RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Arthur Heimbrecht

Internship report

HD-KIP-TODOTODOTODO

KIRCHHOFF-INSTITUT FÜR PHYSIK

Internship report

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 backend of GCC, specifically the one used by the PPU, in order to extend the already implemented set of functions according to the users needs.

As the BrainScaleS project has many facettes, it also incorporates a custom processor in the HICANN-DLS that handles the synaptic weights of neurons on a wafer, hence Plasticity Processor Unit (PPU). Because of the custom nature of the PPU it is not supported by any known compiler and current users have to handle register allocation and memory structures on a standard basis which is uncommon for users mainly familiar with front-end langugaes. Therefore it is planned to extend the GCC back-end to support the PPU. Part of this is the Expansion with custom built-in functions, that any front-end 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.

Contents

1 Compiler Structure

This part explains the structure of a compiler only on a very shallow level. Therefore the reader is encouraged to read further literature as the knowledge of a compilers structure helps a lot in understanding the way a buillt-in functions works and what problems might occur.

As already hinted by the abstract, a compiler consists of a front-end and 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 top and the back-end at the bottom and pass down the program as it is translated and optimized or compiled. But communication between the parts does not go only one way and changes that are made in the back-end affect the front-end as well! 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 tehm. At last the binary file emmited by the linker is loaded into the memory of the processor and then can be executed.

2 Creating a built-in function

The very basic level of creating your own built-in function is coding an insn that is connected to a basic operation the processor can handle. This report will not go into great detail about this as it is substantially different from the other sections in this report and requires more insight into Register Transfer Language than this report could supply or demand. Therefore we will take a look at an insn that already exists and analyze the different steps it takes towards a complete built-in function. The function we will look at is called `vec_addc` or `vec_vaddcuw` when used in a programming language. This function, which takes two vectors that have unsigned int elements as input, "returns a vector containing the carries produced by adding each set of corresponding elements of two given vectors" (https://www.ibm.com/support/knowledgecenter/SSGH2K_-13.1.3/com.ibm.xlc1313.aix.doc/compiler_ref/vec_addc.html), 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 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 acronym vaddcuw depicts also the name of the function on a processor level which is "`vaddcuw  %c,%a,%b`" in assembly. The interface to any assembly function in GCC is a machine description file (*.md) that connects assembly macros with their properties and also assigns them a name that can be used later on. In our case the then insn can be found under "`altivec_vaddcuw`".

```
(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")])
```

There will be no explanatoin of the Register Transfer Language (RTL) that describes an insn since this is not needed in the course of making this insn a built-in function. The only important information we need really need is the number of arguments the insn takes as this is equivalent to the number of arguments the function itself will take and the code/name of the insn. In this case the code is "`altivec_vaddcuw`" and is takes two arguments. The next step is to add an entry into the builtin-description-file rs6000-builtins.def. The very beginning of this file consists of convenience macros that allow for better readability as most of the properties of a builtin function are similar or even the same. First there are the processors macros, that are divided into different groups depending on their properties such as the number of arguments. In our case we focus on those that take 1 to three argumetns and also the special builtins as these are the

most used builtins. Each macro takes the same four arguments besides it's enumeration name. These are: The name of the function as string literal, a bit-mask that indicates which options are enabled, attribute information and the insn code. The macros then go even further as there are seperate builtin macros for different extensions. In our case the macros are called `BU_ALTIVEC_1`, `BU_ALTIVEC_2`, `BU_ALTIVEC_3` all of these have the `RS6000_BTM_ALTIVEC` bitmask and 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. Besides those builtin macros there exist also "overloaded" macros named `BU_ALTIVEC_OVERLOADED_1`, `BU_ALTIVEC_OVERLOADED_2`, `BU_ALTIVEC_OVERLOADED_3`. We will nedd those later but they 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 such as the specific attribute from earlier and the `RS6000_BTC_OVERLOADED` attribute. 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` then the builtin functions name is set as `__builtin_altivec_vaddcuw`. Next the attribute is set as `CONST` which means that there are no other registers altered when the insn is used but the 3 registers that are directly used. At last the insn code is given as `altivec_vaddcuw`, which we know from earlier. 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 typechecking a user could use function in a completely wrong manner. To avoid this though, there are overloaded builtin functions that include a typechecking routine. An overloded 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. For our overloaded builtin we will use a simpler name which will be `__builtin_vec_addc`

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

Now we need to overload the builtin `__builtin_vec_addc` and add argument and return types. In principle this is similar to a functions argument types but 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:

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

This connects the overloaded builtin ALTIVEC_BUILTIN_VEC_ADDC with the working builtin function ALTIVEC_BUILTIN_VADDCUW from earlier defines a return type which is the same as the argument types a vector of unsigned ints. The last entry is 0 because there is no third element.

Basically this is enough for the overloaded builtin function to work properly and it can be used under the name 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 orginal builtin function since we 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 can differ to normal one-to-three-argument-builtins in many ways such as requiring a memory address for assembly macro instead of a register or simply not having a return value. These assembly macros qualify as 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 in a way that it has the insn code CODE_FOR_nothing like the overloaded macros and the macros are not intended to handled normally but will be catched in the main file rs6000.c which we will see later. We will have an example that uses a normal overloded macro since it is slightly easier and special overloded functions tend to need special handling in the main back-end file. Thus 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 assembly macro is mtvscr %a. It is obvious that this function does not generate any return value and therefore not fit a one-argument-builtin.

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

The other difference this builtin has 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 discuss the other attributes but a 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 since there is no insn code connected with it. Thus we will add special cases in the function that handles the builtin functons 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;
```

These lines are 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 nececssary. but we will go through this code briefly: First we see some important varibales that get their respective values. `icode` obviouly holds the insn code, `arg0` holds whatever the function gets as first argument, `op0` gets the operand of that argument, and `mode` holds the mode of the operand that the insn needs. It then checks if the argument is acutally 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. `pat` gets to hold the directive to bulid an insn with code `icode` and operand `op0` and if this give no error the final insn is emmited. 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 non-normal 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 fore bulitins that have different insn codes depending on the used modes thus the type of the input arguments this distinguishes these differences. In this case only `v4si` is chosen as mode because we will also have an overload for this new builtin function. This is done as awe did before 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 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. 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. This sets the last step to completing our special bulitin function that has a normal overloaded part. At last add shorter function name in s2pp.h:

```
#define vec_mtvscr __builtin_vec_mtvscr
```

For implementing a builtin function for earlier GCC versions (3.8-) I highly recommend the guide by ...

3 Discussion

Discussion...

4 Outlook

Outlook...

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 3, 2017

.....
(signature)