, we need the aforementioned flags as well to compiler nux programs. For this reason we add a processor type which combines those flags. There exist several lists that contain available targets and we need to add nux to these. First we create the `asm` **??** flag which tells the assembler which system architecture is used. As nux is based on POWER7 we 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 we saw earlier that option flags usually set masks, we set the respective masks directly. The masks we chose 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.

Unfortunately we could not 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=`.

```
#define SUBTARGET_OVERRIDE_OPTIONS                    \
...
 if (rs6000_sdata_name)                             \
     {                                        \
       if (!strcmp (rs6000_sdata_name, "none"))              \
     rs6000_sdata = SDATA_NONE;                         \
     ...
       else                                   \
     error ("bad value for -msdata=%s", rs6000_sdata_name);      \
     }                                        \
 else if (OPTION_MASK_S2PP \
          && OPTION_MASK_SOFT_FLOAT \
          && OPTION_MASK_STRICT_ALIGN \
          && !OPTION_MASK_RELOCATABLE)                  \
     {                                        \
     rs6000_sdata = SDATA_NONE;                         \
     rs6000_sdata_name = "none";                        \
     }                                        \
else if (DEFAULT_ABI == ABI_V4)                       \
...
```

It is not possible to detect in this file, if the nux flag is set. We therefore need a little work-around 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`. We 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 also set `rs6000_sdata` as we wish which is 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 somewhat not the ideal 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 we also want to check for these flags before starting compilation. First though for each flag are needed which the back-end can identify. This is done in `rs6000-c.c` where global macros can be defined:

```

void
rs6000_target_modify_macros (bool define_p, HOST_WIDE_INT flags,
                 HOST_WIDE_INT bu_mask)
{
  ...
  if ((flags & OPTION_MASK_S2PP) != 0)
   rs6000_define_or_undefine_macro (define_p, "__S2PP__");
  if ((flags & OPTION_MASK_STRICT_ALIGN) != 0)
   rs6000_define_or_undefine_macro (define_p, "_STRICT_ALIGN");
  if ((flags & OPTION_MASK_RELOCATABLE) != 0)
   rs6000_define_or_undefine_macro (define_p, "_RELOCATABLE");
  if (rs6000_sdata != SDATA_NONE)
   rs6000_define_or_undefine_macro (define_p, "_SDATA");
   ...
```

If `flags` and the respective option mask are set, `rs6000_define_or_undefine_macro` will define a macro what 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.

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

```
...
powerpc*-*-*)
    cpu_type=rs6000
    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"
    need_64bit_hwint=yes
    case x$with_cpu in
    xpowerpc64|xdefault64|x6[23]0|x970|xG5|xpower[345678]|xpower6x|xrs64a
        |xcell|xa2|xe500mc64|xe5500|Xe6500)
    cpu_is_64bit=yes
    ;;
    esac
    extra_options="${extra_options} g.opt fused-madd.opt rs6000/rs6000-
        tables.opt"
    ;;
...
```

This is done for GCC to invoke the header file as it is not referenced elsewhere.

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

```
/* _S2PP_H */
#ifndef _S2PP_H
#define _S2PP_H 1

#if !defined(__S2PP__)
#error Use the "-ms2pp" flag to enable s2pp support
#endif
#if !defined(_SOFT_FLOAT)
#error Use the "-msoft-float" flag to enable s2pp support
#endif
#if !defined(_STRICT_ALIGN)
#error Use the "-mstrict-align" flag to enable s2pp support
#endif
#if defined(_RELOCATABLE)
#error Use the "-mno-relocatable" flag to enable s2pp support
#endif
#if defined(_SDATA)
#error Use the "-msdata=none" flag to enable s2pp support
#endif
...
```

for repeti-

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` we add nux to the list of soft-float CPUs in `t-fprules`.

```
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, we start adding fundamental macros 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 [6].

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

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

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

```
...
#define VECTOR_UNIT_S2PP_P(MODE)             \
    (rs6000_vector_unit[(MODE)] == VECTOR_S2PP)
...
#define VECTOR_MEM_S2PP_P(MODE)              \
  (rs6000_vector_mem[(MODE)] == VECTOR_S2PP)
...
```

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

mention VSX?

We will also add checking for specific vector modes which are supported by s2pp. The hardware only supports two types of vectors which are vectors with byte elements (V16QI) and vectors with halfword elements (V8HI).

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

Next we need to find conditions which easily apply to `TARGET_S2PP` as they already do for `TARGET_ALTIVEC`. There exist only 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 as we want to reduce misalignment of 128-bit vectors to a minimum.

`rs6000_common_init_builtins` initializes common built-ins (which we will later refer to) and is needed by all extensions that use built-ins. In these cases we can simply expand the condition for `TARGET_S2PP`.

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

We must 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 they same way they are for AltiVec. In `rs6000_secondary_reload` indirect addressing is enforced. In `print_operand` operand modifier `y` is validated for s2pp as well. `rs6000_vector_mode_supported_p` returns true is a mode is supported by s2pp.

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

Since we established vector modes and vector units by now, we need 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;
      }
...
```

Also preferred modes when vectorizing a non-vector modes in `rs6000_preferred_simd_mode`
.

```
...
  if (TARGET_S2PP)
    switch (mode)
        {
        case HImode:
      return V8HImode;
        case QImode:
```

```
      return V16QImode;
        default:;
        }
...
```

