

RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG



Arthur Heimbrecht

---

Bachelor Thesis

HD-KIP-TODOTODOTODO

KIRCHHOFF-INSTITUT FÜR PHYSIK

---



Department of Physics and Astronomy  
University of Heidelberg

**Bachelor Thesis**  
in Physics  
submitted by  
**Arthur Heimbrecht**  
born in Speyer

**TODO 2123**



# 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 “mux” 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 mux, in order to extend the already implemented set of functions according to the user’s needs.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Methods</b>	<b>3</b>
2.1	Basic Processor Architecture . . . . .	3
2.1.1	Co-processors and Extensions . . . . .	6
	AltiVec Vector Extension . . . . .	7
2.2	Hardware Implementation . . . . .	8
2.2.1	Basics of Neural Networks . . . . .	8
2.2.2	Implementation in Hicann-DLS . . . . .	9
2.2.3	The Plasticity Processing Unit . . . . .	10
2.3	Basic Compiler Structure . . . . .	12
2.3.1	Back-End and Code Generation . . . . .	14
2.3.2	Assembly Basics . . . . .	16
2.3.3	Intrinsics . . . . .	20
2.3.4	GNU Compiler Collection . . . . .	20
	Register Transfer Language Basics . . . . .	22
<b>3</b>	<b>Extending the GCC Back-End</b>	<b>25</b>
3.1	Registers . . . . .	29
3.2	Prologue and Epilogue . . . . .	33
3.3	Reload . . . . .	34
3.4	Built-ins and Machine Instructions . . . . .	35
<b>4</b>	<b>Results and Applications</b>	<b>40</b>
<b>5</b>	<b>Discussion and Outlook</b>	<b>42</b>
	<b>Appendix</b>	<b>43</b>
	<b>Bibliography</b>	<b>48</b>



# 1 Introduction

As neuromorphic computing becomes ever more popular, many applications for such systems emerge and make use of its hardware's advantages over traditional computers. But to do so one must be familiar with a system as it often differs in many ways from common system architectures. As a result programming for such systems can feel odd to new users as they need to abandon some familiar techniques and acquire new skills. This understandably is a hurdle for many users and also initially takes a significant amount of time. The more a system's programming abandons common elements of programming which users are accustomed to the more this can become a problem. Not only do fewer users take the initiative of writing for such systems but also can code easily get confusing, hard to debug and even ineffective.

An example of this is the current state of programming for the plasticity processing unit of the Hicann-DLS. It is responsible for applying learning rules to neural systems on the Hicann-DLS and resembles a common processor which was extended for this cause. Despite basic programming still being the same for this processor it differs for creating the mentioned learning rules. Although these are still programmed in C a user needs to use a set of functions and predefined variables while at the heart of this is basic assembly programming. This was done to include users unskilled in assembly who want to write programs for the PPU. This lead to PPU programs having a distinct look and feel that is only in some regards similar to C but feels like being pushed back to the origins of computing; for example reading out the value of a variable need unhandy workarounds and accessing memory is a repetitive set of program lines. The main reason we feel so reluctant when it comes to outdated programming are compilers.

Since the early stages of computing, compilers have developed towards a standard tool in everyday programming but at the same time became more of a black box that transforms a program into an executable file. For this reason it may be difficult for some users to abandon such convenience and go back to low-level programming.

But luckily the PPU is not completely without compiler support as basic computing is support but new features of the PPU are only usable on a low level. This could still cause major inconvenience for users as these features are necessary to implement learning rules for synapses which are themselves elementary to neuromorphic programming. As of now users are forced under these circumstances to mix high-level programming with very low-level programming. Of course this possibly leads to a series of problems as users have to adapt to this atypical style of programming. At worst this can even lead to ineffective programs — although performance is important for neuromorphic programming and the main advantage of the PPU — and demand an unreasonable amount of time and work to achieve simple results.

## 1 Introduction

Although this makes it obvious that compiler support is desirable the reason for the PPU not having compiler support is the custom nature of the processor architecture, which was developed solely for the BrainScaleS system and the HICANN-DLS. It could be classified as an application specific instruction set architecture (ASIP) which is an intermediate form between general purpose processors and application specific integrated circuits (ASIC). Such architectures offer a partly customized instruction set architecture (ISA) that is optimized for certain applications.

Hence the only way to support complete high-level programming is adding compiler support for the PPU's ISA.

Not only would this permit full C-style high-level programming when working on the PPU but could also include code optimization and code debugging. We therefore aim to achieve “full” compiler support of the PPU's hardware and make programming easier which ultimately should open the platform for new users, simplify programming for existing users and possibly accelerate performance on the PPU. At some point compiler support could also lead to automatic code generation which then could allow for implementation of high-level languages. Ultimately the goal should be that users do not need to code for the PPU itself but create plasticity rules in existing program environments that translate this “meta-code” into PPU programs. Obviously this calls for optimization of PPU code and although this could be done by developers specifically for the PPU it is far easier and likely more efficient to utilize existing optimization techniques that are built into virtually every compiler.

This thesis will focus on the way to achieve mentioned compiler support and briefly explain the process itself. As fundamental knowledge of both processors and compilers is needed along the way it will start with a very basic introduction to both topics and go into detail for the processor and compiler that were used later on. 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 and the result as well as test cases of this presented. The thesis will conclude in a resume and give an outlook to the future applications and development of the compiler.

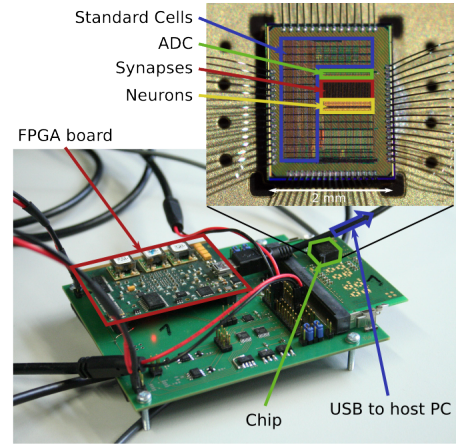


Figure 1.1: set-up of a HICANN-DLS test system

## 2 Methods

### 2.1 Basic Processor Architecture

Almost all contemporary processors used are built using the so called von-Neumann architecture . Although the main goal of this group is to provide an alternative analogue architecture that is inspired by nature, there are advantages to the classic model of processors which come to use for some applications. 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 setting up different parts of an experiment. We will now shortly touch the basics of such processors and explain common terms.

add reference

cite friedmann dissertation

In general a microprocessor can be seen as a combination of two units which are an operational section and a control section. The control logic section is responsible for fetching instructions and operands, interpreting them and controlling their execution as well as reading and writing to the main memory or other buses. The operational section on the other side saves operands and results as long as they are needed and performs any logic or arithmetic operation on these as instructed by the control logic section. Prominent parts of the operational section are the arithmetic logic unit (ALU) and the register file.

The register file can be seen as short-term memory of the processor. It consists of several elements, called registers, that have the same size which is determined by the architecture; a 32-bit architecture has 32-bit wide registers. Typically the number of registers varies for different architectures and also their purpose. Common purposes of registers are:

**general-purpose register GPR** These registers can be used for virtually anything and in most cases carry values that are soon to be used by the ALU. Few of these registers can be reserved as stack pointers. Most registers on a processor are typically GPRs.

**link register LR** This register marks the jump point of function calls. This means that after a function completes, the program jumps to the address in the link register.

**compare register CR** This register's value is set by an instruction that compares one or two values in GPRs. Its value can determine for some instructions if they are to be executed or not.

Non-general-purpose registers are also called special-purpose registers SPRs.

The ALU normally uses the values which are stored in the register file to perform the aforementioned logic or arithmetic operations and saves the result there as well. In case

## 2 Methods

of more complex arithmetics some architectures also have an accumulator that is part of the ALU or sits next to it. Intermediate results then are stored there because access is faster for the accumulator compared to registers. In general it is good to know:

$$\text{speed(accumulator)} > \text{speed(register)} \gg \text{speed(memory)} \quad (2.1)$$

As speed is always important in computing we therefore want to use registers as much as possible and only write results to the memory when not enough registers are available for the current task or the task has finished. Registers such as the accumulator can also either be accessible directly or are only accessible to subsections such as the ALU. This is different for every processor architecture and depends on a multitude of factors:

- available chip space
- maximum clock frequency
- instruction set complexity
- available time and money for the design
- energy consumption

These factors influence one another as a complex instruction set allows for complicated arithmetic operations with many operands to be performed in few clock cycles but this often results in the maximum clock frequency being lower. During a single clock cycle a chip usually performs one so called micro instruction which is part of a machine instruction. An example for an add instruction (`result = add(a,b)`) would be:

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

complexity > lower clock  
freq

inhaltlicher zusammen-  
hang

For different and more complex machine instructions the amount of micro instructions can be much higher, but this basically sets the minimum amount of micro instructions for any machine instruction. Ergo the faster the clock frequency the faster micro instructions can be executed the faster the processor. But also the more complex the instruction set is the fewer machine instructions are needed overall the faster the processor. As mentioned above this results in a trade-off between clock frequency and instruction set complexity. The instruction set includes all available instructions for an ALU thus the ALU gets more complex and needs more space as the instruction set gets more complicated.

