RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Arthur Heimbrecht

Internship report

# HD-KIP-TODOTODOTODO

KIRCHHOFF-INSTITUT FÜR PHYSIK

**Internship report**

As the BrainScaleS project aims to cover both neuroscientific and computational aspects it also incorporates a custom processor in the HICANN-DLS that handles the synaptic weights of neurons on a wafer, hence its name Plasticity Processor Unit (PPU). Because of the custom nature of the PPU it is not fully supported by any compiler right now so current users have to handle register allocation and memory structures on a regular basis which is uncommon for users mainly familiar with higher level languages. Therefore it is planned to extend the GCC back-end to support the PPU. The main part of this is the expansion with custom built-in functions, that any front-end or language is meant to support. These built-in functions then allow for a more comfortable use of directives that still enable the user to trigger certain actions in the PPU.

This report will analyze the way built-in functions are implemented in the rs/6000 or PowerPC back-end in order to check the feasibilty of such a back-end in the near future and to provide a first insight into the rs/6000 back-end in GCC. test

# Contents

# 1 Compiler Structure

Typically a compiler consists of a front-end, middle-end and back-end. These three parts sit on top of each other with the front-end on top and the back-end at the bottom and pass down the programs code as it is translated and optimized or compiled. But communication between the parts does not go only one way (at least for GCC)and changes that are made in the back-end affect the front-end as well! Such modular architecture allows for a compiler to support many different front-ends and back-ends independently because all code needs to pass the same middle-end no matter which front-end or back-end is used. In normal cases a front-ends are associated with programming languages and back-ends with CPU architectures or "targets" while the middle-end is at the heart of the compiler and is usually what distinguishes between different compilers, as does the communication between the three parts.

add at lest one nice structural picture of a compiler

add reference

The first part of the compilation process is the translation of code which is written in some programming language by the front-end into a so called Intermediate Representation (IR) that looks the same for every front-end language and usually is never seen by the user. AS already described, 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 where it is mainly optimized and then passed to the back-end. The back-end first performs further optimization that is target-specific followed by allocating registers and handling relative memory. Finally the code is translated into the assembly language that is supported by the target which the back-end belongs to. This last step executed by the back-end can seen as independent from the back-end as the translation into assembly is pretty straight forward, but the rs/6000 back-end for instance allows for direct assembly output that is not dependent on IR. Therefore we combine these the assembler and the back-end. After the code is compiled and emitted as an object file it is also linked, which means combining different object files and assigning absolute memory addresses to them and at last the symbols which are used in assembly are substituted by their respective opcodes. Opcodes are almost the lowest representation of machine code as the machine would not accept strings a input but rather bits that are represented as hex literals. These opcodes form pairs with assembly macros and can be found in binutils. Finally the resulting binary file is emitted by the linker and loaded into the memory of the processor to be executed.

GCC generally obeys this scheme and therefore the PowerPC architecture (also called rs/6000) is a good example of a back-end that supports different extensions to the standard architecture such as the AltiVec vector extension. This makes is more complex than the Cell Synergistic Processor Units (SPU) back-end but all back-ends in GCC share the same functions defined by macros. Still besides those macros the back-ends obey only few restrictions by the middle-end and thus this guide will not easily be transferable to

other GCC back-ends and especially not to other compilers back-ends!

# 2 Creating an intrinsic function

An intrinsic or builtin function is a medium link between inline assembly and normal functions. They look like normal functions but usually trigger certain machine instructions on a very basic level. This can make intrinsic functions very effective at the cost of being very specific compared to normal functions. Intrinsic functions are usually directly built into the compiler or rather its back-end and provide the compiler with additional information that allows for highly optimized code. Usually there is one specific machine instruction at the core of every intrinsic function. Most of this is done in the machine description that builds an interface between machine instructions and IR. Every instruction that is used within the compiler is part of a so called insn (short for instruction). An insn is an expression that is identified by its code and adds information to a machine instruction. It is written in Register Transfer Language (RTL) that has its own syntax and keywords. We will not dive further into this specific topic since these details are not needed for the purpose of this report. Therefore we will utilize existing insns and see what makes them complete intrinsic functions.

