Andes Custom Extension™ Programmer's Manual

Copyright Notice

Contact Information

Should you have any problems with the information contained herein, you may contact Andes Technology Corporation through

- Email – support@andestech.com
- Website – https://es.andestech.com/eservice/

Please include the following information in your inquiries:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem

General suggestions for improvements are welcome.

Revision History

Rev.	Revision Date	Revised Content
1.0	2024/09/01	First release

Table of Contents

COPYRIGHT NOTICE ..... **I**

CONTACT INFORMATION ..... **I**

REVISION HISTORY ..... **II**

1. INTRODUCTION **1**

2. ACE DESCRIPTION SCRIPT **2**

2.1. OPERAND (OPERAND) 3

2.2. VECTOR MASK (VECTOR_MASK) 3

2.3. CUSTOM C SIMULATION MODEL (CSIM) 4

2.3.1. ELEN and SEW *5*

2.3.2. Data Type for Vector Element *5*

2.3.3. Softfloat Data Types *6*

2.4. OPERATION LATENCY (LATENCY) 7

2.5. CYCLE PER RESULT (CYCLE_PER_RESULT) 7

3. C/C++ PROGRAMMING **8**

3.1. HELPER FUNCTIONS FOR ACE INSTRUCTIONS 8

Document Element	Font	Font Style	Size	Color
Normal text	Georgia	Normal	12	Black
VARIABLES OR PARAMETERS IN COMMAND LINE, SOURCE CODE OR FILE PATHS	Courier New	Normal	12	Blue
Hyperlink	Georgia	Underlined	12	Blue

1. Introduction

Andes Custom Extension (ACE) empowers RISC-V "V" Vector Extension (RVV) with advanced customization capabilities when integrated with AndesCore Vector processors, denoted as ACE-RVV.

ACE-RVV boosts computational performance by offering advanced customization options, supporting a range of data types including vector, floating-point, and integer. This enhancement is particularly beneficial for high-computational applications such as Artificial Intelligence (AI), where it optimizes processing capabilities.

Section 2 describes the syntax and semantics of the ACE description script, explaining how it can be used to tailor instructions.
Section 3 offers the C/C++ programming guide, complemented by ACE helper functions to streamline development.

For more details, the RISC-V "V" Vector Extension Specification 1.0 document is available at: https://github.com/riscv/riscv-v-spec.

2. ACE Description Script

This section describes the syntax and semantics of the ACE description script, using the `exp` instruction as an example to illustrate the detailed syntax of instructions.

```
rvv_insn exp{
  operand = {out vr:fp result, in vr:fp x};
  vector_mask = none;

  csim = %{
    unsigned csr_vtype_vsew = (csr_vtype >> 3) & 0x7;
    switch(csr_vtype_vsew) {
      //sew = 32
      case 2: {
        //elen = 64, each elements has 2 f32 points
        for(unsigned int j = 0; j < 2; j++) {

          float32_t flt_x = x.f32[j];
          flt_x = (flt_x > (float32_t)EXP_F32_MIN_FP32)? flt_x : (float32_t)EXP_F32_MIN_FP32;
          flt_x = (flt_x < (float32_t)EXP_F32_MAX_FP32)? flt_x : (float32_t)EXP_F32_MAX_FP32;

          u_hex_fp32_t v;
          v.hex = (1<<23)*(1.4426950409*flt_x+126.94201519f);

          result.f32[j] = v.flt;
        }
        break;
      }
      default:{
        cout<<"ERROR:Unsupported SEW value!"<<std::endl;
        assert(0);
      }
    }
  %};

  latency = 6;
  cpr = 1;
};
```

In this example:

- An ACE-RVV instruction named `exp` is presented with the syntax: `rvv_insn exp`.
- `operand` specifies the inputs and outputs of the instruction. See Section 2.1 for details.
- `vector_mask` specifies how the mask is used in RVV instructions. See Section 2.2 for details.
- The semantic function, represented by `csim`, explains how the instruction behaves or is executed. See Section 2.3 for a detailed description.
- `latency` specifies the instruction latency. See Section 2.4 for details.
- `cycle_per_result`, abbreviated as `cpr`, represents the number of cycles needed to generate output data. See Section 2.5 for details.

NOTE: For the Hackathon competition, you should modify `csim` with an enhanced algorithm for the exponential function. Additionally, adjust the `latency` and `cpr` attributes to align with a feasible hardware design tailored to your improved algorithm.