Because of this, one usually differs between two kinds of processors:

- Complex Instruction Set Computer CISC
- Reduced Instruction Set Computer RISC

The latter usually is reduced to such simple instructions as `add` or `sub` and connects them to create more complex instructions. Since the PPU is a RISC architecture we focus on RISC's key values. Simple instructions are similar to micro instructions which were mentioned earlier, but every simple instruction has the same 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 micro instruction 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 as that many machine instructions can be initiated in the same time it takes to complete a single machine instruction. It must be noted though that the processor has to detect hazards which are data dependencies between instructions where one instruction needs the result of another. Such instructions usually are postponed to a delay-slot and other instructions that do not cause hazards are executed instead. This results in reordering of instructions on a processor level. Also it takes several cycles for memory instructions to load or store data this effectively stalls the processor until the memory instruction has finished. Therefore RISCs try to avoid memory access as much as possible and use registers instead. Luckily a normal RISC architecture provides more registers as the ALU needs less space due to reduced complexity and also can be operated at higher clock frequencies, therefore it is perfect for simple processors that only need to do simple arithmetic as fast as possible.

Next we take a closer look at memory. Normally the memory of a von-Neumann machine contains both, the program and data (this is contrary Harvard architectures). The program here describes a list of instructions that are part of the instruction set. Each instruction itself is represented as a sequence of bits in memory that resemble the following.

graphic of opcode

The first part which is called an opcode is simply a number that stands for an operation performed by the ALU. The ALU reads this number and performs the necessary steps. Typically this part is about 8 bits long and has an alias string such as `add` that is called a mnemonic. The second part is the result which is of the same type as the third and forth part. These are the argument addresses or operands of an operation and can either be a memory address or a register number as both are valid operands. Many RISC architectures have an instruction set that consists exclusively of 3 operand instructions. Any instructions that seem to have less than three operands are normally mapped on instructions that have three operands. It is quite common to use more complex instructions for relatively simple instruction as this reduces the number of opcodes. An example would be moving the contents of register 1 to register 2. This usually maps to an or comparison between register 1 and a register that is all zeros where the result (the same as register 1) is saved in register 2.

It is important to note that in most RISC architectures if we want a memory address as operand, this is done indirectly. A memory address can not be an operand on its own but is loaded into a different register and a different register gets to hold the data from the memory. This is called a load instruction and its counter part would be a store

## 2 Methods

instruction. Architectures that work like that are called load/store architectures.

This means also that the amount of accessible memory is typically limited by the width of a single register. Memory is often seen as blocks and with addresses. Because the smallest amount of information which we are interested in is a byte, each address is equivalent to one byte in memory. Therefore the maximum amount of memory that can be used is:

$$2^n \text{byte} \xrightarrow{n = 32 \text{ bit}} 2^{32} \text{byte} \approx 4 * 10^9 \text{byte} = 4GB \quad (2.2)$$

Normally though it is not the processor itself that keeps track of the memory. This is usually done by a memory management unit (MMU). It handles all memory access of the processor as it can provide a set of virtual memory addresses which itself then transforms into physical addresses. Most modern MMUs also incorporate a cache that stores memory operations while others are handled and detects dependencies within this cache which it can resolve. This results in faster transfer of data as two or more instructions access the same memory which then is handled in the cache. Not all MMUs support this though and this might lead to certain problems when handling memory. If instructions are reordered due to pipelining and dependencies on the same memory address are not detected, an instruction may write to the memory before a different one could load the previous value it needed. For this reason exist memory barriers. A memory barrier is an instruction that is equal to a guard in code that waits until all load and store instructions issued to the MMU are finished and then allows the code to proceed. It therefore splits the code into instructions issued before the memory barrier and issued after the memory barrier. Even with reordering this prohibits any instruction to be executed on the wrong side of the barrier and thereby ensures conflicting memory instructions to not interfere with one another.

Memory can be split into two popular types which are static random access memory (SRAM) and dynamic random access memory (DRAM). They differ in how bits are set on each RAM. SRAM uses Flip Flops to switch transistors that indicate which bit is set, while DRAM uses capacities that are charged to do so.

We already introduced many parts of a processor which need to be connected somehow. Connections between these parts are called buses and also have a width measured in bytes. 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 buses of such width need much space. Therefore some architectures use narrower buses with fewer bits than a register and use two instructions to transfer the contents of a full register. 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 ones this results in less performance loss than initially expected.

### 2.1.1 Co-processors and Extensions

RISC architectures sometimes need so called co-processors for instructions that are not included in the instruction set but are often enough needed. An example would be multiplication which would need many cycles when split in `add` instructions but as part



a co-processor can be performed in just a few cycles. In such a case the control section recognizes the `mult` instruction and passes it to the co-processor and later on fetches the result.

This can be extended to whole units such as 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 have parallel access to the memory.

Another kind of extensions 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 . But also early supercomputers such as the Cray-1 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 parallel architecture or more easily through pipelining the instruction on one vector over its elements. The latter one makes sense since there are typically no dependencies between single elements in the same vector. Nowadays many of the common architectures support vector processing. A few examples of these are:

reference

reference

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

As mentioned these were mostly intended for speeding up tasks like adjusting the contrast of an image. There is also the possibility to vectorize loops in programming if there are no dependencies between loop cycles.

### AltiVec Vector Extension

In our case we take a special interest in the AltiVec vector extension which 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 the processor with a single-precision floating point and integer SIMD instruction set. The vector register file includes 32 vector registers are each 128-bit wide. AV: These vector registers can either hold sixteen 8-bit `chars`, eight 16-bit `shorts` or four 32-bit `ints` or single precision `floats`, each signed and unsigned. Single elements of these vectors can only be accessed through memory because there is no instruction that combines scalar register with vector registers. Except for one type of instruction that "splats" the value of a scalar register into all elements of the vector register. The reason we take such an interest in this vector extension is that it resembles most characteristics of the PPU's vector extension and is already implemented in the PowerPC back-end of GCC. There a few differences though:

## 2 Methods

First the PPU's VE uses a conditional register (CR) to perform instructions only on those elements of a vector register, that meet the condition in the corresponding part of the CR, which is specified by the user, while the AltiVec VE utilizes the CR which included in the PowerPC architecture. This results in not allowing 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 on its own though, which are the VCSR and VRSAVE registers. The Vector Status and Control Register (VSCR) is responsible for detecting saturation in vector operations and decides which floating point mode is used. The Vector Save/Restore Register (VRSAVE) assists applications and operation systems by indicating for each VR if it is currently used by a process and thus must be restored in case of an interrupt.

Both of these register are not available in the PPU's VE but would likely not be needed for simple arithmetic tasks as the PPU is meant to perform.

## 2.2 Hardware Implementation

As this thesis mainly focuses on a processor that is an essential part of the HICANN-DLS (high input count analogue neural network Digital Learning System) we will focus first on the Hicann-DLS as a whole and then look into the PPU in detail which includes many of the topics of the previous section.

### 2.2.1 Basics of Neural Networks

Neural networks build the main application of the Hicann DLS system. This short chapter is meant to give an overview over neural networks and synaptic weights.

On a very abstract level neurons in the brain resemble nodes of a network. As in a network neurons are interconnected through dendrites, synapses and axons which can be of different strength. Also we assume that a neuron is either spiking, meaning it is activated and sends this information to connected neurons or resting meaning it is not activated. In case a neuron is spiking, it send this information through its axon to other other neurons that are connected to the axon by synapses. These synapses can work quite differently but have in common that there is a certain weight associated to them, which we will call synaptic weight. This is equal to a gain with which the signal is either amplified or attenuated. 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 then sends a signal itself to other neurons.

put the following part in the next section

With all these physiological parts there are only two important parts we need to take a look at in order to copy the function of a neural network: the neuron and the synapses.

add something here

If two neurons are actually not meant to be connected, the spike's address and the SRAM address of the synapse do not match and thus the spike is ignored by the synapse. Now if we display the all neurons inputs and outputs in a 2D plain we get an array of synapses, which is equivalent to a weight matrix.

explain this further

### 2.2.2 Implementation in Hicann-DLS

The Hicann-DLS system tries to implement this structure as close to reality as possible in order to simulate physiological processes in such networks. At its core it therefore has a so called "synaptic array" that connects 32 neurons which are located on a single chip to 64 different pre-synaptic inputs. Each neuron's post-synaptic input is aligned along one axis of an array while the 64 outputs of different neurons are on a rectangular axis. This gives a 2D array of 2048 synapses in total. An FPGA connects the 64 pre-synaptic inputs to various neurons in the system while it can also connect the neurons of the same chip to the pre-synaptic inputs. Along these input lines the signal reaches all synapses where it is processed individually. For this to be possible each pre-synaptic neuron has a 6 bit SRAM address while the synapse itself has a 6 bit SRAM address as well, which can be changed from outside. Each synapse then compares the addresses of the pre-synaptic neuron it is connected to to its own and if they match sends out a signal to other circuits that need this information. Also in this case each synapse multiplies the signal it receives with its weight and sends the result to the post-synaptic input of a neuron. All signals sent by synapses to an input are integrated along the line to a resulting input signal which finally reaches a neuron. Inside the neurons the individual input signal is evaluated in regard to a threshold and other parameters which decide whether the neuron is spiking or not. If the neuron is spiking it sends out an output signal to the FPGA which is responsible for spike routing. The output signal of each neuron is also sent to an analogue digital converter (ADC) in order to analyze the data in digital form. All of this is done continuously and may not follow discrete time steps.