The functions we will take a look at are called `vec_addc` or `vec_vaddcuw` when used in any front-end. This is one of the differences between normal functions and intrinsics as the latter one is the same for every front-end because it is implemented into the back-end. This function, which takes two vectors that have unsigned int elements as arguments, "returns a vector containing the carries produced by adding each set of corresponding elements of two given vectors" , this means for `c = vec_addc(a,b)` that the resulting vector `c`'s elements' bits are 1 if adding `a` and `b` produces a carry at that bit position and `0` otherwise. Therefore the name "Vector ADD Carry Unsigned Wordsize". But the use of this function is less important than the way it is implemented.The alternative acronym vaddcuw depicts the name of the instruction on a processor level which is "`vaddcuw % c,%a,%b`" and is implemented as a synonym intrinsic to `vec_addc`. Hence `vaddcuw` and `vec_addc` have their specific insn with the code `altivec_vaddcuw`.

https://www.ibm.com/support/knowledge-center/ SSGH2K_13.1.3/com.ibm.xlc1313.aix.doc/compiler_ref/vec_addc.html

```
(define_insn "altivec_vaddcuw"
[(set
        (match_operand:V4SI 0 "register_operand" "=v")
        (unspec:V4SI [(match_operand:V4SI 1 "register_operand" "v")
                      (match_operand:V4SI 2 "register_operand" "v")]
                          UNSPEC_VADDCUW))]
"VECTOR_UNIT_ALTIVEC_P␣(V4SImode)"
"vaddcuw␣%0,%1,%2"
[(set_attr "type" "vecsimple")])
```

To explain this in a few words: the first line depicts the insns name `altivec_vaddcuw`, the next four lines specify what the instruction does (in this case an unspecified function this two input operands named `UNPEC_VADDCUW`; there are other insns that consost of known RTL Templates such as `plus`, `minus` ...) and which constriants and predicates

the operands (0,1 and 2) need to fulfill (this is RTL) (A constraint in this regard looks similar to constriants in assembly and has quite the same task of specifiying what kind of operand (memory address, register...) is allowed. A predicate specifies additional, more detailed restrictions for the operand). It follows a boolean function, to check if this insn is valid to be used and the assembly symbol with its operands. The attributes set at the end are only used internally.

Now our first step is to add an entry into the builtin description file rs6000-builtin.def. This file holds all builtin functions and connects them to insns. The very beginning of this file consists of convenience macros for different extensions that allow for better readability as most of the properties of a builtin function are similar or even the same for each extension.

```
#define BU_ALTIVEC_1(ENUM, NAME, ATTR, ICODE)                          \
  RS6000_BUILTIN_1 (ALTIVEC_BUILTIN_ ## ENUM,          /* ENUM */      \
                    "__builtin_altivec_" NAME,         /* NAME */      \
                    RS6000_BTM_ALTIVEC,                /* MASK */      \
                    (RS6000_BTC_ ## ATTR               /* ATTR */      \
                     | RS6000_BTC_UNARY),                              \
                    CODE_FOR_ ## ICODE)                /* ICODE */
```

There are different types of macros for different types of builtin functions. Each macro takes four arguments besides it's enumeration name which are:

- the name of the function as string literal
- a bit-mask that indicates which options are enabled
- attribute information
- the insn code

The macros then add information for each extension and divide the builtin functions into certain groups which depend on the number of arguments. In our case the macros are called `BU_ALTIVEC_1`, `BU_ALTIVEC_2`, `BU_ALTIVEC_3` for up to three arguments respectively and all of these have the `RS6000_BTM_ALTIVEC` bit-mask as well as the same prefix for their enumeration name and function name. These are `ALTIVEC_BUILTIN_` for the enumeration name and `__builtin_altivec_` for the function name. Besides that each macro has a specific attribute such as `RS6000_BTC_TERNARY` for function with three arguments.