By now we can define a vector attribute as we mentioned before. GCC already supports a vector attribute which is also used by AltiVec. Thus we can add s2pp to the `rs6000_attribute_table` and `rs6000_opt_masks[]` array with the same values as for AltiVec but changing the keyword.

```
static const struct attribute_spec rs6000_attribute_table[] =
{
    /* { name, min_len, max_len, decl_req, type_req, fn_type_req, handler
         ,
          affects_type_identity } */
    { "altivec",   1, 1, false, true,  false,
        rs6000_handle_altivec_attribute,
      false },
    { "s2pp",   1, 1, false, true,  false, rs6000_handle_s2pp_attribute,
      false },
...
struct rs6000_opt_mask {
  const char *name;      /* option name */
  HOST_WIDE_INT mask;        /* mask to set */
  bool invert;           /* invert sense of mask */
  bool valid_target;         /* option is a target option */
};

static struct rs6000_opt_mask const rs6000_opt_masks[] =
{
  { "altivec",          OPTION_MASK_ALTIVEC,        false, true  },
  ...
  { "s2pp",          OPTION_MASK_S2PP,       false, true  },
  ...
```

We also add the function `rs6000_handle_s2pp_attribute` which is copied form AltiVec but stripped off unsupported vector modes.

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

```
void
rs6000_cpu_cpp_builtins (cpp_reader *pfile)
{
  ...
  if (TARGET_S2PP){
     builtin_define ("__vector=__attribute__((s2pp(vector__)))");
     if (!flag_iso){
        builtin_define ("vector=vector");
        init_vector_keywords ();
        /* Enable context-sensitive macros.  */
        cpp_get_callbacks (pfile)->macro_to_expand =
            rs6000_macro_to_expand;
     }
  }
...
```

Additionally we indicate to the front-end that special attributes are handled by the back-end.

```
static bool
rs6000_attribute_takes_identifier_p (const_tree attr_id)
{
  if (TARGET_S2PP)
    return is_attribute_p ("s2pp", attr_id);
  else
    return is_attribute_p ("altivec", attr_id);
}
```

3.3 Registers

This section will describe, how s2pp registers are added to the back-end. Also, we will add constraints and predicates **??** for these registers.

There are three types of registers in the s2pp VE:

32 vector registers these are normal 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 became apparent that a reserved vector register, that is all zeros the entire time, will be necessary for some implementations of the back-end. This became 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.2.3.2

Since these instructions are not available, "nulling" a register becomes a problem. Therefore we reserve the first register and splat zeros into it. Moving this register, will have the same effect as `XOR`ing a register. As `OR` is also not available, we use an alternative instruction, which is `fxvselect`. `fxvselect` selects either elements of the second or the third operand depending on the condition register and its forth operand[**?**]. 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 us a simple work-around as `fxvselect` also takes only one clock cycle for execution. An alternative idea would be subtracting the same register from one another with `fxvsubm`, which also nulls the return operand. This would take more clock cycles though and is unfavorable, as we do not know how often registers need to be nulled. Ultimately it is a trade-off between having one less register at hand and wasting clock cycles continuously. The latter could get easily more of a problem than the first one and therefore we will implement the first alternative.

When talking about reserved registers 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 is declared call-used and the other half saved. We will stick to this convention but in the future it might be interesting to optimize this.

Register indexes are declared in `rs6000.md`:

```
(define_constants
  [(FIRST_GPR_REGNO      0)
...
   (LAST_GPR_REGNO       31)
   (FIRST_FPR_REGNO      32)
   (LAST_FPR_REGNO       63)
   (FIRST_S2PP_REGNO     33)
   (LAST_S2PP_REGNO      63)
   (S2PP_COND_REGNO      32)
   (S2PP_ACC_REGNO       64)
   (LR_REGNO             65)
...])
```

We reuse the reserved vector register's index 32 (this register is null) for the conditional register and use the free index for the accumulator. Registers which may be available on the same processor must not share and index! As the GPRs need the first 32 registers numbers (0-31) and there is never an FPU on nux, we want to use the 32 registers normally reserved to FPRs.

We must define which purpose the registers will serve in the back-end. This is declared by macros that are assigned a register number like 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)               \
     )
...
```

Of course we again need to go through all files and find usage of those macros. The only use the macros though is in the prologue and the epilogue, which will be discussed in the next section.

After we declared all register indexes, we must also specify them further. Each register type (or register class) needs and 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,
  ...}
...
#define REG_CLASS_NAMES                          \
  {                                    \
  ...
  "GENERAL_REGS",                            \
  "S2PP_C_REG",                              \
  "S2PP_REGS",                               \
  "FLOAT_REGS",                              \
  "S2PP_ACC_REG",                            \
  "ALTIVEC_REGS",                            \
  ...
  "ALL_REGS"}
```

in columns?

Then we need to specify the contents of our new register classes 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 where the least significant bit is the first register, the next higher order bit the second register and so on. As a number is 32-bit, 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.

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 definition of s2pp registers. Subsequent entries must not me subsets of previous masks but may extend these. Also should higher order bits follow lower order bits. Since we also added a register index which was not masked before, we also have to change some subsequent masks accordingly.

There exist macros for register classes as well, which we also need to implement. Although we only need this for general s2pp registers.