The Hicann-DLS system 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 and thus can read and write synaptic weights.

The synapses in the synapse array are realized as small repetitive circuits that contain 8 bits of information each. The weights themselves are 6 bit large and always right aligned. The most significant bit of each weight has a value of  $2^{-1}$  with subsequent bits having half the value of the previous bit. The spare two bits at the beginning are used for calibration. The synapse array can also be used in 16 bit mode for higher accuracy. This combines two synapses to a single virtual synapse with 12 bit weights and 4 bits for calibration.

The whole chip itself is also connected to a field programmable gate array (FPGA) that is able to read and write to the synaptic values as well as the memory of the PPU.

### 2.2.3 The Plasticity Processing Unit

The PPU, which was designed by Simon Friedmann *Friedmann et al.* (2016), is a custom processor in this system, that is based on the Power Instruction Set Architecture (PowerISA), which was developed by IBM since 1990. Specifically the PPU uses POWER7 which is a successor of the original POWER architecture and was released in 2010 and runs at 100 MHz clock frequency. It is a 32-bit architecture therefore registers are 32 bit wide which is also the word size.

It was developed to handle plasticity and as such apply different learning 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  $10^4$  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 and record data.

The PPU is accompanied by 16 kiB of memory as well as 4 kiB of instruction cache which together is called the plasticity sub-system. The PPU's distinct feature is its special-function unit or vector extension (VE) that allows for Single Input Multiple Data (SIMD) operations. The VE is only weakly coupled to the general purpose part (GPP) of the PPU and mostly both parts can operate in parallel while interaction is highly limited. All vector instructions that are intended for the VE must first pass the GPP though, which detects vector instructions and passes them to the VE as it is usual for most processor extensions. These instructions then go into a queue that holds all vector instructions where they are fetched from in order. Going from there the instructions 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. Therefore it is also possible during the process of accessing a vector on memory to perform some arithmetic operations on a different vector. This allows for faster processing speeds as pipelining for each instruction is also supported. The limiting factor for this though remains the vectors register file's single port for reading and writing.

The main limiting factor in processing speed is the memory access. Both, the GPP and the VE, share the same MMU and thus any access of the GPP to vectors in memory must be handled with care as the GPP and VE are not synchronized. The MMU is very simple as it does not cache memory instructions and also has matching virtual and physical addresses. For this reason one must be aware of the `sync` instruction that is a memory barrier and stops the GPP from executing instruction until all memory requests of GPP and VE are handled. This 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. `sync` is a standard instruction of the PowerISA and further information can be found [here](#).

reference

The PPU is also able to read out spiking times and additional information through a bus which is accessible through the memory interface. It uses the first bits of a memory address which are available because the memory is only 16 kiB large which is equivalent to 16-bit addresses and registers are 32-bit wide. Thus it only needs a pointer to such a memory address to read spiking rates during an experiment. Besides the VE and the

GPP, the memory bus also provides access to the FPGA in order to allow for external access to the system. This is needed for writing programs into the memory as well as getting results during or after experiments. This also allows for communication during runtime of the PPU.

check this

The VE was added due to the need for fast handling and writing of synaptic weights into the array of synaptic values on the HICANN. Parallelizing this gives up to an 16x increase in performance. Hence the vector unit was equipped with an extra bus that connects to the mentioned synapse array. The synapse array though is also 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 more comfortable and gives more structure to the program, As mentioned before do GPP and VE share a memory bus but vector memory instructions need to pass the VE first which leads to the delay that makes inserting a heavy weight memory barrier or “syncing” necessary at times.

Specifically the vector extension allows for either use of 8 element vectors with elements being halfword (1 halfword = 2 bytes) sized or 16 element vectors with each element byte sized. Thus every vector is 16 bytes or 128 Bits long. This is also the size of each vector register that is available, which are 32 in total, in contrast to 32 general purpose registers with 32 bit each. The VE also features a vector accumulator of 128 bit which can be read and written by hand and a vector condition register which holds 3 bits for each half byte of the vector, making 96 bit in total, that determine which condition applies.

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 like is not going to in the future. An overview of all opcodes is provided by *Friedmann* (2016), 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 a maximum that is equivalent to one when we see the MSB as  $2^{-1}$  and the minimum is  $2^{-15}$  for halfword and  $2^{-7}$  for byte instructions.

check with ahartel on this

**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.

When using these instruction one must always keep in mind that the weights of the synapses only consist of the latter 6 or 12 bits which are in a vector register and are right aligned. As a user still wants the full functionality and also as much accuracy as possible, a vector is typically shifted left when reading from the synapse array to align

the MSB to the very left and thus right shifted when stored in the synapse array. We will keep this big-endian description throughout this thesis.

applications of the PPU today like in-the-loop experiments and controlling

### 2.3 Basic Compiler Structure

At its core every compiler translates a source-language into a target-language, 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. But compilers do not exist as long as computers do and their development played a big role in making computers such an important part of everyday life as they are today. 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 as it is run, a compiler takes the time to read the source-language file completely (often several times) and only then creates the executable files which are run after the process has finished. The advantages of this are simple: While a compiler takes some time at first until the program can be run, 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 several 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. Though a compiler is not solely involved into the processing of a programming language towards an executable program. Figure 2.1 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. After the compiler is finished with its job the **assembler** takes over and 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 for different source files with standard library functions that are also already compiled and other resources. The result is a single file that is directly 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.

But since we are more interested in compilers than other components, we will take a better look at the compiler itself. Figure 2.1 shows the common separation of a compiler into front-end, back-end and the 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

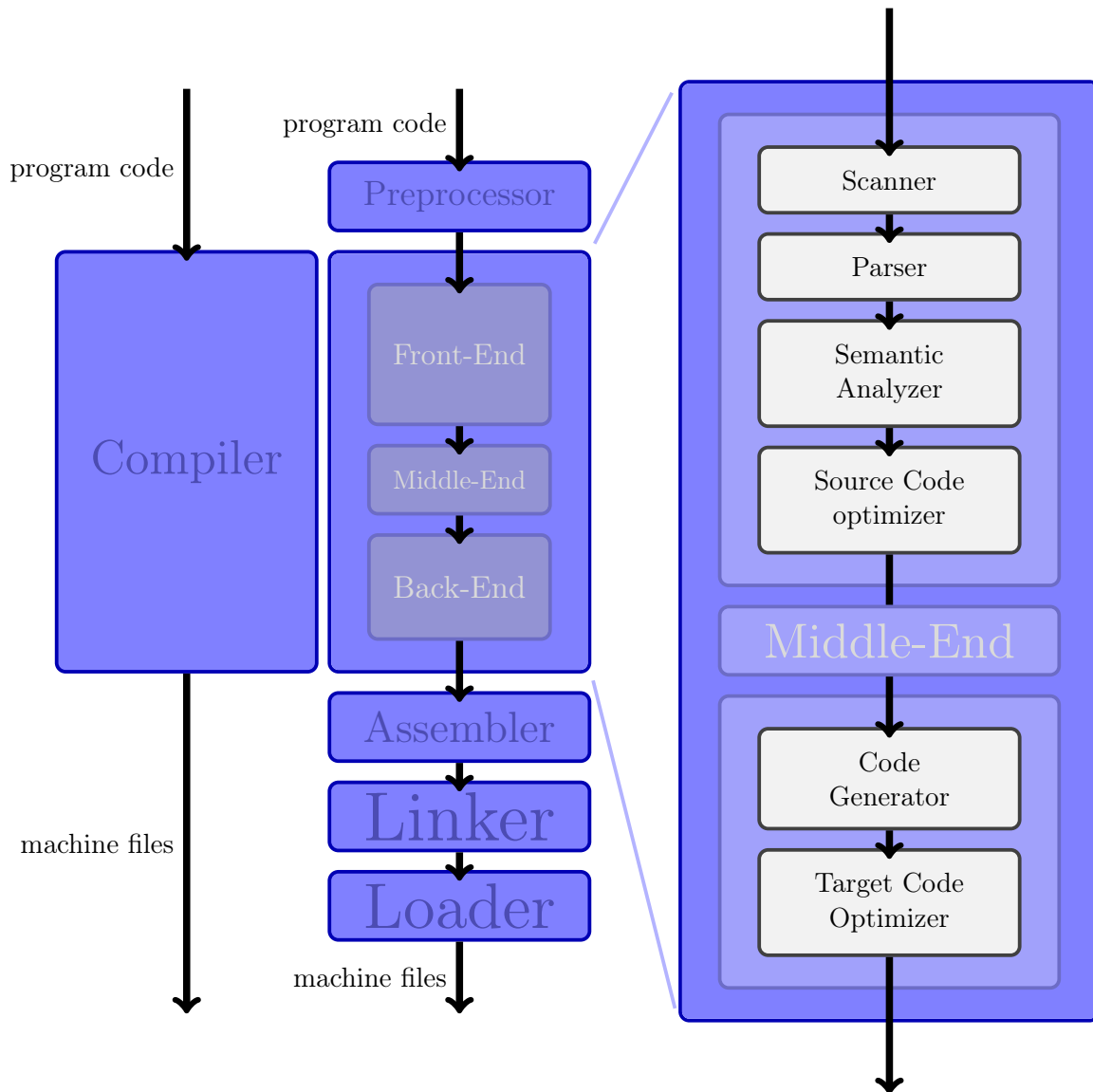


Figure 2.1: Schematic overview of different compiler stages.

