

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	1
1.1	Basic Processor Architecture	1
1.1.1	Memory	3
1.1.2	Vector Extensions	4
	AltiVec Vector Extension	4
1.2	Basics of Neural Networks	5
1.3	Implementation in Hicann-DLS	6
1.3.1	The plasticity processing unit	6
2	Compiler Structure	9
2.1	back-end	10
2.2	intrinsic	10
3	Extending the GCC Back-End	11
4	Test cases	13
5	Results	14
6	Discussion	15
7	Outlook	16
	Appendix	17
	Bibliography	19

1 Introduction

The goal of this thesis is the extension of an existing back-end of GCC to a point where it supports the existing architecture of the PPU a.k.a. nux.

1.1 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 freidmann 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.

1 Introduction

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

$$\text{speed(accumulator)} < \text{speed(register)} \ll \text{speed(memory)} \quad (1.1)$$

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 instruction. An example for an add instruction (`result = add(a,b)`) would be:

1. clock cycle load a from `register 1`, load b from `register 2`
2. clock cycle add a and b
3. clock cycle write result to `register 3`

The faster the clock frequency the faster this instruction will have finished the faster the processor. But also the more complex the instruction set is the fewer 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 provides more registers as the ALU needs less space and 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. Also RISC processors allow for pipelining which means that instructions that take more than one clock cycle are put in a so called pipeline where each step of an instruction is can be used by subsequent instructions. This means that as the first clock cycle after starting an instruction has finished a different instruction is started directly as the first step of the previous instruction is completed and therefore available again. This virtually increases the clock frequency as many instructions can be performed in parallel.

Next we take look at memory. Normally the memory of a von-Neumann machine contains both, the program and data. 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 that resemble the following.

graphic of opcode

The first part which is called opcode is simply a number that stands for an operation performed by the ALU. Typically this is something like 8 bits long. The second and third parts are the argument addresses or operands of an operation. Addresses may only be as wide as a register i.e. 16 bit for the processor to be able to access all addresses. The last part is the result's address which is also as wide as a register in this case 16 bit. An instruction therefore takes a total of 56 bit in this case which is a little bit less than 4 addresses (remember 16 bit each) of memory. Therefore instructions may be limited in size which is supported by the architecture. In this case a third operand would not be possible.

little more on operands next operand types, register memory then memory and mmu

Therefore it is important, especially for larger scale systems, to keep track of memory usage. This is e.g. to prevent malicious programs from accessing memory which is used by other software. For this reason next to all CPUs include 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. The PPU though includes only a very simple MMU that does not cache memory instructions and also has matching virtual and physical addresses. The memory of the Hicann-DLS is 16 kiB in total (ranging from addresses 0x0000 to 0x4000) and both the vector extension and the main processor use the same MMU.

bus

use this:

This may result in unintended reordering of memory operations which then leads to errors. Because of this there is the possibility to prohibit such reordering by introducing a memory barrier. A memory barrier basically is a line of code that splits the remaining code into code that occurs before the barrier and code that occurs after the barrier. It prohibits the compiler from mixing instructions from different sides to the

barrier due to reordering. Typically such a memory barrier is called like `asm (::: memory)`, which the compiler recognizes as a memory barrier. On a processor level there exists also a memory barrier that would typically not be called by a user. It's called syncing and on PowerPC the instruction `sync` is used for this. Because the processor itself delays certain instructions to optimize performance (as part of its out-of-order architecture) it may happen that a dependency between two instructions is not detected. This would result in the very same problem as described earlier. Therefore the processor can be instructed to wait until all store and load instructions are executed. The memory controller then sends a signal to the processor that it is all done. It is fairly obvious that only the second kind of memory barrier helps us in or cause to avoid wrong ordering in the memory controller. Luckily both the VE and the basic processor use the same memory controller which makes `sync` apply in both cases. Also since all vector instructions must go through the main processor first, any hold executing instructions on the MP affects the VE the same way. Besides the VE and the MP, the memory also provides access to the FPGA through a different interface in order to allow for external access to the memory. This is needed for writing programs into the memory as well as getting results during or after execution. This also allows for communication during runtime of the PPU.