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

As all registers are specified we now have to decide on short names, that are used in assembly. Normally these are the same as the constraints that refer to these registers and an 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 are not used as constraints already. `k` is the first of those and 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},   \
  {"kc", 64}, {"ka", 65}                 \
}
```

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

After we defined these names we 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.[7]

Before we can assign register classes though, we must modify an enumeration in `rs6000.h`

```
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 classifying register classes. The s2pp registers we defined in this section qualify as standard and vector register type and thus are added to these macros.

```
enum rs6000_reg_type {
  ...
  FPR_REG_TYPE,
  S2PP_REG_TYPE,
  ...
  S2PP_C_REG_TYPE,
  S2PP_ACC_REG_TYPE,
  ...
};
...
#define IS_STD_REG_TYPE(RTYPE) IN_RANGE(RTYPE, GPR_REG_TYPE,
    S2PP_REG_TYPE)
...
#define IS_FP_VECT_REG_TYPE(RTYPE) IN_RANGE(RTYPE, VSX_REG_TYPE,
    S2PP_REG_TYPE)
...
static void
rs6000_init_hard_regno_mode_ok (bool global_init_p)
{
  ...
  for (r = 32; r < 64; ++r)
    rs6000_regno_regclass[r] = FLOAT_REGS;

  if (TARGET_S2PP){
    for (r = 32+1; r < 64; ++r)
      rs6000_regno_regclass[r] = S2PP_REGS;
    rs6000_regno_regclass[32] = NO_REGS;
  }
  ...
  reg_class_to_reg_type[(int)S2PP_REGS] = S2PP_REG_TYPE;
  ...
  if (TARGET_S2PP)
    {
    reg_class_to_reg_type[(int)FLOAT_REGS] = NO_REG_TYPE; //S2PP_REG_TYPE
        ;
    reg_class_to_reg_type[(int)S2PP_REGS] = S2PP_REG_TYPE; //
        S2PP_REG_TYPE;
    rs6000_regno_regclass[S2PP_COND_REGNO] = S2PP_C_REG;
    rs6000_regno_regclass[S2PP_ACC_REGNO] = S2PP_ACC_REG;
    reg_class_to_reg_type[(int)S2PP_C_REG] = S2PP_C_REG_TYPE; //
        S2PP_REG_TYPE;
    reg_class_to_reg_type[(int)S2PP_ACC_REG] = S2PP_ACC_REG_TYPE; //
        S2PP_REG_TYPE;
    }
  ...
  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;
    }
  ...
  if (TARGET_S2PP){
    rs6000_constraints[RS6000_CONSTRAINT_kv] = S2PP_REGS;
    rs6000_constraints[RS6000_CONSTRAINT_kc] = S2PP_C_REG;
    rs6000_constraints[RS6000_CONSTRAINT_ka] = S2PP_ACC_REG;
  }
  ...
}
```

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 the registers:

```
static void
rs6000_conditional_register_usage (void)
{
  ...
  if ((TARGET_SOFT_FLOAT || !TARGET_FPRS) && !TARGET_S2PP)
    for (i = 32; i < 64; i++)
      fixed_regs[i] = call_used_regs[i]
                    = call_really_used_regs[i] = 1;

  if (TARGET_S2PP){
    fixed_regs[32] = call_used_regs[32] = call_really_used_regs[32] = 1;
    fixed_regs[64] = call_used_regs[64] = call_really_used_regs[64] = 1;
  }
  ...
}
```

We need to prevent the back-end form fixing the FPRs even though `TARGET_SOFT_FLOAT` is set and fix the registers 32 and 64 (`kc` and `ka`).

We 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`.

```
#ifdef __S2PP__
#define k00 0
#define k0  1
...
#define k29 30
#define k30 31
#endif
```

This tells the compiler to substitute `k0` for 1 as only integers without constraints are valid machine operands.

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

```
...
(define_predicate "s2pp_register_operand"
  (match_operand 0 "register_operand")
{
  if (GET_CODE (op) == SUBREG)
    op = SUBREG_REG (op);

  if (!REG_P (op))
    return 0;

  if (REGNO (op) > LAST_VIRTUAL_REGISTER)
    return 1;

  return S2PP_REGNO_P (REGNO (op));
})
...
(define_predicate "easy_vector_constant"
  (match_code "const_vector")
{
  ...
  if (VECTOR_MEM_S2PP_P (mode))
    {
  if (zero_constant (op, mode))
    return true;

  return easy_s2pp_constant (op, mode);
    }
  ...
})
...
(define_predicate "indexed_or_indirect_operand"
  (match_code "mem")
{
  ...
  if (VECTOR_MEM_S2PP_P (mode)
      && GET_CODE (op) == AND
      && GET_CODE (XEXP (op, 1)) == CONST_INT
      && INTVAL (XEXP (op, 1)) == -16)
    op = XEXP (op, 0);

  return indexed_or_indirect_address (op, mode);
})
...
(define_predicate "s2pp_indexed_or_indirect_operand"
  (match_code "mem")
{
  op = XEXP (op, 0);
  if (VECTOR_MEM_S2PP_P (mode)
      && GET_CODE (op) == AND
      && GET_CODE (XEXP (op, 1)) == CONST_INT
      && INTVAL (XEXP (op, 1)) == -16)
    return indexed_or_indirect_address (XEXP (op, 0), mode);

  return 0;
})
...
```

If one wants to add more predicates, GCC offers a manuals entry [15].

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 through 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.

```
bool
easy_s2pp_constant (rtx op, enum machine_mode mode)
{ //cause of problem?
  unsigned step, copies;

  if (mode == VOIDmode)
    mode = GET_MODE (op);
  else if (mode != GET_MODE (op))
    return false;

  step = GET_MODE_NUNITS (mode) / 4;
    copies = 1;

  /* Try with a fxvsplath  */
  if (step == 1)
    copies <<= 1;
  else
    step >>= 1;

  if (vspltis_constant (op, step, copies))
    return true;

  /* Try with a fxvsplatb  */
  if (step == 1)
    copies <<= 1;
  else
    step >>= 1;

  if (vspltis_constant (op, step, copies))
    return true;

  return false;
}
```

This is the only time, we will use an AltiVec function (`vspltis_constant`) instead of defining a new function, as this function has not to be altered. A similar function `gen_easy_s2pp_constant` can also 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, we have to take 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 we must scan the source files for uses of `FP_REGNO_P (N)` and add an exception with `&& !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 **??** `reload` is a deprecated process in GCC that mainly performs register allocation. Obviously we need to add special handling for vector registers to `reload` because because register allocation is an important part of the compilation process. Thus we must add support for `S2PP_REGS`.