2.1. Operand (operand)

The `operand` attribute, abbreviated as `op`, specifies operands and their types. The operands are separated by commas (,) within the list.

- `IO_TYPE` specifies the input (`in`) or output (`out`) type of an operand and can take on one of the following values:

`in`	Input
`out`	Output
`io`	Input and Output

- `OPERAND_TYPE` specifies the source type of an operand and can take on the following value:

`vrf:int`	Vector register, integer type
`vrf:2int`	Vector register, integer type, double EEW's size
`vrf:uint`	Vector register, unsigned integer type
`vrf:2uint`	Vector register, unsigned integer type, double EEW's size
`vrf:fp`	Vector register, floating point type
`vrf:2fp`	Vector register, floating point type, double EEW's size
`vrf:mask`	Vector register, mask type
`xrf`	General-purpose register, input only

In the `exp` instruction, the input is represented by a floating-point type vector register named `x` (notated as `in vr:fp x`), and the output is a floating-point type vector register named `result` (notated as `out vr:fp result`).

2.2. Vector Mask (vector_mask)

The `vector_mask` attribute specifies how the `v0t` and `vm` are used, determining whether a

masked version of an instruction is available. Legal options include `out` (default), `in`, `io`, and `none`.

The following table lists the functions supported by each option.

	Mask			Non-Mask
	`out`	`in`	`io`	`none`
Insert an operand of `imm1 vm` into operand list	O	O	O	X
Generate masked version of ACE helper functions	O	O	O	X
Use the Mask value (`v0t`) to mask off output elements	O	X	O	X
Use the Mask value (`v0t`) to compute output results	X	O	O	X

O: Supported, X: Unsupported

For the `exp` instruction, the `vector_mask` is set to `none`, indicating that no mask is used with this instruction.

2.3. Custom C Simulation Model (csim)

The semantic function is defined by the `csim` attribute, which can be classified into two types: global `csim` and instruction-level `csim`. Global `csim` attributes are used outside `rvv_insn` statements, while instruction-level `csim` attributes are used inside `rvv_insn` statements.

Global `csim` attributes facilitate programming, debugging, and performance analysis by allowing the declaration of global variables shared across multiple instruction-level `csim` attributes, as well as defining functions invoked within them. However, they are not recommended for actual instruction functionality.

Instruction-level `csim` attributes describe the semantics of an instruction. It can contain:

- Pure C/C++ statements
- Calls to user-defined C/C++ functions in the global `csim`
- Calls to ACE built-in functions for controlling the processor

In a csim attribute, instruction operands defined in the operand attribute can be used directly as operation operands while additional variables must be declared before they are used. The C/C++ codes in a csim attribute are used for simulating the instruction in the instruction set simulator.

In the exp instruction, a reference algorithm is provided for calculating the exponential function. You can replace the code in csim with your own enhanced exponential algorithm.

The following sub-sections provide detailed information on how to modify csim.

2.3.1. ELEN and SEW

ELEN stands for the element size, while SEW denotes the selected element width, a standard terminology used in the RISC-V V-extension standard.

In the exp instruction, ELEN=64 indicates that the design needs to be implemented only for 64-bit data. The ACE framework will then automatically extend the 64-bit design to match the VLEN width, which is 512 bits in this example.

SEW=32, represented by CSR vtype[5:3], indicates that the selected element width for the exp instruction is 32-bit. Both the input and output operands of this instruction are floating-point, meaning the data type is FP32.

2.3.2. Data Type for Vector Element

The computation model uses a union data type, rvv_elen_t, to encapsulate all data within a SEW. This union data type supports normal operations and encompasses all data within an ELEN. Table 1 provides a detailed breakdown of the elements included in this union data type.

Table 1: List of Elements Within Union Data Type

```
union rvv_elen_t {
  uint64_t u64;
```

```
  int64_t i64;
  uint32_t u32[2];
  int32_t i32[2];
  uint16_t u16[4];
  int16_t i16[4];
  uint8_t u8[8];
  int8_t i8[8];
  float64_t f64;
  float32_t f32[2];
  float16_t f16[4];
}
```

To access individual members within a union, use the member access operator (.). For example, in the `exp` instruction, when SEW is 32 and the data type is floating-point, `x.f32[0]` and `x.f32[1]` correspond to the first and second components, respectively, of the input vector operand `x`.

