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.

mention ASIPs and the papers

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 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 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 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 ...., 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. 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 ... 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 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.

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. 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 ... . 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 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.

s.o.

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 `nop`s before every instruction that needs to wait for an operand as `nop` tells the processor to wait until the previous instruction has finished. For critical memory usage the compiler can also insert `sync`s as memory barriers before hazardous memory instructions. Alternatively it can stall only the instruction which depends on the result which is currently computed but perform instructions that do not depend on the result in the mean time. By doing so the scheduler increases performance noticeably and thus can partly be seen as part of the optimization process. On RISC architectures this is especially important as load and store instructions can take a few hundred times more clock cycles than normal register instructions and pipelining depends mainly on the instruction sequence. Thus the scheduler is also involved parallelization of code. As a result of this a compiler would usually accumulate all load instructions at he beginning of a procedure and start computing on registers that already have a value while the others

are still loaded. This is done vice versa at the end of a procedure for storing the results in memory. This process of course needs the compiler to know the amount of time it takes for an instruction to be executed and works hand in hand with hazard detection on processor level.

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

During and after code generation the compiler also applies optimizations to the machine code. Any optimization to the code though must take three things into consideration, which are safety, profitability and risk/effort. The first thing which always must be fulfilled, is safety. Only if the compiler can guarantee that an optimization does not change the result of the transformed code compared to the original code it may use this optimization! Only if this applies the compiler may check for the profit of an optimization which most often is a gain in performance but could also be the size of the program. At last the effort or time it takes for the compiler to perform this optimization and the risk of generating actually bad or 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 a named after the first letters of the instructions full name, which is emphasized in the following table. The operand can be of three different types which are all shown above. They either represent a specific register (r1 = register 1), a memory address (0x3000 = the value at memory location 0x3000) or an immediate value (5 = the integer 5). Register operands can also have an indirect use, which means that that content of the register is taken into account. 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.

mnemonic	operands	description
add	RT, RA, RB	**add** RB to RA and store the result in RT
addi	RT, RA, SI	**add** SI to RA and store the result in RT
addis	RT, RA, SI	**add** SI **s**hifted left by 16 bit to RA and store the result in RT
and	RA, RS, RB	RS and RB are **and**ed and the result is stored in RT
b	target_addr	**b**ranch to the code at target_addr
ble	BF, target_addr	**b**ranch to the code at target_addr if BF is **l**ess or **e**qual
blr		**b**ranch to the code at address in the **l**inker **r**egister
cmp	BF, L, RA, RB	RA and RB are **comp**ared and the result (gt,lt, eq) is stored in BF, L depicts if 32-bit or 64-bit are compared
cmplwi	BF, RA, SI	RA **comp**ared **l**ogically **w**ordwise with **i**mmediate SI and the result is stored in BF
and	RA, RS, RB	RS and RB are **and**ed and the result is stored in RT
eieio		RS and RB are **and**ed and the result is stored in RT dit this
isync		RS and RB are **and**ed and the result is stored in RT dit this
la	RT, D(RA)	**l**oad **a**ggregate D + RA into RT
li	RT, SI	**l**oad **i**mmediate value SI into RT
lis	RT, SI	**l**oad **i**mmediate value SI **s**hifted left by 16 bit into RT
lbz	RT, D(RA)	**l**oad **b**yte at address D+RA into RT, fill the other bits with **z**eros
lwz	RT, D(RA)	**l**oad **w**ord at address D+RA into RT, fill the other bits with **z**eros
mflr	RT	**m**ove **f**rom **l**inker **r**egister to RT
mr	RT, RA	**m**ove **r**egister RA to RT
nop		halts execution until the previous instructions are finished dit this
rlwinm	RA, RS, SH, MB, ME	**r**otate **l**eft **w**ord in RS by **i**mmediate SH bits then a**n**d with **m**ask which is 1 from MB+32 to ME+32 and 0 else, store to RA
stw	RS, D(RA)	**st**ore **w**ord from RS to address D+RA
stwu	RS, D(RA)	**st**ore **w**ord from RS to address D+RA and **u**pdate RA to D+RA
sync		halts execution until the memory controller is finished dit this

caption and reference to PPC book and asm website

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

i

indicates that the instructions uses an immediate value

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

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

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

`s`
: indicates that one of the operands is shifted

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

There are also special operands which might occur in `asm` which behave like pointers:

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

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

Additionally there exist markers which are intended for debugging:

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

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

`.LFB`
: marks the begin of a function

`.LFE`
: marks the end of a function

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

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):);
```

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 section:compiler 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. 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.

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 , which we will refer to as the sole source of information in this regard, the back-end of GCC has the following structure:

Each architecture has a directory with its respective name in gcc/config e.g. gcc/config/rs6000 that contains a minimum amount of files. These are the machine description rs6000.md which is an overview of machine instructions with additional information to each instruction and the header files rs6000.h and rs6000-protos.h and source file rs6000.c that handle the target description through macros and functions. Every back-end needs these files in the GCC source and the final back-end is build from these files through the macros and functions just mentioned. To notice a back-end in the first place the back-ends directory -here "rs6000"- must be added to the file config.gcc which also includes a list of all files in the aforementioned directory. Most back-ends include additional files which makes a back-ends complex structure clearer but these are not mandatory and we

will address these later.

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

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

Register Transfer Language Basics

RTL, which is not to be mixed up with Register Transfer Level, is a form of IR the back-end uses to generate machine code. Usually GCC uses the IR GIMPLE which looks like stripped down C code with 3 argument expressions, temporary variables and `goto` control structures. The back-end transforms this into a less readable IR that inherits GIMPLEs structure but brings it to a machine instruction level. It is inspired by Lisp lists and thus we will need to take a look at those at last in before we take on the task of extending a GCC back-end.

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

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

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

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

`define_insn` is an RTL expression that generates an RTL equivalent to a machine instruction. One such instruction is called an insn (short for instruction) hand has several properties like a name, an RTL template, a condition template, an output template and attributes. The name in this case is `add<mode>3` (`3` for three operands) where `<mode>` is to replaced by a set of values that describe a modes. 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. 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. 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. 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. 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 section:asm. `=` again means that the register must be writable and `v` stands for an AltiVec vector register.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.

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

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

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

reference the definitive guide to GCC introduction

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

reference

reference to GCC wiki

reference define_insn

reference standard names

reference machine modes

reference mode iterator

reference side effect expression

iterator and `<VI_char>` will be substituted by a character that belongs to a machine mode.

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.

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.

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

reference predicates

reference constraints

reference arithmetic expresssion

examples of RTL code

MMU ?= memory controller PPC must handle syncing in compiler when I/O is added

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

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 21, 2017

.....
(signature)