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 2.1. There is no middle-end included into this scheme as it is not a mandatory part of the compiler and would only be responsible for optimizations. But we will take a short look at the other phases: First the source code is fed into the **scanner** that performs lexical analysis, which is combining sequences of characters to symbols of something 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 the first shot at optimizing the code. Typically only light optimization is possible at this point such as pre-computing simple arithmetic instructions and different kinds of optimization exist. After the source code optimizer is done the syntax tree is converted to intermediate representation 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 and strings and makes them available globally 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 `nops` 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 `syncs` 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 the beginning of a procedure and start computing on registers that already have a value while the others are still loaded. This is done vice versa at the end of a procedure for storing the results in memory. This process 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.

## 2 Methods

At last the compiler handles register allocation which also includes memory handling. Typically the previous processes expect an ideal target machine which provides an endless amount of registers. As in reality the processor only has  $k$  registers the register allocator reduces the number of “virtual registers” or “pseudoregisters” that are requested to the available number of “hard registers”  $k$ . For this to be possible the compiler decides whether a value can live throughout a procedure in a register or must be placed 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 the 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 ineffective 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

Assembly (`asm`) was mentioned a few times in this thesis already and we need to know at least a few basic assembly instructions that will help us later on to understand resulting machine code. Assembly is usually the lowest level of representation of a program that

still is human-readable. Assembly code is basically equivalent to machine code and therefore by many seen as such though assembly code needs to be assembled by the assembler first. Assembly instructions all follow a certain scheme which is:

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

For RISC architectures instructions typically consist of 3 operands because operations are usually between registers only (except for load/store a.k.a. memory instructions). The mnemonic in most cases is 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. I.e. a memory address can be saved to the register and an operation uses the value at the memory location which the register refers to.

## 2 Methods

mnemonic	operands	description
<code>add</code>	<code>RT, RA, RB</code>	<b>add</b> <code>RB</code> to <code>RA</code> and store the result in <code>RT</code>
<code>addi</code>	<code>RT, RA, SI</code>	<b>add</b> <code>SI</code> to <code>RA</code> and store the result in <code>RT</code>
<code>addis</code>	<code>RT, RA, SI</code>	<b>add</b> <code>SI</code> shifted left by 16 bit to <code>RA</code> and store the result in <code>RT</code>
<code>and</code>	<code>RA, RS, RB</code>	<code>RS</code> and <code>RB</code> are <b>anded</b> and the result is stored in <code>RT</code>
<code>b</code>	<code>target_addr</code>	<b>branch</b> to the code at <code>target_addr</code>
<code>ble</code>	<code>BF, target_addr</code>	<b>branch</b> to the code at <code>target_addr</code> if <code>BF</code> is less or equal
<code>blr</code>		<b>branch</b> to the code at address in the linker register
<code>cmp</code>	<code>BF, L, RA, RB</code>	<code>RA</code> and <code>RB</code> are <b>compared</b> and the result ( <code>gt,lt,eq</code> ) is stored in <code>BF</code> , <code>L</code> depicts if 32-bit or 64-bit are compared
<code>cmplwi</code>	<code>BF, RA, SI</code>	<code>RA</code> <b>compared</b> logically wordwise with immediate <code>SI</code> and the result is stored in <code>BF</code>
<code>and</code>	<code>RA, RS, RB</code>	<code>RS</code> and <code>RB</code> are <b>anded</b> and the result is stored in <code>RT</code>
<code>eieio</code>		<code>RS</code> and <code>RB</code> are <b>anded</b> and the result is stored in <code>RT</code> EDITTHIS
<code>isync</code>		<code>RS</code> and <code>RB</code> are <b>anded</b> and the result is stored in <code>RT</code> EDITTHIS
<code>la</code>	<code>RT, D(RA)</code>	load aggregate <code>D + RA</code> into <code>RT</code>
<code>li</code>	<code>RT, SI</code>	load immediate value <code>SI</code> into <code>RT</code>
<code>lis</code>	<code>RT, SI</code>	load immediate value <code>SI</code> shifted left by 16 bit into <code>RT</code>
<code>lbz</code>	<code>RT, D(RA)</code>	load byte at address <code>D+RA</code> into <code>RT</code> , fill the other bits with zeros
<code>lwz</code>	<code>RT, D(RA)</code>	load word at address <code>D+RA</code> into <code>RT</code> , fill the other bits with zeros
<code>mflr</code>	<code>RT</code>	<b>move</b> from linker register to <code>RT</code>
<code>mr</code>	<code>RT, RA</code>	<b>move</b> register <code>RA</code> to <code>RT</code>
<code>nop</code>		halts execution until the previous instructions are finished EDITTHIS
<code>rlwinm</code>	<code>RA, RS, SH, MB, ME</code>	rotate left word in <code>RS</code> by immediate <code>SH</code> bits then <b>and</b> with <b>mask</b> which is 1 from <code>MB+32</code> to <code>ME+32</code> and 0 else, store to <code>RA</code>
<code>stw</code>	<code>RS, D(RA)</code>	<b>store</b> word from <code>RS</code> to address <code>D+RA</code>
<code>stwu</code>	<code>RS, D(RA)</code>	<b>store</b> word from <code>RS</code> to address <code>D+RA</code> and <b>update</b> <code>RA</code> to <code>D+RA</code>
<code>sync</code>		halts execution until the memory controller is finished EDITTHIS

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

<code>b</code>	stands for byte and references the size of the operand
<code>h</code>	stands for halfword and references the size of the operand
<code>w</code>	stands for word and references the size of the operand
<code>s</code>	indicates that one of the operands is shifted
<code>g, ge, l, le, e</code>	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:

<code>@l(C)</code>	is equivalent to the lower order 16 bits of <code>C</code> in the symbol table
<code>@ha(C)</code>	is equivalent to the higher order 16 bits of <code>C</code> in the symbol table and minds the sign extension

Additionally there exist markers which are intended for debugging:

<code>.loc # # #</code>	marks a line of code (file, line, column) in the source file
<code>.LVL</code>	is a local label which can be discarded
<code>.LFB</code>	marks the begin of a function
<code>.LFE</code>	marks the end of a function
<code>.LC0</code>	is a constant of the literals table at position 0

all these tables in the appendix

We want to spend just a little more time on assembly as it is useful to know how to program in assembly while using C. This is done by the following scheme:

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

## 2 Methods

The line of 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.

### 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. *von Hagen* (2006) 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 GCC (2017a), 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.

## 2 Methods

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. GCC (2013) 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 the 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) and has several properties like a name, an RTL template, a condition template, an output template and attributes. GCC (2017g) The name in this case is `add<mode>3` (3 for three operands) where `<mode>` is to be replaced by a set of values that describe a mode. GCC (2017p) 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. GCC (2017k) 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. GCC (2017j) 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. GCC (2017o) 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. GCC (2017m) 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. GCC (2017e) 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. GCC (2017b)

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 and must render `true` in order to allow the matching RTL pattern to be applied. In this case the condition is also depending on the mode iterator which substitutes `<VI_unit>` for equivalent code to that of the next `define_insn` with a matching mode. GCC (2017g)

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. GCC (2017g)

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. GCC (2017d)

## 2 Methods

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. GCC (2017n)

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 es a set 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** this is the machine description of the back-end in general and contains the insn definitions for all scalar functions

**rs6000.h** is a header file which contains all macros and declaration of registers

**rs6000.c** is the source file which implements all functions that are needed for generation of the back-end and other accompanying functions

**rs6000.opt** lists the options and flags which can be set for the target

**rs6000-builtins.def** contains the definitions of built-ins/intrinsics

**rs6000-cpus.def** lists subtargets that belong to the rs6000 family

**rs6000-c.c** links built-ins to 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 string/int equivalent to an enumeration which lists CPU types

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

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

**s2pp.md** is a new file that is the machine description of the nux' VE and contains insn definitions for the VE

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

**constraints.md** contains definitions of the constraints that are used

**predicates.md** contains definitions of the predicates that are used

**vector.md** declares insn definitions that apply to vector types in general

### 3 Extending the GCC Back-End

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

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

Before we start it should be emphasized where inspiration was taken from in order to take the right steps when extending the back-end. There exists no specific documentation on the rs6000 back-end whatsoever and most information on the ways it works is available through comments in code only. There is however an existing vector extension which is included in the back-end and is quite similar to the vector extension we want to add. The existing vector extension is the AltiVec Vector extension, we described earlier in this thesis. The source code thus often shows strong similarities between AltiVec and s2pp functions. Still handling AltiVec and s2pp vectors is separated throughout the back-end since mixing these and reusing function can both get messy and dangerous since at times one vector extension must be distinguished from another. Also there do exist enough differences between these two VEs that combining functions and having separate ones would not make a big difference in the end.