Now we finally move to specifying the builtin function. The line of code we are interested in is:

```
BU_ALTIVEC_2 (VADDCUW, "vaddcuw", CONST, altivec_vaddcuw)
```

First the name for the enumeration is set as `ALTIVEC_BUILTIN_VADDCUW`, which is used in later code when the builtin is referenced to, then the builtin functions name is set as `__builtin_altivec_vaddcuw`. Next the attribute is set as `CONST` which means that no other registers are altered when the insn is used but the three registers that are directly called. At last the insn code is given as `CODE_FOR_altivec_vaddcuw`, which references the insn code by its name, that we saw earlier, in later code. This already gives us a usable builtin function! To use it we first set our vector variables:

```
vector unsigned int a,b,c;
c = __builtin_altivec_vaddcuw(a, b);
```

But this function still has some flaws as it would not give an error for this case:

```
short a,b,c;
c = __builtin_altivec_vaddcuw(a, b);
```

Because there is no type-checking the user could use the function in a completely wrong manner. To avoid this though, there are overloaded builtin functions that include a type-checking routine.

An overloaded builtin function is basically just another builtin function that is less specific than the previous function as it only specifies two names and no insn code. Thus they have their own macros (`BU_ALTIVEC_OVERLOADED_1`, `BU_ALTIVEC_OVERLOADED_2`, `BU_ALTIVEC_OVERLOADED_3`) and differ in a way that the enumeration prefix is `ALTIVEC_BUILTIN_VEC_` and the function name prefix is `__builtin_vec_` also the attributes are completely set in advance as well as the specific attributes. For an overloaded builtin we will use a simpler name which will be `__builtin_vec_addc`

```
BU_ALTIVEC_OVERLOAD_2 (ADDC, "addc")
```

Since there is no insn code given all overloaded builtins are connected to existing builtins which we will do next in rs6000-c.c.

We will overload the builtin `__builtin_vec_addc` with `__builtin_altivec_vaddcuw` and add argument and return types. In principle this is similar to a functions argument types but for intrinsics these are declared in a `struct` that allows for different combinations of argument and return types. The `struct` is called `altivec_builtin_types` and consists of an overloaded builtin code, a normal builtin code, the return type and up to three argument types. For `ADDC` exists only one `struct` though because it only works for vectors of unsigned `ints` (`V4SI` = Vector of 4 Single Integers):

```
{ ALTIVEC_BUILTIN_VEC_ADDC, ALTIVEC_BUILTIN_VADDCUW,
RS6000_BTI_unsigned_V4SI, RS6000_BTI_unsigned_V4SI, RS6000_BTI_unsigned_V4SI, 0 }
```

`ALTIVEC_BUILTIN_VEC_ADDC` is now overloaded with the working builtin function `ALTIVEC_BUILTIN_VADDCUW` from earlier and has a return type and argument types which are vectors of unsigned `ints`. The last entry is 0 because there is no third argument.

Basically this is enough for the overloaded builtin function to work properly and it can be used in a way such as

```
vector unsigned int a,b,c;
c = __builtin_vec_addc(a, b);
```

and would give an error if the types would not match those we set earlier.

To give all of this a nice touch and increase usability in the end. We define synonyms for our newly created overloaded builtin function. We will not do this for the original builtin function since we want to avoid the missing type checking.

```
#define vec_vaddcuw vec_addc
...
#define vec_addc __builtin_vec_addc
```

Here the first line defines a synonym for the function for people familiar with the assembly macro. This brings our task of defining a builtin function to an end!

But there is still one kind of common builtin function left that differs to normal one-to-three-argument-builtins in some ways such as requiring a memory address for assembly macro instead of a register or simply not having a return value. These are called special builtin functions that have the convenience macros `BU_ALTIVEC_X` and `BU_ALTIVEC_OVERLOADED_X`, though a special macro can also be overloaded with a normal overload macro like `BU_ALTIVEC_OVERLOADED_2`. The special X-macro has `CODE_FOR_nothing` as insn code like the overloaded macros did earlier and thus is not intended to be handled normally but will be caught in the main file `rs6000.c` which we will see later.