As `reload` is capable of moving the contents of registers, it must be specified that s2pp registers are not compatible with GPRs or any other registers. It is also important to implement that memory instructions take two GPRs as registers.

```
static reg_class_t
rs6000_secondary_reload (bool in_p,
                rtx x,
                reg_class_t rclass_i,
                enum machine_mode mode,
                secondary_reload_info *sri)
{...
  /* Handle vector moves with reload helper functions.  */
  if (ret == ALL_REGS && icode != CODE_FOR_nothing)
    {...
      if (GET_CODE (x) == MEM)
      {...
        if (rclass == GENERAL_REGS || rclass == BASE_REGS)
          {...
            /* Loads to and stores from vector registers can only do reg+
                reg
            addressing.  Altivec registers can also do (reg+reg)&(-16).
                Allow
            scalar modes loading up the traditional floating point
                registers
            to use offset addresses.  */
            else if (rclass == VSX_REGS || rclass == ALTIVEC_REGS
                     || rclass == FLOAT_REGS || rclass == NO_REGS
                     || rclass == S2PP_REGS)
              {...
              }}}}}
...
```

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

```
static enum reg_class
rs6000_preferred_reload_class (rtx x, enum reg_class rclass)
{...
  if ((rclass == S2PP_REGS)
      && VECTOR_UNIT_S2PP_P (mode)
      && easy_vector_constant (x, mode)){
    return rclass;
  }
}
...
static enum reg_class
rs6000_secondary_reload_class (enum reg_class rclass, enum machine_mode
    mode,
                   rtx in)
{...
```

```
  if ((regno == -1 || S2PP_REGNO_P (regno))
      && rclass == S2PP_REGS)
    return NO_REGS;
...
}
...
```

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

```

void
rs6000_secondary_reload_inner (rtx reg, rtx mem, rtx scratch, bool
    store_p)
{...
  switch (rclass)
    {...
    case S2PP_REGS:
    ...
    }
}
```

At last we also need to validate the mode which is used.

```
static bool
rs6000_cannot_change_mode_class (enum machine_mode from,
                                 enum machine_mode to,
                                 enum reg_class rclass)
{
  if (TARGET_S2PP && rclass == S2PP_REGS
    && (S2PP_VECTOR_MODE (from) + S2PP_VECTOR_MODE (to)) == 1)
   return true;
  ...
}
```

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 to do, is specifying machine instructions that move registers or access memory. For this reason we add a machine description file `s2pp.md` to the back-end which will contain all available vector instructions.

We must index this new file in `rs6000.md` and `t-rs6000`.

```
...
$(srcdir)/config/rs6000/s2pp.md
$
...
```

```
...
(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,*")])
```

We explained the basics of insn definition earlier in section **??** thus we will only explain specific for this insn. The name is proceeded by an asterisk that renders the name not accessible directly because this insn shall only be referred to by the RTL sequence that follows. 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.

Each case is indexed according to the list of constraints so we take a look at `case 0`, which is the first pair. This alternative return 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 [5] 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 that register (`j` is the respective constraint for a zero vector).

Returning `#` as 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 has 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` [10].

As there exist non-AltiVec-specific splits for cases 3 through 5 in AltiVec, we do not need to define splits ourself.

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 then automatically search for code that fits the operands of the instruction. The same applies for `*s2pp_mov` which can not be referenced by name, but by an RTL template of `mov` in `vector.md`. Inorder to fulfill the condition, we add the `S2PP_VECTOR_MODE_P` to the insn.

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

`define_expand` is an **insn expansion** which is similar to insn splitting, but may combine several RTL templates and can not specify any machine instruction directly. Usually this is used to compose built-ins from an instruction sequence [8].

`define_insn_and_split` is a combination of insn definition and insn splitting and is also described in the GCC manual [10].