MMU

1.1.1 Vector Extensions

Using the same principles as mentioned in the section about von-Neumann architectures, there is also the possibility to create registers or whole architectures that make use of even more than the common 64-bit. 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:

- 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. The AltiVec extension provides the processor with a single-precision floating point and integer SIMD instruction set. The vector registers are 128-bit each and 32 in total. These 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 are a few differences though:

First the PPU's VE uses a conditional register (CR) to perform instructions only on those elements of a register, that meet the condition, which is specified by the user, while the AltiVec VE utilizes the CR which is 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 registers on its own though, which are the VCSR and VRSAVE registers. The Vector Status and Control Register (VCSR) 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 registers 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.

vector register file ALU FPU memory memory controller = MMU clockcycle pipeline

preliminary paragraph with: physiology -> hicann -> PPU -> processor in general -> compiler -> backend methods in next chapter

This thesis mainly focuses on an essential part of the HICANN-DLS system. HICANN-DLS stands for high input count analogue neural network Digital Learning System. For this reason it is important to understand the basics behind this and what the PPU is meant to do.

1.2 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

<http://www.nxp.com/assets/documents/manuals/ALTIVECPM.pdf>

1 Introduction

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 damped . 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. 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.

1.3 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. 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 6 bit weights are always right aligned and the most

significant bit has a value of 2^{-1} with subsequent bits having half the value of the previous bit. The first two of these bits 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.

sram

1.3.1 The plasticity processing unit

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

PPU paper

One such task is applying learning rules to the synapses during or in between experiments which can either be done by hand or with the help of the aforementioned PPU. The second option is especially valuable when updating synaptic weights during an experiment as the PPU does this much faster than a system which interacts from the outside. This is important for achieving experimental speeds that are 10^4 times faster than their biological counterparts.

the PPU 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. These instructions then go into a queue that holds all vector instructions. Vector instructions then are processed in-order from this queue. Between being fetched from the queue and execution the instructions shortly stay in a reservation station that is specific for each kind of operation and thus allows for pipelining of such instructions. This means i.e. that during the process of accessing a vector on memory, some arithmetic operations can be performed on a different vector. This allows for faster processing speeds as such instructions can be run out of order. The limiting factor for this remains the vectors register file's single port for reading and writing.

The other limiting factor in processing speed is the memory access. Both, the GPP and the VE, share the same memory management unit (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. For this reason one must be aware of the `sync` instruction that 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 and therefore ~~should only be done~~ if necessary. `sync` is a standard instruction of the PowerISA and further information can be found here

reference

1 Introduction

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. It is therefore a key feature of the hicann-dls as it allows for implementation of different plasticity by programs running on the PPU during experiments. Therefore the PPU is also able to read out spiking rates and other additional information of synapses as they are needed for plasticity.

mention vector pipeline mention processor cycles. minimum 3 cycles per instruction This is realized through the same omnibus that connects the PPU to the memory and virtually the spiking times are accessible through a pointer with a special address.

The VE was added due to the need for fast handling and writing of the synaptic weights into the array of synaptic values on the HICANN. Hence the vector unit was equipped with an extra bus that connects to the mentioned synapse array. The basic processor does not have access to this bus and therefore must use the VE in order to communicate with the synapse array. On the other side both, the basic processor and the VE are connected to the same memory controller that manages 16 kiB of memory. This memory is used for data as well as the program itself and thus must be handled with some care as programs must not be larger than these 16 kiB. Having the VE and basic processor to share this memory controller without any caching can cause problems at times. An easy example is the sequential storing of a vector in memory followed by a loading instruction that requests the very same address of the vector in order to load one of its elements. This would lead to both instructions being in the wrong order as the vector pipeline delays the vector store instructions way to the memory controller by a few cycles ultimately reaching the memory controller after the load instruction which is passed on directly. But this not only happens for the VE but also can be caused by the compiler as it optimizes memory access mostly for performance. This may result in unintended reordering of memory operations which then leads to errors. Because of this there is the possibility to prohibit such reordering by introducing a memory barrier. A memory barrier basically is a line of code that splits the remaining code into code that occurs before the barrier and code that occurs after the barrier. It prohibits the compiler from mixing instructions from different sides to the barrier due to reordering. Typically such a memory barrier is called like `asm (:::memory)`, which the compiler recognizes as a memory barrier. On a processor level there exists also a memory barrier that would typically not be called by a user. It's called syncing and on PowerPC the instruction `sync` is used for this. Because the processor itself delays certain instructions to optimize performance (as part of its out-of-order architecture) it may happen that a dependency between two instructions is not detected. This would result in the very same problem as described earlier. Therefore the processor can be instructed to wait until all store and load instructions are executed. The memory controller then sends a signal to the processor that it is all done.