We will start with 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 variable 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 be simulated by software.

**-mno-relocatable**

states that the program code has a fixed memory address that may not be altered.

To do so we first create the flag **-ms2pp** along with an option mask. This should activate the vector extension and everything associated with it. In **rs6000.opt** and we simply need to add the lines:

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

`ms2pp` is the name of the target flag and the next lines defines which macros shall be defined. `Target` means that the option is target specific, `Report` means that the option is to be printed when `-fverbose-asm` is activated. `Mask(S2PP)` initializes a bitmask that is available through `OPTION_MASK_S2PP` which is attached to `rs6000_isa_flags` specified by `var` and in parallel specifies a macro `TARGET_S2PP` that is set to 1. GCC (2017l) When we are finished with this we need to specify `#define MASK_S2PP OPTION_MASK_S2PP` which is done in `rs6000.h` as `MASK_` is the standard.

Now we add the processor type which uses this flag as a standard. There exist several lists that contain available targets and we need to add the `nux` to these. First we add the `asm` flag that tells the assembler which system architecture is used. As the `nux` is based on Power7 and the VE does not influence `asm` for the GPP we want `nux` to behave like Power7. To do so we add `%(mcpu=nux: %(asm_cpu_power7))` to `ASM_CPU_SPEC` and simultaneously add `{ "nux", "%(asm_cpu_power7)" }` to `static const struct asm_name asm_names[]` as these arrays must be kept identical. This will induce the assembler flag to be set to `-mpower7`.

Now we want to achieve something similar for the preceding phases of the back-end as as the description of flags above is mandatory to running `nux`. First we will create an entry in `rs6000-cpus.def` which sets the flags accordingly:

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

will define the processor as a Power7 type and add the masks for soft-float, strict-align and no-relocatable(negative of relocatable) as well as the new `s2pp` mask. Setting a mask is the low-level equivalent to adding a flag as every flag induces a mask, which we saw earlier, and thus this will fit our cause. Calling `RS6000_CPU` will basically generate an entry to the enumeration `rs6000_cpu_opt_value` which must be linked to a string following `-mcpu=`. This is done in `rs6000-tables.opt` where a string, in our case `nux`, is associated with the `n-1`-th call of `RS6000_CPU` through:

```
EnumValue
Enum(rs6000_cpu_opt_value) String(nux) Value(n)
```

One must basically count the calls of `RS6000_CPU` to get `n`.

As we are still missing the `-msdata=none` flag, we will take care of this now. Because `-msdata` is not an ordinary target flag, we need to initialize it differently. Depending on the string following `-msdata=` the back-end decides which mask is referred to when calling `rs6000_sdata`. This is decided in `sysv4.h` through comparing with different valid strings. Since we were only able to set option masks before we need a little work-around here that helps us setting the value of `rs6000_sdata`. If `-msdata` was not defined as it is the case with `-mcpu=nux` the compiler will check through `else ifs` what to do for `rs6000_sdata`. We simply add one such check in front of all other `else ifs` and check for all the target flags that `-mcpu=nux` sets. Hence the target options will also set `rs6000_sdata` as we wish which is to `SDATA_NONE`. Although there exists a case for which this condition applies even if `nux` is not set as target and that is when all target flags are set by hand. Still if one chooses an explicit value for `-msdata` the option does

### 3 Extending the GCC Back-End

not apply and the explicit value is taken into account. This is somewhat not the ideal way of solving this problem but fulfills its purpose with as few side effects as possible.

Already this would allow us to use `-mcpu=nux` as target flag and `-ms2pp` as option flag. But since the flags we used are mandatory to the s2pp extension we also want to check these flags before starting compilation. First though we must create macros for each flag which the back-end notices. This is done in `rs6000-c.c` in function `rs6000_target_modify_macros` where we can define macros depending on target flags.

```
if ((flags & OPTION_MASK_S2PP) != 0)
    rs6000_define_or_undefine_macro (define_p, "__S2PP__");
```

If `flags` and the respective option mask are set, `rs6000_define_or_undefine_macro` will define a macro which is the second argument if a macro is defined or undefined depends on the boolean `define_p` but the back-end takes care of this on its own. We do this for all flags we are interested in. Now we will use these new macros to check if the flags are set. Before we do this though we create a new file that is `s2pp.h` and index this in `gcc/config.gcc` under `extra_headers` at `powerpc*-*-*`. We do this so GCC actually invokes the header file as it is not referenced elsewhere. After we added the usual conditional header macro to the file we add the first lines of code which are:

```
#if !defined(__S2PP__)
#error Use the "-ms2pp" flag to enable s2pp support
#endif
```

do more often like this:  
one example for repetitive use

Hence if `__S2PP__` was not defined because the target flag was not set, the compiler will emit an error that tells the user to set the target flag. This is done for all macros that we set above so to prohibit false use of `nux`. But we will also put in a second measure to prevent false use of `floats` by adding `nux` to the list of soft-float CPUs in `t-fprules`.

Since the preliminary requirements for the activation of the VE are now met, we start adding the remaining fundamental macros and also add a `vector` attribute that the user may use later. First we add a new Vector type which we will need from time to time. We call it `VECTOR_S2PP` and add it to the enumeration `rs6000_vector` in `rs6000-opts.h`. To put this to use we will create macros in `rs6000.h` which test vector modes if they correspond to the available modes in s2pp. Hence we add many macros of this form which take an argument and compare it to a valid s2pp type/value.

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

checks the mode's associated vector unit and compares this to `VECTOR_S2PP`. We also add checking for specific vector modes which shall be chosen carefully. The hardware only supports two types of vectors which are vectors with byte elements and vectors with halfword elements. Thus the respective modes are `V16QImode` and `V8HImode` which can be checked by `S2PP_VECTOR_MODE(MODE)`.

We also need to find conditions which should apply to `TARGET_S2PP` as they already to for `TARGET_ALTIVEC`. Luckily there exist only two options which prohibit misalignment of vectors and control the right alignment. Besides that only a few functions which we will discuss later need `TARGET_S2PP` to be called. Other options such a permutation of vectors

are not supported by s2pp or need a different implementation all together. The same procedure must be applied for checking vector modes by checking using `VECTOR_UNIT_S2PP_P` and similar macros. These are mainly used in `rs6000.c` when addressing vectors in memory through registers as operands. Similar but more complex implementations will build the rest of this chapter and therefore we will not go into detail for every macro like

```
#define UNITS_PER_S2PP_WORD 16
```

that is used.

Now that we have added the s2pp vector we identify it with the available modes which is done in `rs6000.c`. In `rs6000_init_hard_regno_mode_ok` we check for the s2pp target flag and assign `VECTOR_S2PP` to the modes `V16QImode` and `V8HImode`. Also we add the modes for preferred modes when vectorizing a value in `rs6000_preferred_simd_mode` and decline the usage of offset addressing for these modes in `reg_offset_addressing_ok_p`.

We will now define an attribute to use these vector modes. Thankfully 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_mask[]` array with the same values as for Altivec but changing the keyword. This forces us to add the function `rs6000_handle_s2pp_attribute` which is the same as the Altivec version but stripped off some vector modes which are not supported. We could use those attributes already but they would come in quite unhandy thus we will define an alias to represent the new attribute in `rs6000-c.c`:

```
builtin_define ("__vector=__attribute__((s2pp(vector_)))");
```

This is the same for Altivec and copies the usage of vectors in Altivec. As does our addition of an s2pp case in `rs6000_attribute_takes_identifier_p` to complete the attribute.

## 3.1 Registers

Now we start adding the registers of s2pp to the back-end. There are three types of registers we need to add:

**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

Registers are declared in `rs6000.md` thus we will do so as well.

```
...
(S2PP_COND_REGNO      32)
(FIRST_S2PP_REGNO     33)
(LAST_S2PP_REGNO      63)
(S2PP_ACC_REGNO       64)
...
```

necessary?

is added to the Definition of constants at the beginning of `rs6000.md`, where we give a range for the vector registers and after which we pick specific register numbers for the special vector registers. The number of these special vector registers does not matter though as we only need a number which no other registers that is used by nux refers to. There is a reason why we put the conditional register at 32 and only provide 31 vector registers: As the GPRs need the first 32 registers numbers (0-31) and there is never an FPU in nux, we want to use the 32 registers normally reserved to FPRs. However during the first tests of nux with its vector extension it became apparent that a reserved register with all bits set to 0 would come in handy at times. For one reason nux' instruction set does not include logical instructions especially no exclusive or (`xor`) function which is the standard way of having a registers set to all 0s (since `xoring` a value/register with the same value/register will always return 0) and also it does not provide an `or` function which is the standard way of moving registers around as it was shown in section 2.3.2.

check this reference

This led to the problem of “nulling” a register at times it was needed. Therefore the first register was left out and filled with zeros in order to be able to simply move/copy this register to a different one that needed to be nulled. Moving around registers was also realized differently than for normal architectures as `fxvselect` was used instead of `or` where the first operand is the destination while second and third operand are the source. The last operand, which is the conditional one, is set to 0 or left out as this substitutes the contents of the first operand always with that of the second operand. Now as this renders a simple work-around, substituting `xor` is more difficult and although there are ways of utilizing `fxvselect` to copy its functionality which we will care about later on.

explain this later

Particularly as `fxvselect` is the fast vector instruction as it takes only one cycle to complete and all other arithmetic instruction most likely take significantly more cycles. For this reason nulling a register by subtracting it by itself and storing the result in the same register would take more time hence would cost performance. Therefore the trade-off of having one less register at hand instead of wasting clock cycles continuously is adequate. Therefore since we do not want to reference the first vector register and cannot access the condition register by reference we simply reuse the register number for this cause. Unfortunately we do not have a second dispensable register of that kind and thus need to find a different index for the accumulator - though the same non-availability as for the conditional register applies. Luckily the POWER architecture does not reference the register index 64 for some reason so we can use that index without causing any conflict.