2.3.3. Softfloat Data Types

To support floating-point operations in CSIM, a softfloat library is provided. The softfloat library includes three data types: `float16_t`, `float32_t`, and `float64_t`, which correspond to the IEEE754 standard for 16-bit, 32-bit, and 64-bit floating-point formats, respectively.

In addition to fundamental operators like addition and subtraction, the softfloat library includes a built-in function to support multiply-add (MAC) operations. The prototype of the MAC operator is as follows:

```
result = ace_fp_mul_add(x, y, z)
```

where `x`, `y`, and `z` serve as inputs, while `result` denotes the output. It is imperative to note that all inputs and the output must share the same floating-point data type, such as `float32_t`. In

other words, the function executes the following computation:

```
result = x * y + z
```

2.4. Operation Latency (latency)

The `latency` attribute, which is optional, specifies the instruction latency with a default value of 1 and a maximum value of 9999.

This attribute can be expressed either as a numerical value in decimal format or as any valid C/C++ expression. The instruction set simulator uses this value to estimate the cycle count.

The `latency` attribute shares the same context or variable scope with the `csim` attribute, allowing for the computation of complex latency values within `csim` that can be referenced in the `latency` attribute.

For the `exp` instruction, the latency is set to assume that the exponential algorithm completes in 6 cycles.

2.5. Cycle Per Result (cycle_per_result)

In ACE-RVV instructions, the `cycle_per_result` attribute, abbreviated as `cpr`, specifies the number of cycles required to generate each output data point. This attribute eliminates fractional numbers in cycle calculations (e.g., 1/3 = 0.3333...).

The valid range for this `cpr` attribute extends from 1 to the value specified by the `latency` attribute, with the default being the latency value itself.

For the `exp` instruction, it is assumed that subsequent results are ready in every cycle following the generation of the first result.

3. C/C++ Programming

COPILOT generates the ACE header file, `ace_user.h`, which is used in the C application. This header file includes definitions for custom data types, their utility functions, and the helper functions for custom instructions.

3.1. Helper Functions for ACE Instructions

Helper functions are C functions for invoking the implemented ACE instructions.

Helper function names start with the `ace_` prefix, followed by the name of an instruction. The prototypes for these helper functions are defined in `ace_user.h`, a file generated by COPILOT located in the `include` subdirectory of your working directory.

All C source files calling these helper functions should include the `ace_user.h` file. The order of arguments in a helper function matches the order of operands as declared in the `operand` attribute, and the actual prototype varies depending on the operand types.

It is recommended to carefully review the prototype declarations in `ace_user.h` to ensure that the arguments of a helper function meet your expectations, particularly for custom instructions that use advanced operand types.

COPILOT, similar to RVV intrinsic functions, generates a range of helper functions for ACE-RVV instructions, accommodating various LMULs and SEWs. Typically, both standard RVV and ACE-RVV instructions support different SEWs. As a result, the CSIM code must include specific modeling considerations for these various SEWs. Specifically, in the `exp` instruction, only SEW=32 is handled for the FP32 data type.

The naming convention for ACE-RVV helper functions is as follows:

`ace_`**`InsnName`**`_[i|u|f][SEW]m[LMUL]`

where `ace_` serves as the prefix indicating the ACE (and ACE-RVV) helper function, **`InsnName`** represents the name of the instruction, `i` denotes a signed integer, `u` signifies an unsigned integer, and `f` indicates a floating point.

For example, consider the source code of the `riscv_nn_softmax_f32` function below, which calls the `exp` instruction through the helper function `ace_exp_f32m8`.

```
while(size2 > 0)
    {
        vl = __riscv_vsetvl_e32m8(size2);
        vfloat32m8_t vData = __riscv_vle32_v_f32m8(in_vec2, vl);

        in_vec2 += vl;
        vData = __riscv_vfsub_vf_f32m8(vData, max, vl);
        size2 -= vl;

        //--- exp_f32 begin ---
        vData = ace_exp_f32m8(vData, vl);
        //--- exp_f32 end ---

        vSum = __riscv_vfadd_vv_f32m8(vSum, vData, vl);     //accumulate sum_f32
        __riscv_vse32_v_f32m8(out_vec2, vData, vl);

        out_vec2 += vl;
    }
```

`ace_exp_f32m8` is one of the helper functions specifically designed for the `exp` instruction, supporting a floating-point data type where SEW=32 and LMUL=8.