It is fairly obvious that only the second kind of memory barrier helps us in or cause to avoid wrong ordering in the memory controller. Luckily both the VE and the basic processor use the same memory controller which makes `sync` apply in both cases. Also since all vector instructions must go through the main processor first, any hold executing instructions on the MP affects the VE the same way.

1.3 Implementation in Hicann-DLS

Besides the VE and the MP, the memory also provides access to the FPGA through a different interface in order to allow for external access to the memory. This is needed for writing programs into the memory as well as getting results during or after execution. This also allows for communication during runtime of the PPU. Specifically the vector extension allows for either use of 8 element vectors with element 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, 32 in total, in contrast to 32 general purpose registers with 32 bit each. Also there is an accumulator featured as part of the vector extension which is used in every arithmetic operation and a condition register which holds 3 bits, that determine which condition applies, for every half byte or nibble, making 96 bit in total. The PPU also features 4 KiB of memory as well as access to the synapse array of the HICANN which holds up to 32x32, thus 1024 different values. To handle the vector unit the instruction set was extended by xz new instructions that partly share their opcodes with existing AltiVec instructions.

CURRENT STATE OF PROGRAMMING ON HICANN -> MOTIVATION main feature analogue spiking PPU can read spiking times size and alignment of weights read paper first 64 outputs 32 inputs to 32 neurons

check this

add special regs of normal processor part, CC, LR, etc.

count number of vector instructions.

theoretical gain in speed when using vector unit compared to normal instructions especially with single load and store. include PPU clock frequencies for each part in calculation.

2 Compiler Structure

As already hinted by the abstract, a compiler consists of a front-end, a back-end, but also a third part that is the middle-end. These three parts sit on top of each other with the front-end on the very top and the back-end at the bottom and pass down the program as it is translated and optimized or “compiled”.

The first part of the compilation process is the translation of code which is written in some programming language into a so called Immediate Representation (IR) that looks the same for every front-end language and usually is never seen by the user. Any supported programming language (C, C++, Java...) is implemented in its own front-end that defines how the language is translated into IR. After that the IR is send to the middle-end, which generally optimizes the IR and then passes the code to the back-end. The back-end first executes further optimizations that are target-specific followed by allocatiing registers and handling relative memory. Finally the code is translated into the assembly language that is supported by the target.

After the code is compiled and emmited as an object file it is also linked, which means combining different objectfiles and and assigning absolute memory addresses to tehml. At last the binary file emmited by the linker is loaded into the memory of the processor and then can be executed.

Most Compilers that are used nowadays are built of three basic components which handle different steps in the process of converting human-readable programming language to machine-readable machine code. As does the GNU Compiler collection (GCC) which also can be seen as made of three main parts. The so called front-end, middle-end and back-end. All three parts work more or less independently from each other and communicate over a compiler-specific „language“, which is described a the Intermediate Representation (IR). It is typically never seen by the user and exists for a fact in many different forms [reference] one of which is Register Transfer Language (RTL), which is the lowest-level IR used by GCC. It is the most interesting IR when working on a back-end and will get more attention later in this report. But in before that we need to understand the structure an functioning of a compiler in general.