Now since we declared the vector registers' indexes we will create fitting macros as well. We need to decide on which registers shall serve which purpose. In this regard we shall also emphasize a little on fixed, call-used and saved registers. Fixed registers serve purpose that does not allow them to hold any other values at any time at all. Call-used registers are registers that are also not available for general register allocation as they are used for function calls and returning values. Only saved registers can be used for general register allocation as these may hold values over function calls such as variables that are often used throughout the program. This is done through macros which decide on the first saved register (in our case the 20th vector register) and list registers available for function arguments (a set of `S2PP_ARG_` macros). Also `FUNCTION_..._REGNO_P(N)` tells us if register `N` is used for function arguments in general. This number of course can be



optimized depending on the applications for the nux and also the style of programming but we keep the same number of registers saved as AltiVec and the FPU do.

Of course we need to go through the source files to find any use of those macros and add the new s2pp macros. We will not mention this every time as this is a standard procedure.

Now we need to get even more specific on the way the registers are organized. First we add `S2PP_REGS`, `S2PP_C_REG`, and `S2PP_ACC_REG` to the enumeration `reg_class` and define the identical names in `REG_CLASS_NAMES`. Then we need to specify the contents of our new register classes in `REG_CLASS_CONTENTS`.

```
/* S2PP_REGS. */
{ 0x00000000, 0xffffffff, 0x00000000, 0x00000000, 0x00000000 }, \
/* S2PP_C_REG. */
{ 0x00000000, 0x00000001, 0x00000000, 0x00000000, 0x00000000 }, \
/* FLOAT_REGS. */
{ 0x00000000, 0xffffffff, 0x00000000, 0x00000000, 0x00000000 }, \
/* S2PP_ACC_REG. */
{ 0x00000000, 0x00000000, 0x00000001, 0x00000000, 0x00000000 }, \
```

Each hexnumber in these arrays can be viewed as a bit mask where the least significant bit is the first register, next higher order bit the second register and so on. Each number is 32-bit and therefore equivalent to 32 registers. The following numbers start where the previous one ended, therefore is the second number the bit mask for registers 32 through 63 (32 is the 33rd register). Therefore does `0xffffffff` mask all register except for number 32 which is covered by `0x00000001` and 64 is masked the same way. One can see that the FPRs are masked completely as `FLOAT_REGS`. The reason for this order is that subsequent entries must not be subsets of previous masks but may extend these. Since we also included a register which was not used at all before we also have to change subsequent masks accordingly.

Now that we defined register classes we again add macros to compare these as we did before.

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

Next we need to pick names for those new registers that we just defined. These were chosen according to the constraints we will define next which are `kv` for normal vector registers, `kc` for the conditional register, `ka` for the accumulator. Defining (in this case alternative) names for registers is also part of a macro `ADDITIONAL_REGISTER_NAMES` with elements that look like:

```
{"kc", 32}, {"kv0", 33}, {"kv1", 34}, {"kv2", 35}, \
...
```

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

Now we also want to add the constraints which we just proposed. we therefore go to `constraints.md` and add the following code for `kv`, `kc` and `ka`:

```
(define_register_constraint "kv" "rs6000_constraints[RS6000_CONSTRAINT_kv
]"
```

```
"s2pp vector register")
```

where the first string is the register constraint's name and `rs6000_constraints[RS6000_CONSTRAINT_kv]` will be assigned a register name later on in `rs6000.c`, where it is also declared, which in this case is `S2PP_REGS` though this is only done if the target flag `-ms2pp` was set. The last string is only for documentary purposes. GCC (2017e) We chose `k` as the first initial of `s2pp` registers because there are very few letters left which are not used as a constraint already and `k` is one of them which is at least somewhat associated with the processor ("nuks"). The second character was chosen from the initial of the respective register type.

The next step which is necessary to make the registers widely accessible is adding them to `rs6000.c`. For once we add the new registers to `rs6000_debug_reg_global` in order for them to be listed when printing out debug information. This is done the same way `Altivec` registers are implemented and fairly easy. The same applies for pretty much all debugging support (THIS IS NOT `gdb`) therefore we do not mention additions to debugging, but in general these were done. It is far more important to add the registers to `rs6000_init_hard_regno_mode_ok` which initializes global tables that are based on register size. Basically every index in `rs6000_regno_regclass[]` is given a register class which corresponds to the register of that index and afterwards each register class is assigned a register type in `reg_class_to_reg_type[]`.

```
for (r = 32+1; r < 64; ++r)
    rs6000_regno_regclass[r] = S2PP_REGS;
...
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_REGS] = S2PP_REG_TYPE;
...
}
```

This is done only when `TARGET_S2PP` is true and substitutes the `FLOAT_REGS` at indexes 33 through 63 with `S2PP_REGS` and also adds the conditional and accumulator registers. To complete the process the register classes are assigned a respective register type which was added to the enumeration `rs6000_reg_type` before that. It is important to note that the `s2pp` normal register type was also added to the list of standard register types (again as a macro), as `s2pp` registers are meant to act like normal registers and also to the floating point/vector register types which is necessary as they are handled differently later in the source file.

previous section

Register can also be fixed () according to compiler flags which is done in `rs6000_conditional_register_usage`. E.g. when the flag for soft-float is set, the compiler calls all FPRs fixed (ergo not available), which is contrary to our plan of using these registers. That is the reason why we add the alternative condition that the `s2pp` flag may not be set for this to happen. Instead we fix the registers 32 and 64 which are the used otherwise (`kc` and `ka`).

Now that we fulfilled all measures of adding the registers to `rs6000.c` we need to add them to the list of possible `asm` operands in `ppx-asm.c` where a simple definition like

`#define k0 1` is enough if this is done for each register name. This tells the compiler to substitute `k0` for `1` as only pure integers are valid machine operands.

The last step to completing register usage for the back-end is adding fitting predicates in `predicates.md`. We can copy these from the respective AltiVec predicates and change both the vector specific macros and the predicates' names. The affected predicates are:

`s2pp_register_operand`

is a vector that is stored in a s2pp register

`easy_vector_constant`

returns 1 if the operand is a constant vector and now also tests for s2pp vectors and calls a new function `easy_s2pp_constant`

`indexed_or_indirect_operand`

applies for indexed or indirect operands (e.g. memory addresses in registers) and now supports the case of a s2pp vector operand as for AltiVec vectors

`s2pp_indexed_or_indirect_operand`

is the same but only for s2pp vectors, not AltiVec

If one wants to add own predicates, GCC offers a manual entry GCC (2017m).

The `easy_s2pp_constant` function checks if an operand is “splittable” which means that all elements are the same and thus can be generated through a split instruction. To check this the operand is analyzed sequentially if either of the two available splats can synthesize the same operand. This was also transferable from an AltiVec equivalent with the exception that one vector mode had to be removed. prologue epilogue Additionally the similar function `gen_easy_s2pp_constant` could also be transferred as it works the same way but generates RTL code that will create a constant vector operand from a different operand.

Now as we have completely added the new registers, we must take care of some consequences this caused. Mainly our problem is that we reused the floating point registers and did not mark them as fixed. For this reason we must scan the source files for `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.

earlier

## 3.2 Prologue and Epilogue

Also we have a second problem that reusing FPR indexes causes. Since there is an ending number of registers the compiler can use for saving values, and this number is also reduced by the amount of fixed registers, the compiler occasionally stores values in memory before calling a function that needs some of the available registers by calling a so called “prologue” (GCC, 2017i). The compiler then restores the registers after the function has finished through an “epilogue”. To understand the principles of these we will take a look at how the back-end saves registers. Before the compiler saves the registers it

### 3 Extending the GCC Back-End

must check which registers need to be saved. It then sets a variable for each register type accordingly. In `rs6000_emit_prologue` the compiler checks for these variables and starts saving the respective registers. Since the compilers looks for register numbers instead of types, we need to alter the case for FPRs and add a target flag, which activates these lines of code:

```
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 = 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);
  }
```

The compiler first loops over every save vector register and checks if the register needs to be stored. Then an offset is computed that takes the current frame offset and adds 16 bytes (this is the register size) for every register of this type. `savereg` and `areg` are two kinds of register that will later serve as operands. `savereg` is the vector register we want to save and `areg` will be the GPR operand needed to use vector memory instructions. Hence `areg` loads the offset just calculated through `emit_move_insn`. `emit_move_insn` will then insert an insn to the IR that moves the second argument to the first. 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 to where all registers are saved, 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 moves belong to a function call. After the functions code has been compiled the back-end invokes `rs6000_emit_epilogue` which uses the same functions as the prologue but restores the registers from memory. Of course this function can not work on its but needs other functions that set parameters and prepare statements as well. These functions were also extended by adding conditional cases for s2pp that mostly agree with the respective AltiVec cases. 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.

fxvstax and fxvlax

## 3.3 Reload

As hinted in chapter 2 `reload` is a now deprecated process in GCC that mainly performs register allocation. Obviously we need to add special handling of vector registers to

`reload` because the registers we just added to the back-end must be used somehow. We thus get back to `S2PP_REGS` which were the normal vector register form earlier. As `reload` is capable of moving the contents of registers, we must specify that `s2pp`'s registers are not compatible with GPRs or any other registers and also that storing/loading registers takes two GPRs that hold the memory address. Because registers are quite different in their specifications and `reload` could possibly ask for any combination of source/destination register we declare only registers moves between `s2pp` registers only or between `s2pp` registers and memory valid. This also creates the need for checking the register class of an RTL expression and eventually correcting the register class. Additionally as `reload` does not support the addressing mode which `Altivec` and `s2pp` both use, we must rearrange indirect addresses the same way `Altivec` does. At last we also need to validate the mode which is used. All of this is done in a set of `rs6000_secondary_reload_` functions which need to be modified.