Besides `mov` there is number of insn expansions in `vector.md` that must also apply to s2pp.

```
(define_expand "vector_load_<mode>"
  [(set (match_operand:VEC_M 0 "vfloat_operand" "")
        (match_operand:VEC_M 1 "memory_operand" ""))]
  "VECTOR_MEM_ALTIVEC_OR_VSX_P (<MODE>mode) || VECTOR_MEM_S2PP_P (<MODE>
     mode)"
  "")

(define_expand "vector_store_<mode>"
  [(set (match_operand:VEC_M 0 "memory_operand" "")
        (match_operand:VEC_M 1 "vfloat_operand" ""))]
  "VECTOR_MEM_ALTIVEC_OR_VSX_P (<MODE>mode) || VECTOR_MEM_S2PP_P (<MODE>
     mode)"
  "")

;; Splits if a GPR register was chosen for the move
(define_split
  [(set (match_operand:VEC_L 0 "nonimmediate_operand" "")
        (match_operand:VEC_L 1 "input_operand" ""))]
  "(VECTOR_MEM_ALTIVEC_OR_VSX_P (<MODE>mode)  || VECTOR_MEM_S2PP_P (<MODE
     >mode))
   && reload_completed
   && gpr_or_gpr_p (operands[0], operands[1])
   && !direct_move_p (operands[0], operands[1])
   && !quad_load_store_p (operands[0], operands[1])"
  [(pc)]
{
  rs6000_split_multireg_move (operands[0], operands[1]);
  DONE;
})
...

(define_expand "vector_s2pp_load_<mode>"
  [(set (match_operand:VEC_X 0 "vfloat_operand" "")
        (match_operand:VEC_X 1 "memory_operand" ""))]
  "VECTOR_MEM_S2PP_P (<MODE>mode)"
  "
{
  gcc_assert (VECTOR_MEM_S2PP_P (<MODE>mode));
}")

(define_expand "vector_s2pp_store_<mode>"
  [(set (match_operand:VEC_X 0 "memory_operand" "")
        (match_operand:VEC_X 1 "vfloat_operand" ""))]
  "VECTOR_MEM_S2PP_P (<MODE>mode)"
  "
{
  gcc_assert (VECTOR_MEM_S2PP_P (<MODE>mode));
}")
...
(define_insn_and_split "*vec_reload_and_plus_<mptrsize>"
  [(set (match_operand:P 0 "gpc_reg_operand" "=b")
        (and:P (plus:P (match_operand:P 1 "gpc_reg_operand" "r")
                       (match_operand:P 2 "reg_or_cint_operand" "rI"))
               (const_int -16)))]
  "(TARGET_ALTIVEC || TARGET_VSX || TARGET_S2PP) && (reload_in_progress
      || reload_completed)"
  "#"
  "&& reload_completed"
  [(set (match_dup 0)
        (plus:P (match_dup 1)
                (match_dup 2)))
        (parallel [(set (match_dup 0)
                        (and:P (match_dup 0)
                               (const_int -16)))
                   (clobber:CC (scratch:CC))])])
...
(define_insn_and_split "*vec_reload_and_reg_<mptrsize>"
  [(set (match_operand:P 0 "gpc_reg_operand" "=b")
        (and:P (match_operand:P 1 "gpc_reg_operand" "r")
               (const_int -16)))]
  "(TARGET_ALTIVEC || TARGET_VSX || TARGET_S2PP) && (reload_in_progress
      || reload_completed)"
  "#"
  "&& reload_completed"
  [(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)"
  "")
...
}])))]]}}]}]} !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! edit this away
    , included for spellchecking, some bracked not escaped
```

Before we can continue with other insns, we shall go back to `*s2pp_mov` and take a look at `case 7` which applies for constant vectors.

Constant vectors have a constant value that are 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. Still, we keep this function, since we can achieve a similar effect for constant zero vectors by moving the zero register. To distinguish between cases where this applies, we use the function `output_vec_const_move()` in`rs6000.c` which is also used by AltiVec. Only if the vector is a zero vector, this functions returns a machine instruction, otherwise it will return `#`.

```
const char * /*p_o_i*/
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 "#";
    }
    else if (mode == V16QImode){
  return "#";
    }
    else
  gcc_unreachable ();
  }
```

Since we are now missing an instruction to use this split we will define a split ourselves which converts the immediate splat into a normal splat:

```
(define_split
  [(set (match_operand:FXVI 0 "s2pp_register_operand" "")
   (match_operand:FXVI 1 "easy_vector_constant" ""))]
  "TARGET_S2PP && can_create_pseudo_p()"
  [(set (match_dup 2) (match_dup 3))
   (set (match_dup 0) (unspec:FXVI [(match_dup 2)] UNSPEC_FXVSPLAT))]
  "{
   operands[2] = gen_reg_rtx (SImode);
   operands[3] = CONST_VECTOR_ELT(operands[1], 1);
   }")
...
(define_insn "*s2pp_fxvsplat<FXVI_char>"
  [(set (match_operand:FXVI 0 "register_operand" "=kv")
   (unspec:FXVI
         [(match_operand:SI 1 "register_operand" "r")] UNSPEC_FXVSPLAT))]
  "TARGET_S2PP"
  "fxvsplat<FXVI_char> %0,%1"
  [(set_attr "type" "vecperm")])
```

We split the upper RTL template, which moves a constant vector to a vector register 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. 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 `UNSPEC_...` tells the compiler that the operation is not specified but has a name to distinguish it from other unspecified operations.

Now that memory insns for s2pp exist, we can also implement them in `rs6000.c` as well. In `rs6000_init_hard_regno_mode_ok` we must assign code for store and load instructions to supported modes in case the target flag is set. As this is also done for AltiVec, we simply 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 of **built-ins**. The different steps of adding built-ins were also described in a previous internship report [21]. Therefore we will only describe this briefly and refer to the report at times.

First we start defining insns for each vector instruction that is listed in the nux user guide [19]. In order to allow access to the synram we implement insns that work similar to `fxvstax` and `fxvlax` and 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 hard 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 conditionals are realized As tests revealed, the conditional execution of arithmetic instructions perform unexpectedly 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 we implement a workaround through insn splitting, that utilizes `fxvselect` and its conditional execution, as this instruction does work as intended. This was shown by another series of tests. Still, we should offer arithmetic operations without splits 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. Nonetheless were accumulator instructions implemented together with a conditional argument which will allow for easier testing later on. 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 though we add macros that simplify adding intrinsics. The exact definition of those macros is described for AltiVec built-ins in the internship report [21] and we can easily transfer those for s2pp by adding the `RS6000_BTM_S2PP` built-in mask in `rs6000.h`.

```
#define RS6000_BTM_S2PP      MASK_S2PP   /* s2pp-mark/s2pp vectors.  */

#define RS6000_BTM_COMMON   (RS6000_BTM_ALTIVEC          \
...
                  | 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 easier and prevent false usage. Overloads are also further explained in the intership report[21].

To make overloads usable one must link them to existing buili-ns 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`