The first mentioned Front-end resembles the main interaction point between the human programmer and the machine. Front-ends are usually divided by their respective programming languages such as C, C++, Java... and have the main task of converting any programming language into unified IR, that can be passed to the middle-end in GIMPLE or GENERIC language. Therefore no matter which language you prefer, in an ideal case code, that is syntactically identical, should not differ after it is processed by the front-end. This is due to the goal of compilers such as GCC and LLVM to support as many languages and machines as reasonably possible while offering the equally good optimization and saving themselves overall work. It is obvious that a single compiler for

every combination of language and machine would simply not be practical, especially as the optimization taking place in the middle-end follows the same rules for pretty much any architecture. The middle-end main task is basically this sort of optimization, and makes for the main difference between compilers as most compilers offer the same range of front-ends and back-ends (Part about the optimizations taking place in the middle-end). After all optimizations are through, the middle-end passes IR in form of RTL to the back-end. As you can see the middle-end rarely needs to be modified except for fundamental changes in the compilers architecture such as new kinds of optimizations and „multiple memory handling“ (also Harvard-Architecture (vielleicht)).

2.1 back-end

The back-end is responsible for the final steps of the compilation process as it translates the general RTL IR into specific Assembly commands. It uses some sort of table of available assembly instructions, that is provided, and finds the best fitting instructions. GCC for example uses a Lisp-like language (is this RTL?) that uses something called insns. These combine different properties with the Assembly commands like an equivalent set of actions that are executed, the operands and the constraints it must satisfy. These will be further explained later. The back-end also contains the code which implements processor-specific built-in functions. Depending on the Compiler architecture the back-end it finally emits IR to the assembler which emits the machine code in assembly or emits assembly itself. Then finally the linker links the assembly code of all program files together and substitutes the offset addresses with absolute addresses to generate the final machine code.

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

2.2 intrinsics

3 Extending the GCC Back-End

To allow the compiler to differ which insns we actually want to use as a built-in, the rs/6000 back-end has a special file which contains macros for every builtin function that will be available. These macros will be loaded 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 different 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 operands our built-in function is going to have and the number in the template name is meant to be equal to the number 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 two 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 to interact with the second string which is the new built-in's name for the front end. (for altivec built-ins the template adds `__builtin_altivec_` in front of the given name). Next the attribute for the builtin 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 one final builtin. here we first need a Macro name and second the function name we want to use later (this function name is preceded 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 naming 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 interesting case for our exemplary AltiVec builtin function. because there is already a `vec_mul` builtin we will choose the descriptive abbreviation `vec_smul` for scalar multiplication. The other naming convention gets interesting 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 general 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

4 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

5 Results

still compiles old code optimizes runs on PPU

6 Discussion

Discussion...

7 Outlook

Outlook...

Appendix

Notes

add reference	1
cite freidmann dissertation	1
check this	3
reference	4
reference	4
http://www.nxp.com/assets/documents/data/en/reference-manuals/ALTIVECPEM.pdf	4
vector register file ALU FPU memory memory controller = MMU clockcycle pipeline	5
preliminary paragraph with: physiology -> hicann -> PPU -> processor in general -> compiler -> backend methods in next chapter	5
is this word right?	5
put the following part in the next section	5
add something here	5
explain this further	5
add part about sram here	6
sram	6
PPU paper	6
reference	7
mention vector pipeline mention processor cycles. minimum 3 cycles per in- strucion This is realized through the same omnibus that connects the PPU to the memory and virtually the spiking times are accessible through a pointer with a special address.	7
check this	8
add special regs of normal pccessro part, CC, LR, etc.	8
count number of vector istructions.	8
theoretical gain in speed when using vector unit compared to normal instructions especially with single load and store. include PPU clock frequencies for each part in calculation.	8
8section*.27 different parts of an instruction, mention: opcodes, asm instruction, operation, operand, insn, IR, builtin function/intrinsic 10section*.28	
asm tests user level tests compiler tests	13

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 14, 2017

.....
(signature)