### 3.4 Built-ins and Machine Instructions

Basically the back-end is now almost compatible with `s2pp` vector instructions. The only thing that is left is specifying the machine instructions which move registers or access memory. We therefore add a machine description file `s2pp.md` to the back-end which will contain all available vector instructions. On which is `*s2pp_mov<mode>`:

```
(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 this shortly. 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. The sequence is fairly simple and states that operand 0 is set by operand 1. It only

### 3 Extending the GCC Back-End

gets interesting when taking a look at the constraints which build pairs for each operand and are separated by commas. The first pair for example (`=z` and `kv`) tells the compiler that a memory which is accessed by an indirect operand will be set by the contents of a vector register (this is the first use of our new constraint. Which machine instruction is used for each pair of constraints is stated in the output template which may also be in C form. Each case is indexed according to the position in the list of constraints so we take a look at `case 0` where we see the matching machine instruction for storing a vector in memory `fxvstax`. The second operand contains also a character besides its number which is an operand modifier GCC (2017c) that will cause the operand 0 to be split into two address operands. Besides the vector load instruction `fxvlax` we also note the `fxvsel` instruction which is used to move one vector register to another (pair `kv` and `kv`) and `case 6` which moves the contents of vector register 0 (this register is all 0s) and thereby nulls that register. The constraint `j` is the respective constraint for a zero vector.

But still there exist instruction which are not specified in detail but only by “#”. A # is the equivalent to stating that there is no instruction which can perform the instruction which is described by the RTL above. This causes the compiler to look for different RTL that has 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 also define which RTL statements are equivalent by using `define_split` which is documented in GCC (2017h). As there exist splits for cases 3 through 5 in AltiVec, we may not define splits ourself.

Splitting insns is quite common for the GCC back-end because splitting insns with unspecific constraints allows for easier generation of code, since the back-end will search for code that fits the operands of the instruction which is called in the source files. Even `s2pp_mov` is mainly used by calling a general vector move insn in `vector.md` that has the same RTL code without any constraints and then is split. We only need to add the `s2pp` vector mode macro to the condition of the insn which is specified through `define_expand "mov<mode>".` Expanding an insn which is done for this code basically follows the principle of splitting and goes even further in not giving the option for a machine instruction at all but relies on matching RTL in some machine description of the back-end. Detailed information can be acquired in GCC (2017f).

As this is done for register moves is general we will do the same for specific loads and stores, by adding mode checking in `vector.md` and adding a matching insn to `s2pp.md`.

Beside adding the other insns we also need to add the function `rs6000_address_for_s2pp` which converts a standard address to an `s2pp` compatible form. This code could be taken from an AltiVec equivalent since both vector units share the same form of memory referencing.

Now before we continue to other insns, we shall go back to `s2pp_mov` and take a look at `case 7` which applies for constant vectors. Constant vectors which have a constant value that does not depend on a register thus is known at compile time. GCC will look for such vectors and check if these vectors can be splatted (this means all vector elements have the same value). This is a relict of AltiVec which offers a special `splat` instruction that takes a constant immediate value as operand. We kept the 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()` which is also used by AltiVec and implemented similar code to the existing AltiVec code but reduced this to the null vector case and emit a `#` for splitting otherwise.

Since we are now missing an instruction to use this split we will create 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);
  }")
```

We split the upper RTL sequence which moves a constant vector to a vector register onto the bottom sequence which inserts an intermediate step. `match_dup n` means that the operand should match the operand with the same index. The C code that follows specifies the newly added operands 2 and 3 further and converts operand 1 into a single integer element because all values are the same. The second RTL sequence uses the newly created integer and moves it to a GPR which then is 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.

```
(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")])
```

is the insn the split refers to.

After we finally added memory instructions we can also implement them in `rs6000.c`. In `rs6000_init_hard_regno_mode_ok` we assign the code for store and load instructions to the two supported modes in case the target flag is set but do not specify any others.

All that is left now are intrinsic functions that make use of the memory instructions. Although by now the compiler would already support `asm` usage as shown in 2.3.2 and which we will get back to later on. Still we do not want to stop there and also add intrinsic functions to the back-end. This is done through several steps that were also described in an internship report *Heimbrecht* (2017). 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 *Friedmann* (2016). In order to allow access to the synram we implement insns that work similar to `fxvstax` and `fxvlax` and are called `fxvoutx` and `fxvinx`. 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



### 3 Extending the GCC Back-End

instructions exist in multiple versions that either support conditional execution or do not. This is due to a problem that was discovered only recently with the PPU. As tests revealed the conditional execution of arithmetic instructions works improperly and in case a condition does not apply the result of a previous instruction is written to the first operand. Normally the operation should leave the contents of the operand untouched instead. For this reason we implement a workaround through `insn` splits that utilizes `fxvselect` and its conditional execution as this does work as intended. Though we still offer arithmetic operations without splits as this saves a clock cycle in comparison to having an additional `fxvselect` instruction. Besides those simple arithmetic operations, `fxvadd...`, `fxvsub...` and `fxvmul...`, there also exist more complex operations that make use of the accumulator. As of now these instructions should not be used with conditionals as further testing is imminent and it is not clear whether those conditionals would work. Besides that does an extra instruction render the advantages of an accumulator meaningless. Nonetheless do we implement those accumulator instructions together with a conditional argument which would allow for easier testing later on. Also does the additional operand not influence the performance of the instruction in a bad manner since for the conditional 0 the instruction works the same as if the conditional was true for all elements.

We will refer back to this when discussing the results of extending the back-end.

Since this completes the `insns` we can go on and create built-ins in `rs6000-builtins.def` from these `insns`. First though we must add macros that makes adding intrinsics easier. The exact definition of those macros is described for AltiVec built-ins in *Heimbrecht* (2017) but we can easily transfer those for s2pp by adding the `RS6000_BTM_S2PP` built-in mask which is identical with the target mask. We then use the 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 exists 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 the types of their arguments later on what makes using them easier and stops false usage. Overloads are also further explained in *Heimbrecht* (2017) As we are adding overloads we can put them to use in `rs6000-c.c` where overloads and built-ins are connected. This is done through structures that combine the built-in and overload names with a return type and up to three arguments.

To actually make use of 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. Therefore we must tell the back-end which functions to use by making a target dependent decision in `rs6000.h` which overload-resolving function is handling overloads in general. Also we need a functions in `rs6000.c` that handle built-ins in general. First we create a new function `s2pp_expand_builtin` which is invoked by `rs6000_expand_builtin`. This function handles all special built-ins that belong to s2pp and picks expander functions according to their name. We therefore must create those expander functions that take care of certain kinds of built-ins. One such group are



memory intrinsics that handle explicit memory addressing and also apply to synram intrinsics. These expanders are needed because of the special way that memory is accessed through indirect register referencing. As for the operand modifiers we can use the same implementation that AltiVec uses but create a new function in case improvements may become necessary at some point in time.

Besides these expander functions we additionally create a new kind of expander function that did not exist before s2pp. As there is a great number of instructions that do not return any values because they write the accumulator or the conditional registers, we create expander functions for various numbers of arguments that each do not handle a return operand. These functions which were called `s2pp_expand_unaryx_builtin`, `s2pp_expand_binaryx_builtin` and `s2pp_expand_ternaryx_builtin` otherwise behave quite similar.

Now that we are already handling special built-ins we need to define those built-ins in `rs6000.c` as well. `s2pp_init_builtins` takes care of that as it is a series of `define_builtin` functions which must be written explicitly in contrast to normal built-in functions, where this is done automatically.

We did not mention so far the special case of three other built-in function that belong to the group of special built-ins as well but differ in how the RTL code/machine instructions are generated. `vec_ext`, `vec_init` and `vec_promote` describe functions which do not belong to a specific vector instruction but are compiler-constructed sequences of machine instructions that correspond to a certain action on a vector which is mainly performed in memory. These functions are inspired by AltiVec's implementation which easily was transferable to s2pp.