We will have an example that uses a normal overloaded macro since it is slightly easier and special overloaded functions tend to need special handling in the main back-end file because they have very specific properties. Therefore we take a look at `vec_mtvscr(a)` which copies the value of `a` into the Vector status and Control Register (`VSCR`) (Move To `VSCR`). The insn code for this builtin function is `altivec_mtvscr` and the machine instruction is `mtvscr %a`. It is obvious that this function does not generate any return value and therefore would not fit a one-argument-builtin.

```
BU_ALTIVEC_X (MTVSCR, "mtvscr", MISC)
...
BU_ALTIVEC_OVERLOAD_1 (MTVSCR, "mtvscr")
```

The other difference for this builtin is that it is not a `CONST` builtin but carries a `MISC` attribute. This argument is only used in special cases that make an exception to `CONST` or any of the other special attributes and means specifically that there are no special attributes. We will not discuss the other attributes but an explanation can be found in the `rs6000.h` file. In contrast to the builtin function `__biultin_altivec_vaddcuw` from earlier, the builtin function `__builtin_altivec_mtvscr` is of no use now since there is no insn code connected with it. Thus we will add special cases in the function that handles the builtin functions or “expands“ them. This is done in the `altivec_expand_builtin` function that handles special builtins exclusively. Normal builtins are expanded depending on their number of arguments at the very end of `rs6000_expand_builtin` where `altivec_expand_bulitin` is called before hand.

In the expander function the compiler switches between all special cases, which means there has to be an entry for every special builtin there is. For `mtvsrc` this entry looks something like:

```
case ALTIVEC_BUILTIN_MTVSCR:
      icode = CODE_FOR_altivec_mtvscr;
      arg0 = CALL_EXPR_ARG (exp, 0);
      op0 = expand_normal (arg0);
      mode0 = insn_data[icode].operand[0].mode;

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

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

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

This code is probably the most difficult part when adding a special builtin function. The easiest way is to look for a similar function, copy its code and modify it if necessary. But we will go through this code briefly: First we see some important variables that get their respective values. `icode` obviously holds the insn code, `arg0` holds whatever the function gets as first argument, `op0` makes an operand of that argument, and `mode` holds the mode of the operand that the insn needs (modes are for example Single Integer, Vector of 16 Quarter Integers etc.. It then checks if the argument is actually valid and returns an error otherwise. Next it checks whether `mode` and `op0` match and tries to convert the operand if they do not match (it tries to match the proveded operand and requested mode to the predicate for that operand in the insn). `pat` holds the directive to build an insn with code `icode` and operand `op0` and if this gives no error the final insn is emitted. The return value has no purpose but detecting errors and thus is `NULL_RTX`.

Now the compiler knows the insn code of this special insn but it needs to define the builtin as well. For special builtin functions this is not done automatically but in `altivec_init_builtins`:

```
def_builtin ("__builtin_altivec_mtvscr", void_ftype_v4si, ALTIVEC_BUILTIN_MTVSCR);
```

This adds `__builtin_altivec_mtvscr` to the list of defined functions and also gives the argument and return types (`void_ftype_v4si`, everything before `ftype` is the return type everything after the arguments). In this case it is not obvious why this needs to be done but there do exist builtins that have different insn codes depending on the used modes thus the type of the input arguments helps distinguish these differences. In this case only `v4si` is chosen as mode because we will also have an overloaded builtin for this new builtin function. This is done similar to earlier by adding entries in `rs6000-c.c` for each combination of arguments and return types:

```
{ ALTIVEC_BUILTIN_VEC_MTVSCR, ALTIVEC_BUILTIN_MTVSCR,
  RS6000_BTI_void, RS6000_BTI_V4SI, 0, 0 },
{ ALTIVEC_BUILTIN_VEC_MTVSCR, ALTIVEC_BUILTIN_MTVSCR,
  RS6000_BTI_void, RS6000_BTI_unsigned_V4SI, 0, 0 },
{ ALTIVEC_BUILTIN_VEC_MTVSCR, ALTIVEC_BUILTIN_MTVSCR,
  RS6000_BTI_void, RS6000_BTI_bool_V4SI, 0, 0 },
{ ALTIVEC_BUILTIN_VEC_MTVSCR, ALTIVEC_BUILTIN_MTVSCR,
  RS6000_BTI_void, RS6000_BTI_V8HI, 0, 0 },
{ ALTIVEC_BUILTIN_VEC_MTVSCR, ALTIVEC_BUILTIN_MTVSCR,
  RS6000_BTI_void, RS6000_BTI_unsigned_V8HI, 0, 0 },
{ ALTIVEC_BUILTIN_VEC_MTVSCR, ALTIVEC_BUILTIN_MTVSCR,
  RS6000_BTI_void, RS6000_BTI_bool_V8HI, 0, 0 },
{ ALTIVEC_BUILTIN_VEC_MTVSCR, ALTIVEC_BUILTIN_MTVSCR,
  RS6000_BTI_void, RS6000_BTI_pixel_V8HI, 0, 0 },
{ ALTIVEC_BUILTIN_VEC_MTVSCR, ALTIVEC_BUILTIN_MTVSCR,
  RS6000_BTI_void, RS6000_BTI_V16QI, 0, 0 },
{ ALTIVEC_BUILTIN_VEC_MTVSCR, ALTIVEC_BUILTIN_MTVSCR,
  RS6000_BTI_void, RS6000_BTI_unsigned_V16QI, 0, 0 },
{ ALTIVEC_BUILTIN_VEC_MTVSCR, ALTIVEC_BUILTIN_MTVSCR,
  RS6000_BTI_void, RS6000_BTI_bool_V16QI, 0, 0 }
```

The return type obviously should be the same for all entries since there is no return type thus `RS6000_BTI_void` as first entry. Next there are 3 modes with different submodes, because all integer vector modes are allowed as first argument. A normal mode means that the elements are signed integers and an unsigned mode has unsigned elements. `Bool`

elements have a single `bool` variable at each element and pixel is used for graphic usage of the AltiVec extension. The `0` at the end marks the second and third argument unused. This sets the last step to completing our special builtin function that has a normal overloaded part.

At last we define a shorter function name in `s2pp.h`:

```
#define vec_mtvscr __builtin_vec_mtvscr
```

This completes our special intrinsic function!

For implementing a builtin function for earlier GCC versions (4.4 and earlier) I highly recommend the guide by Mauricio Alvarez

(http://people.ac.upc.edu/alvarez/media/ gcc-isa-extensions.html)

# 3 Discussion

This report had the goal of explaining the implementation of AltiVec intrinsic functions. It started straight forward with explaining the basic structure of compilers such as GCC and emphasized on the rs/6000 back-end. This followed by a step-by-step guide on how intrinsic functions are build bottom-up and added the case of special builtins. The value of intrinsic functions can be questioned when compared to inline functions and inline assembly but in any way intrinsics do allow calling specific actions on the processor that are otherwise not available while also allowing for optimization by the compiler. This generates the perfect use case for extensions such as AltiVec where different versions of add instruction are crucial to the usefulness of such an extension. Intrinsics therefore provide the common user to take control over the Processing Unit in a simple way that neither requires knowledge of assembly nor forces the user to produce efficient code by himself.

This is contrary to the current state of PPU support which demands assigning registers by hand as well as choosing the right memory address. In comparison this is more complicated and prone to inefficiency when done by an unexperienced user especially for large programs.

As it is obvious that this report does not help a lot in understanding how a compiler back-end completely works but rather is a tutorial on how to add intrinsic functions to the rs/6000 back-end it is meant to rather advertise the future use of implementation of intrinsic functions for the PPU. Though it also did not show all possibilities there are for intrinsics, most of these are barely needed and it would require further knowledge of a back-end. This goes beyond what is needed in case of the reduced instruction set of the PPU. Ultimately this guide may help the most when there is an instruction set to be added to the rs/6000 back-end and to give an overview of what may be possible to include in a GCC back-end. We leave further explanation of the internals of GCC to the official GCC internals handbook as it provides a more detailed look into RTL and insn defintion.

add link

# 4 Outlook

As it was hinted in the discussion of this report, there are plans on extending the rs/6000 back-end of GCC so it supports the PPU's vector unit This is surrently in the works currently as there are even some PPU-specific intrinsics already. Because the PPU is a simple PowerISA chip with a custom vector unit the implementation of intrinsics is easily transferable from the AltiVec vector extension to the PPU by swapping the `altivec` keyword with `s2pp`. With some knowledge of RTL this report then allows to add new composite intrinsic functions based on the instruction set of the PPU. By then this guide should be accompanied by a more complex thesis that explains the extension of the rs/6000 back-end and is meant to provide the reader with more insight into things like RTL and maybe makes reading the GCC internal obsolete.

# Appendix

## 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, January 24, 2017

.....
(signature)