```
#define REGISTER_TARGET_PRAGMAS() do {              \
  c_register_pragma (0, "longcall", rs6000_pragma_longcall);    \
  targetm.target_option.pragma_parse = rs6000_pragma_target_parse; \
  if(OPTION_MASK_S2PP) \
    targetm.resolve_overloaded_builtin = s2pp_resolve_overloaded_builtin
       ; \
  else \
      targetm.resolve_overloaded_builtin =
          altivec_resolve_overloaded_builtin; \
    rs6000_target_modify_macros_ptr = rs6000_target_modify_macros; \
} 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 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 intrinsicsi (`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.

```
static rtx
s2pp_expand_unaryx_builtin (enum insn_code icode, tree exp)
{
      tree arg0;
      rtx op0, pat;
      enum machine_mode mode0; //, tmode;
      arg0 = CALL_EXPR_ARG (exp, 0);
      op0 = expand_normal (arg0);
      mode0 = insn_data[icode].operand[0].mode;

      /* If we got invalid arguments bail out before generating bad rtl.
          */
      if (arg0 == error_mark_node)
        return const0_rtx;

      if (! (*insn_data[icode].operand[0].predicate) (op0, mode0))
        op0 = copy_to_mode_reg (mode0, op0);

      pat = GEN_FCN (icode) (op0);
      if (pat)
        emit_insn (pat);
      return NULL_RTX;
}
```

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 the appendix .1.

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 an ending 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]. The compiler then restores the registers after the function has finished through an **epilogue**.

Before the compiler can save registers, it must check 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 compilers checks register numbers instead of types, we need to alter the case for FPRs and add a target flag:

```
...
  if (!WORLD_SAVE_P (info) && (strategy & SAVE_INLINE_FPRS))
    {
      int i;
      if (!TARGET_S2PP){
        for (i = 0; i < 64 - info->first_fp_reg_save; i++)
          ...
      }
      else{
        for (i = 0; info->first_s2pp_reg_save + i <= LAST_S2PP_REGNO; i
            ++)
          if (save_reg_p (info->first_s2pp_reg_save + i)){

            int offset = info->s2pp_save_offset + frame_off + 16 * i;
            rtx savereg = gen_rtx_REG (V8HImode, i+info->
                first_s2pp_reg_save);
            rtx areg = gen_rtx_REG (Pmode, 0);
            emit_move_insn (areg, GEN_INT (offset));
            rtx mem = gen_frame_mem (V8HImode,gen_rtx_PLUS (Pmode,
                frame_reg_rtx, areg));
            insn = emit_move_insn (mem, savereg);
            rs6000_frame_related (insn, frame_reg_rtx, sp_off-frame_off,
                areg,
            GEN_INT(offset), NULL_RTX);
        }
      }
      ...
    }
```

```
...
```

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

If 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.

fxvstax and fxvlax

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.

```
    ...
    int first_s2pp_reg_save;
    ...
    int s2pp_save_offset;
    ...
    int s2pp_size;
    ...}
...
int
direct_return (void)
{
  if (reload_completed)
    {
      rs6000_stack_t *info = rs6000_stack_info ();

      if (info->first_gp_reg_save == 32
      ...
      && info->first_s2pp_reg_save == LAST_S2PP_REGNO + 1
      ...)
    return 1;
    }

  return 0;
}
...

static int
first_s2pp_reg_to_save (void)
{
  int i;

  /* Find lowest numbered live register.  */
  for (i = FIRST_SAVED_S2PP_REGNO; i <= LAST_S2PP_REGNO; ++i)
    if (save_reg_p (i))
      break;

  return i;
}
...
static rs6000_stack_t *
rs6000_stack_info (void)
{...
    info_ptr->first_s2pp_reg_save = first_s2pp_reg_to_save ();
    info_ptr->s2pp_size = 16 * (LAST_S2PP_REGNO + 1
                    - info_ptr->first_s2pp_reg_save);
...
}
```

stack boundary -> put this together frame offsetr stackoffset

4 Results and Applications

Now that the GCC back-end supports s2pp, we want to examine features and early applications of this.

So far the compiler back-end can be built into a working compiler which creates working machine files for the PPU. This newly built 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. Target flags and header file together will this allow for using the `vector` attribute which creates 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 need for user interaction. Despite the different bus, 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 accompanied to the existing vector macros and new users that are somewhat familiar with AltiVec.

In Addition to intrinsics the compiler also supports coding in `asm` with vector instructions. In contrast to before the user does not need to choose hard registers or implement store and load instructions, as this is done by the compiler for `asm` as well. This is possible through the addition of the `kv` constraint that marks vector registers as `r` does for GRPs. Overall `asm` for nux became more intuitive than it had been before.

!listing!

This will ultimately make it easier to combine low-level coding in high-level programs if this is ever needed.

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

First tests which conducted during extension development, made use of intrinsic over macros and produced working machine code as well as correct results for small examples. More complex tests will be discussed in 5 David Stöckel further implemented a small series of tests which used a newly implemented unit testing framework (`libnux`) along intrinsics for nux in order to conduct high-level software tests. Through these early test, he was able to find a bug which previously was not known and could be identified as undocumented behavior of the nux for conditional execution of arithmetic instructions. In the end it was possible to implement a workaround for this which was already mentioned in **??** that has minimal influence on performance and code size.

