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 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. To accelerate the speed at which this so called plasticity processor unit (PPU) computes all synaptic weights of every neuron used, the processor has an extended instruction set architecture (ISA) that supports vector registers and single input multiple data (SIMD). This report deals with the task of adding built-in functions to an existing back-end of GCC, specifically the one used by the PPU, in order to extend the already implemented set of functions according to the users needs.

# Contents

# 1 Introduction

As neuromorphic computing becomes ever more popular, many applications for neuromorphic systems emerge and make use of neuromorphic hardware's advantages over traditional computers. But to do so one must be familiar with a system as often it differs in many ways from common system architectures. As a result programing 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 have got used 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. As basic programming is still 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 for the reason of not making a user learn assembly that wants 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; thing like 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 such reluctant when it comes to "antique" programming are compilers.

What compilers have done and do for us every single day has become normal and we enjoy the state of not worrying about what a computer does at its core as long as we get a result at the end. Hence the absence of a compiler strikes us quite hard.

But luckily the PPU is not without compiler support, though this does not apply completely. The part which is responsible for computing learning rules -of all parts- is the one part that is not supported by any compiler there is. This is due to the custom nature of the PPU, which was developed solely for the BrainScaleS project and the hicann-DLS. A user therefore needs the most basic kind of programming to make use of the additional features of the PPU, which is the current state of programming. In the worst case this can lead to ineffective programs - as performance is important for neuromorphic programming - and demand an unreasonable amount of time and work to achieve simple results.

The only way to fix this situation is adding support of the PPU to a compiler.

Not only could this allow for full C-style programming when working on the PPU but also include code optimization and code debugging. We therefore aim to achieve "full"

compiler support of the PPU's hardware and make programming easy again.

This thesis will focus on the way to achieve this and briefly explain the process itself. As a fundamental knowledge of both processors and compilers is needed along this way we will start with a very basic introduction to both in general and also apply this to the specific processor and compiler we use. This may not make actual literature for both obsolete but should explain the basic concepts to an extend which is sufficient for our cause. The following chapter will then explain a few test scenarios that put the contests of the previous chapter to use and also make us more familiar with the PPU. Afterwards the process of extending the compiler is explained and the result of this presented. It follows a recap of what was done and an outlook to what might be done in the future.

# 2 Methods

## 2.1 Basic Processor Architecture

Next to all processors used these days are built upon the so called von-Neumann architecture . Though 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 are needed at some point. The main advantage of digital systems over analogue systems such as the human brain, is the ability to do calculations at much higher speeds. For this reason “normal" processors are responsible for handling experiment data as well as setting up different parts of the experiment. We now dive shortly into 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 a 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 told 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 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 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 registers. Its value can determine for some instructions if they are 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 for perform the aforementioned logic or arithmetic operations and saves the result there as well. In case of more complicated 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

$$\text{speed(accumulator)} < \text{speed(register)} \ll \text{speed(memory)} \tag{2.1}$$

As speed is always important in computing we therefore want to use registers as much as possible and only write results to the memory or save registers when there are not enough registers available for the current task. 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 things like:

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

These items always influence each other as a complex instruction set means that complicated arithmetic operations with many operands can be done in few clock cycles but this often also means that the maximum clock frequency must be lower as the circuit design has a longer time constant until the next instruction may follow. During a single clock cycle a chip usually does 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
```

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. Now the faster the clock frequency the faster these micro instructions will have finished 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 easily more complicated and needs more space as the instruction set gets more complicated.

Because of this one usually differs between two kinds of processor:

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

The latter one usually is reduced to such simple instructions as `add` or `sub` and connects them to create more complex instructions overall. As the PPU is a RISC architecture we focus on its key values. This is similar to the micro instructions which were mentioned earlier, but now every instruction has the same set of microinstructions with a different operation at 5. RISC architectures therefore start "pipelining" their instructions which means stating the next machine instruction as the previous machine instruction finished 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 would take to complete a single machine instruction. It must be noted though that the processor must detect hazards which are data dependencies between instructions where one instruction needs the result of another. Such instructions usually are postponed in 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.


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

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:

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

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). The 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. These vector registers can either hold sixteen 8-bit `char`s, eight 16-bit `short`s or four 32-bit `int`s or single precision `float`s, 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:

http://www.nxp.com/assets/do
manuals/ALTIVECPEM.pdf

First the PPU's VE uses a conditonal 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 SRAMaddress of the synapse do not match an thus the spike is ignored by the synapse. explain this further 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.

### 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 ship 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 changes 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

PPU paper

The PPU, which was designed by Simon Friedmann , 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 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 an thus allows for little out of order operation for instructions in these reservations 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. 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

check this

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.

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 preformance. 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 ...., which is recommended as accompanying literature besides this thesis. In general these opcodes are divided into 6 groups of instructions:

reference to nux manual

**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.

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 ... illustrates the chain of tools that is involved into this process:

1st column: a graph from p. 5 of book "compilers" 2nd column: front-end, middle-end, back-end 3rd column: the different phases of a compiler

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 strings 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.

s.o.

But since we are more interested in compilers than other components, we will take a better look at the compiler itself. Figure ... 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

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.

The different phases of a compilation process are illustrated to the far right of figure ... . 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. The **source code optimizer** which is the last phase of the front-end takes the syntax tree and takes the first shoot 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.

### 2.3.2 Instrinsics

### 2.3.3 Assembly Basics

light memory barrier

### 2.3.4 GNU Compiler Collection

reload register spilling register handling endianess wordsize machine isntructions
different parts of an instruction, mention: opcodes, asm instruction, operation, operand, insn, IR, builtin function/intrinsic

light memeory barrier Typically such a memory barrier is called like `asm (::: memory)`, which the compiler recognizes as a memory barrier.

## 2.4 Current State of the System

This section is probably unnecessary

# 3 Test cases

not all cases can be handles by the aforementioned method. For example a built-in function without a return type. In this case we use the BU_ ALTIVEC_ X template that slightly differs from other templates. . . . . .

asm tests user level tests compiler tests

# 4 Results

still compiles old code optimizes runs on PPU

## 4.1 Extending the GCC Back-End

To allow the compiler to differ which insns we acutally want to use as a built-in, the rs/6000 back-end has a special file which contians macros for every builtin function that will be available. These macros will be loadad in the rs6000.c file and also handled there. But before this can happen we have to create a built-in macro for our newly created insn. WE must first differ between the different kinds of macros. The rs/6000 back-end already provides us with templates for differnt altivec built-ins. These templates basically help identifying the right macros later on and avoid naming conflicts. Next we have to differ between the number of input opernad our built-in function is going to have and the number in th etemplate name is meant to be equal to th enumber of input operands . all Macros expect output operand except for the X-names macros. which are special cases we will emphasize later. We now choose `BU_ALTIVEC_2` because our newest insn has to input operands and add the following lines:

the first argument is the name of the macro and will be used by the compiler internally. the user only gets interact with the second string which is the new built-ins name for the front end. (for altivec built-ins the tempalte adds `__builtin_altivec_` in front of the given name). Next the attribute for the builtn must be given. typically this `CONST` but for builtins that load or store in the memory this is `MEM` for example. The last argument is the name of the insn we decided on earlier.

You can also add an „overload“ macro for your builtin as this allows for type checking and reducing builtins for different vector modes to just on final builtin. here we first need a Macro name and second the function name we want to use later (this functon name is preceeded by `__builtin_vec_`)

After we fully implemented our builtin functions we will finalize the functions by defining their final names. There are two naming conventions we can follow and typically it is advised to follow both in order to offer the user a set of alternatives. The first naming convention is the nameing convention already used by the AltiVec extension which starts every builtin function with vec_ and chooses a general descriptive name which is derived from the used Assembly macro if possible. This is the most intresting case for our exemplary Altivec builtin function. because there is already a vec_ mul builltin we will choose the descriptive abbreviation vec_ smul for scalar multiplication. The other naming convention gets intresting when extending the backend with another vector extension. If there were an S2PP vector extension, all assembly macros would begin with fxv which is a good alternative for the vec_ prefix altivec uses for its builtins.

then it is a good idea so offer all assembly macros with their already known names e.g. fxvaddbm would become fxv_ addbm. for more genral builtins that combine the halfword and byte macro it is recommended to leave out the letter h or b likewise. rendering fxv_-addm the builtin function that works with both V8HI and V16QI vectors.

important files:

**rs6000.md**

**rs6000.h**

**rs6000.c**

**rs6000.opt**

**rs6000-builtins.def**

**rs6000-cpus.def**

**rs6000-c.c**

**rs6000-opts.h**

**rs6000-protos.h**

**rs6000-tables.opt**

**driver-rs6000.c**

**ppc-asm.h**

**s2pp.md**

**s2pp.h**

**constraints.md**

**predicates.md**

**vector.md**

**sysv4.h**

**t-fprules** bla bla bla

# 5 Discussion

Discussion...

# 6 Outlook

Outlook...

# Appendix

# Notes

## 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 19, 2017

.....
(signature)