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 ??.

? target pragmas compiler anweisung stack boundary -> put this together frame offsetr stackoffset fix pck instructions

## 4 Results and Applications

Now that the extension of the back-end is finished we want to emphasize on the positive effects of this.

As of now the compiler back-end works and can be built to a working compiler. This new compiler supports the use of target flags which are `-mcpu=nux` and `-ms2pp`. Also a new header file named `s2pp.h` exists and can be included the same way GCC standard header files are included. Only together will this allow for using the `vector` attribute which creates vector variables of different types.

These vector variables specifically serve as arguments for newly implemented intrinsic functions that exist for 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.

Assigning registers as well as memory works as expected and does not need for user interaction. Despite the different bus, the back-end also rightfully accesses the synapse array when told to. Specifically the user must use the intrinsics `fxv_inx` and `fxv_outx` to access the synapse array and load/store variables from/to there.

As many intrinsics as possible were designed according to existing intrinsics for AltiVec and use the `vec_` prefix as well a `fxv_` prefix. For every previously available nux macro now exists an intrinsic function with the same name and the same arguments.

The compiler also supports coding in assembly using vector instructions without choosing hard registers but again loads and stores variables automatically and supports the constraints we implemented. This will make it easier to combine low-level coding in high-level programs if this is ever needed. One problem that was detected in this regard appears when choosing hard vector registers with `asm` which then are used in the output template. It causes a replicable compiler error which should be addressed soon. Besides that the compiler encountered various errors during development that had different origins. One such error which often emerged was due to the overlapping register numbers with FPRs but could be fixed by adding a target dependency to register dependent actions and also adding prologue and epilogue functions which became necessary when compiling with optimization. Other errors included non-matching RTL code and the compiler trying to move registers between GPRs and vector registers. Many errors shaped the back-ends code but of course further testing could reveal unknown insufficiencies as bugs are constantly appearing for GCC as well.

Nonetheless, first tests which used intrinsics exclusively instead of macros produced working machine code as well as correct results. Through these these early testing stages David Stökel was able to find a nux bug which previously was not known. This thesis already referred to that bug which affects conditional usage of arithmetic instructions. It also lead to the first in a newly proposed series of tests for nux which should check for

more bugs like this in the near future. Thanks to the `asm` support that nux gained these tests may not completely be written by hand but could be computer generated.

The compiler also supports different levels of optimization but tests for this are yet to come as checking for bugs caused by optimization is quite difficult.

Maybe the biggest question yet to answer is the reliability of the back-end as only correct code will be of any value. Therefore this thesis highly encourages the implementation of automated tests and a simulation platform as currently tests can only be conducted on hardware which is not fully tested either. This causes a conundrum where locating an error automatically is impossible.

Still there are ways of checking the correctness of code, mainly by reading the assembly output. To illustrate this, listing ?? shows the comparison of newly written code that uses intrinsics, the assembly code that results from this with and without optimization and equivalent code that is written using the old `fxv.h` header file.

write an analysis of this

still compiles old code -> check this works with libnux is available in waf some users are interested -> this will be a challenge

## 5 Discussion and Outlook

Discussion...

future of PPU additonal intrinsics on request FPGA speicher

# Appendix

## Acronyms

<b>ADC</b>	Analogue Digital Converter
<b>ALU</b>	Arithmetic Logic Unit
<b>ASIP</b>	Application Specific Instruction set Processor
<b>asm</b>	Assembly
<b>CISC</b>	Complex Instruction Set Computer
<b>CPU</b>	Central Processing Unit
<b>CR</b>	Conditional Register
<b>DLS</b>	Digital Learning System
<b>DRAM</b>	Dynamic RAM
<b>insn</b>	instruction
<b>IR</b>	Intermediate Representation
<b>FPGA</b>	Field Programmable Gate Array
<b>FPR</b>	Floating Point Register
<b>FPU</b>	Floating Point Unit
<b>GCC</b>	GNU Compiler Collection
<b>GPP</b>	General Purpose Processor
<b>GPR</b>	General Purpose Register
<b>HICANN</b>	High Input Count Neural Network
<b>ISA</b>	Instruction Set Architecture
<b>LLVM</b>	Low Level Virtual Machine
<b>LR</b>	Linker Register

## *5 Discussion and Outlook*

**LRA** Local Register Allocator

**MMU** Memory Management Unit

**MSB** Most Significant Bit

**nux** alternative name for PPU

**POWER** Performance Optimization With Enhanced RISC

**PPU** Plasicity 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

**VRF** Vector Register File

**VMX** Vector Media eXtension

## **List of nux Intrinsics**

intrinsic name	use	data types				effect
		d	a	b	c	
fxv_add vec_add	d = fxv_add(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		add a and b modulo element-wise and write the result in d
fxv_sub vec_sub	d = fxv_sub(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		subtract b from a modulo element-wise and write the result in d
fxv_mul vec_mul	d = fxv_mul(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		multiply a and b modulo element-wise and write the result in d
fxv_addfs	d = fxv_addfs(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		add a and b saturational element-wise and write the result in d
fxv_subfs	d = fxv_subfs(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		subtract b from a saturational element-wise and write the result in d
fxv_mulfs	d = fxv_mulfs(a,b)	same as a	vector signed char vector unsigned char vector signed short vector unsigned short	same as a		multiply a and b saturational element-wise and write the result in d
fxv_stax vec_st	fxv_stax(a,b,c)		vector signed char vector unsigned char vector signed short vector unsigned short	int	vector signed char* signed char* vector unsigned char* unsigned char* vector signed short* signed short* vector unsigned short* unsigned short	a is stored to memory address c + b
fxv_outx	fxv_outx(a,b,c)		vector signed char vector unsigned char vector signed short vector unsigned short	int	vector signed char* signed char* vector unsigned char* unsigned char* vector signed short* signed short* vector unsigned short* unsigned short	a is stored to synaptic address c + b
fxv_lax vec_ld	d = fxv_lax(a,b)	vector signed char vector unsigned char vector signed short vector unsigned short	int	vector signed char* signed char* vector unsigned char* unsigned char* vector signed short* signed short* vector unsigned short* unsigned short		d is read from memory address a + b
fxv_inx	d = fxv_inx(a,b)	vector signed char vector unsigned char vector signed short vector unsigned short	int	vector signed char* signed char* vector unsigned char* unsigned char* vector signed short* signed short* vector unsigned short* unsigned short		d is read from synaptic address a + b
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_extract(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
vecinsert vecpromote vecsh fxvsh fxvpcku fxvpckl fxvupckl fxvupckr vecsplats16 fxvsplab fxvsplath vecsplats8 fxvcmp fxvmtac fxvmtacf fxvaddactacm fxvaddactacf fxvaddacm fxvaddacfs fxvmam fxvmafs fxvmatacm fxvmatacfs fxvmultacm fxvmultacfs fxvaddtac						

# Notes

add reference . . . . .	3
cite friedmann dissertation . . . . .	3
complexity > lower clock freq . . . . .	4
inhaltlicher zusammenhang . . . . .	4
graphic of opcode . . . . .	5
reference . . . . .	7
reference . . . . .	7
put the following part in the next section . . . . .	8
add something here . . . . .	8
explain this further . . . . .	9
reference . . . . .	10
check this . . . . .	11
check with ahartel on this . . . . .	11
applications of the PPU today like in-the-loop experiments and controlling . . . . .	12
caption and reference to PPC book and asm website correct the table . . . . .	18
all these tables in the appendix . . . . .	19
reference LLVM vs. GCC on ARM, LLVM vs. Gcc in EISC . . . . .	21
examples of RTL code . . . . .	24
MMU ?= memory controller PPC must handle syncing in compiler when I/O is added explain stack and frame pointer PPU instruction set . . . . .	24
do more often like this: one example for repetitive use . . . . .	28
necessary? . . . . .	30
check this refernce . . . . .	30
explain this later . . . . .	30
previous section . . . . .	32
earlier . . . . .	33
fxvstax and fxvlax . . . . .	34
? target pragmas compiler anweisung stack boundary -> put this together frame offsetr stackoffset fix pck instructions . . . . .	39
still compiles old code -> check this works with libnux is available in waf some users are interested -> this will be a challenge . . . . .	41



# Bibliography

Altivec manual, chapter.

Gcc wiki, accessed 2017.03.26, 2013.

*GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., <https://gcc.gnu.org/onlinedocs/gccint/index.html>, 2017a.

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

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

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

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

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

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

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

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

*GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 16.23 Iterators, 2017j.

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

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

*GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 16.7 Predicates, 2017m.

- GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 13.8 Registers and Memory, 2017n.
- GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 13.15 Side Effect Expressions, 2017o.
- GNU Compiler Collection Internals Manual*, Free Software Foundation Inc., 16.9 Standard Pattern Names For Generation, 2017p.
- Friedmann, S., *nux Manual*, 2016.
- Friedmann, S., J. Schemmel, A. Grübl, A. Hartel, M. Hock, and K. Meier, Demonstrating hybrid learning in a flexible neuromorphic hardware system, 2016.
- Heimbrecht, A., Internship report — altivec intrinsics, 2017.
- 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, February 27, 2017

.....  
(signature)