He also used the nux back-end for first experiments that made use of different functionalities of nux. One experiment had the PPU automatically increasing the synaptic weights of all synapses up to a maximum value and then decreasing the weights the same way. Another experiment updated all synaptic weights depending on spike counts to create homeostasis and yet a different experiment implemented simple STDP that relies on accessing the hardware correlation of HICANN-DLS. As all these experiments still used `asm` instead of intrinsics we have yet to encourage the use of intrinsics over `asm` in the future. Nonetheless did the nux back-end simplify usage of `asm` 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 group. Hence a patched cross-compiler is already commonly available.
As pointed out in the beginning of this thesis we also wanted to support optimization of vector specific machine code. As for now this could be achieved for simple optimization (`-O1`) which mainly reduces memory access 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.

This is due to the way these macros were implemented. Since the compiler did not recognize s2pp instructions before, `asm` statements had to be `volatile` to prevent vector instructions to be removed by optimization as `volatile` statements are protected from any optimization. Thus only code around vector instruction was affected by optimization and performance of vector instructions was up to the user's skill. Optimization going beyond (`-O1`) is yet to be tested for reliability with the new back-end which will be discussed later on.

Besides these rather 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 simplify programming for nux and listing **??** shows this for an exemplary program. We compare an existing test program that showcases the abilities of nux to an equivalent program that uses intrinsics instead of C macros.

write an analysis of this

supports functions

5 Discussion and Outlook

To summarize this thesis we want to go back to the initial motivation for this work. The future of analogue neuromorphic systems shows many possibilities that utilize the advantage over exiting simulation environments.

Extensive experiments that would exceed the boundaries of software based simulations regarding experimental time and simulated time are possible and as hardware improves their complexity will do the same. At the same time these analogue systems offer easy manipulation of different parameters during experiments without a decrease in performance. This encourages users to create new and daring test scenarios that would not be realizable for software simulations or inefficient. Even more experimental approaches involve interfaces to virtual environments and running hardware "in-the-loop".

A key feature of neuromorphic systems could be longterm experiments at high speeds that could also run on their own and apply plasticity at experimental speeds. This would give an immense amount of possible applications for the future but also generates new requirements to achieve this.

The PPU may be a key element on the way there as experiments that so not utilize the PPU usually run slower and are less flexible than PPU based experiments. This is due to the performance of vector processing that the PPU offers because of its customizations and the immediate access to the analogue system as both are combined in the HICANN-DLS. As of now the PPU offers fast calculation of simple plasticity rules and thus flexibility when compared to static systems that have predefined setting that are only updated in between experimental runs. Although the PPU can do this for only so much complexity as the hardware is quite limited, it fulfills he requirements for simple optimization, implementing reduced virtual environments that need high latency and managing calibration of the system. It also gives the possibility of running experiments without supervision as the PPU could collect data as well.

All of this needs code that is at the same efficient and favorably of small size. At the same time software should be easy to write in order to appeal for many users and raise interest for these systems. One way to get all of this is working compiler support where the compiler translates simple code into optimized machine code that is specifically created for this target. Creating a stand-alone compiler that supports the PPU's custom architecture could therefore improve development for the PPU noticeably. An equally effective but maybe more complete approach to this is extending the back-end of an existing compiler that also offers different tools for development and multiple front-ends. As this is the case for GCC this thesis dealt with the task of extending an existing back-end that already supported the core architecture of the PPU's nux architecture and implementing support for the custom s2pp vector extension .
This goal could be achieved to some degree as the back-end does support nux' vector

extension but however is still in a testing stage that hopefully will transition into the standard cross-compiler for the PPU in the near future.

For once the back-end needs testing as only this can show if it is reliable and may reveal bugs that were not obvious before. There even exist first test cases that aim for high-level software tests and already test single intrinsics and 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 which tested on the PPU as well as in simulation. This may also find from eventual bugs or unexpected behavior in the architecture itself or in the compiler and overall improve reliability.

The most recent release of the GCC 4.9 release series dates only back to 2016 and though maintenance is officially discontinued [?] there exist newer bugs which were fixed through patches thanks to the active community behind GCC. Therefore it is possible —although unlikely — that internal compiler errors are caused by the original compiler instead of the back-end. It should be therefore considered if moving to a newer release — which is 4.9.4 — made sense to reduce compiler liability to a minimum and focus solely on the back-end. There is also a number of tests for the AltiVec back-end extension that should be taken in consideration when implementing new nux tests.

Besides high-level testing we can also utilize the newly extended back-end for low-level testing of the architecture itself that is not pure assembly coding but utilizes the `asm` support of the back-end. Through this we were able to find undocumented undocumented behavior of the PPU when using conditional instructions which seemed like a bug in the back-end at first..

Since there exists software simulation for nux but only a small number of low-level test cases, the development of more tests should be focused as this leads to a well documented nux architecture which is necessary for reliable high-level usage. This should involve testing random combinations of instructions and operands which could be realized by a random code generator that works hand in hand with a simulation platform. The planned nux simulation on a FPGA would accelerate this development and also encourage continuous testing.

Creating such a comprehensive testing environment would allow to constantly test the reliability of generated code and different optimizations and thereby help establishing GCC with an extended back-end as the preferred compiler for nux.

Another feature that might help push the general usage of a patched cross-compiler is the availability of `gdb` for nux. As debugging PPU software right now is not possible with `gdb` it might be possible in the future but it is likely that more work on the back-end is necessary for this. Right now there have been no tests of `gdb` with the new back-end and other features such as optimization and testing are more important at the moment.

At most, only time will tell if the extended-GCC build will be used regularly and become the standard tool for nux development.

This depends on various factors besides what we already mention. GCC support for the POWER architecture, reimplementation of the GCC back-end and the future development of the nux architecture are also important. As the POWER architecture is well established and first systems with POWER9 under way we can safely assume that the POWER architecture will be supported by GCC for years to come. Also the radical reorientation of GCC by the GCC steering committee might seem unlikely we can still rely on existing older releases as newer versions of GCC rarely offer striking new features and older versions continue to receive patches although there are no longer officially maintained.

There 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 but this build has not been tested yet and only demonstrates the possibility of porting the back-end patch to different GCC releases if it ever became necessary.

Hence the most crucial development is that of the nux architecture. If it is ever decided that the nux is to be completely redesigned this would likely render the current back-end pointless and one must start from scratch when creating compiler support for that architecture. Smaller changes however, i.e. adding a set of logical vector operations, may be supported by adding these to the machine description and creating intrinsics from this as described in **??** and [**?**].

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 extended GCC back-end may be usable for many years and even longer if the back-end is maintained.
Besides getting future compiler support there are many features yet to come for the PPU:

Adding new vector instructions to the instruction set as well as allowing for direct access to the memory of the FPGA that is connected are only two features that could help creating new experiments that were not possible by now and that may also profit from established compiler support. Also should `gdb` support help creating those applications on HICANN-DLS. Having established the prerequisites for code generation for the PPU we may also expect development in this particular area which further facilitates development of new experiments by many users.

Over all the PPU should get ever more capable over time and take care of different tasks like calibration, creating simple virtual environments, configuration of the system and also supervision of experiments.

Ultimately PPUs might be running independently on multiple systems, realizing experiments with large simulated networks or running standalone experiments for users in parallel. Still, the future of the PPU is welded by users, developers and the applications they for this innovative system, and giving them a functional compiler for this system is the tool that might come in handy at times.

Appendix

Acronyms

ADC Analogue Digital Converter

ALU Arithmetic Logic Unit

ASIP Application Specific Instruction set Processor

asm Assembly

CADC Correlation Analogue Digital Converter

CISC Complex Instruction Set Computer

CPU Central Processing Unit

CR Conditional Register

DLS Digital Learning System

DRAM Dynamic RAM

insn instruction

IR Intermediate Representation

FPGA Field Programmable Gate Array

FPR Floating Point Register

FPU Ploating Point Unit

GCC GNU Compiler Collection

GPP General Purpose Processor

GPR General Purpose Register

HICANN High Input Count Neural Network

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

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 Computer

rs6000 RISC system/6000

s2pp synaptic plasiticity processor a.k.a PPU

SIMD Single Input Multiple Data

SPR Special Purpose Register

SRAM Static RAM

VE Vector Extension

VR Vector Register

VSCR Vector Status and Control Register

VRSAVE Vector Save/Restore register

VRF Vector Register File

VMX Vector Media eXtension

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
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)	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_promote	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_sh fxv_sh	d = fxv_sh(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_splat_s16 vec_splat_u16 \\ fxv_splatb	d = fxv_splatb(a)	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_splat_s8 vec_splat_u8 \\ fxv_splath	d = fxv_splath(a)	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
fxv_cmp	fxv_cmp(a)	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
fxv_mtac fxv_mtacfs	fxv_mtac(a)	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
fxv_addactacm fxv_addactacf	fxv_addactac(a)	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
fxv_addacm fxv_addacfs	fxv_addac(a)	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
fxv_mam fxv_mafs	fxv_ma(a)	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
fxv_matacm fxv_matacfs	fxv_matac(a)	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
fxv_multacm fxv_multacfs	fxv_multac(a)	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
fxv_addtac	fxv_addtac(a)	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
fxv_pckbu fxv_pckbl	d = fxv_pckbu(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
fxv_upckbl fxv_upckbr	d = fxv_upckbl(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

Table 1: List of all implemented built-ins and how they are used

List of nux Intrinsics

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` shifted 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`		halts execution until the previous instructions are finished EDITHIS
`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

caption and reference to PPC book and asm website correct the table

Obviously the 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 `asm` which behave like pointers:

`@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

Additionally there exist markers which are intended for debugging:

`.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`

all these tables in the appendix

Notes

Bibliography

[1] Altivec manual, chapter.

[2] Gcc wiki, accessed 2017.03.26, 2013.

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

[4] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 13.9 RTL Expressions for Arithmetic, 2017.

[5] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 6.45.2 Extended Asm - Assembler Instructions with C Expression Operands, 2017.

[6] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 16.19 Instruction Attributes, 2017.

[7] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 16.8 Operand Constraints, 2017.

[8] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 16.15 Defining RTL Sequences for Code Generation, 2017.

[9] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 16.2 Everything about Instruction Patterns, 2017.

[10] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 16.16 Defining How to Split Instructions, 2017.

[11] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 17.9.11 Function Entry and Exit, 2017.

[12] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 16.23 Iterators, 2017.

[13] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 13.6 Machine Modes, 2017.

[14] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 8 Option specification files, 2017.

[15] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 16.7 Predicates, 2017.

[16] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 13.8 Registers and Memory, 2017.

[17] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 13.15 Side Effect Expressions, 2017.

[18] *GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 16.9 Standard Pattern Names For Generation, 2017.

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

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

[21] Heimbrecht, A., Internship report — altivec intrinsics, 2017.

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

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

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 2, 2017

....
(signature)