

## ARTICLE TYPE

## GCC compiler support for posit floating-point representation

Hsing Yu Peng<sup>1</sup> | Hung-Ru Wu<sup>1</sup> | Wei-Hua Chen<sup>1</sup> | Peng-Sheng Chen<sup>\*1,2</sup> | Vijay Holimath<sup>1</sup><sup>1</sup>Department of Computer Science and Information Engineering, National Chung Cheng University, Taiwan, R.O.C.<sup>2</sup>Advanced Institute of Manufacturing for High-tech Innovations, National Chung Cheng University, Taiwan, R.O.C.<sup>3</sup>VividSparks, , BSK Layout, Hubli - 580031, India

## Correspondence

\*Peng-Sheng Chen, Department of Computer Science and Information Engineering, National Chung Cheng University, Chia-Yi 621, Taiwan, R.O.C.

Email: pschen@cs.ccu.edu.tw

## Present Address

Department of Computer Science and Information Engineering, National Chung Cheng University, Chia-Yi 621, Taiwan, R.O.C.

## Abstract

The IEEE-754 floating-point format is a widely used floating-point arithmetic standard. Posit is an alternative format for floating-point representation that can overcome the weaknesses of the IEEE-754 format. Here we enhance the GNU Compiler Collection (GCC) to support posit numbers, with a focus on the C compiler, `float` data type, and 32-bit posit number. The posit number operations are handled using the software library. A program with a `float`-related computation can be compiled using an additional `-fposit` option, which uses posit numbers to do the computation. We conduct an experiment to verify the correctness of the posit representation generated by the proposed compiler using tested benchmark programs, and evaluate the results obtained from the IEEE-754 and posit representations. The experimental results show that the proposed compiler can correctly generate posit numbers. To the best of our knowledge, this work is the first to detail how to automatically switch from floating-point to posit arithmetic in the most commonly-used C compiler, GCC.

## KEYWORDS:

Compiler; Floating point; GCC; Posit

## 1 | INTRODUCTION

The IEEE-754 Standard for Binary Floating-Point Arithmetic is a technical standard for floating-point arithmetic that was established in 1985 and updated in 2008 and 2019<sup>1,2</sup>, with the ability to support portable floating-point arithmetic across multiple computing platforms. It has been adopted by many processors and floating-point arithmetic units, and is broadly embedded in the compiler and low-level software routines. An IEEE floating-point number is defined by three fields of bits: a sign that states if the number is positive or negative, a fraction that stores the significand bits, and an exponent that states the necessary scaling of the fraction. Although the IEEE-754 floating-point format is a standard, it has some inevitable problems, such as wasted bit patterns, mathematically incorrect values, and predefined fixed-size portions. For example, there are over 16 million 32-bit floating-point values that are defined as not a number (NaN), and negative zero and positive zero exhibit different behaviors. These nuances introduce derivative problems for hardware exception handling. Computing systems have recently become more closely integrated into our daily life due to the rapid development of hardware and software technologies. Emerging technologies, such as data processing/analytics, machine learning, Internet of Things, and automotive electronic systems equipped with embedded systems, continue to improve the convenience of incorporating computing systems into our daily life. These techniques employ massive floating-point operations, such that the accuracy and effectiveness of the operations greatly impact their outputs. One of the major characteristics of embedded systems is their requirement to effectively complete floating-point operations using limited computer hardware resources. For diverse application domains, a program that is run in an embedded system is usually expected to obtain better floating-point computing accuracy using limited computer hardware resources than what we presently have using floating-point. Various number systems and data representation techniques have been proposed to overcome the above-mentioned challenges by either enhancing or replacing floating-point numbers<sup>3</sup>. John L. Gustafson<sup>4</sup> proposed posit as a new format that can serve as an alternative to the IEEE-754 standard. Posit is defined as  $(n, es)$ , where  $n$  is the number of total bits and  $es$  (exponent bits) is used to control the dynamic range. Posit numbers closely resemble the distribution of values used in calculations involving real numbers. Therefore, posit numbers have a key advantage over IEEE-754 numbers in their ability to obtain a higher precision from a given number of bits. Recent studies have shown that using

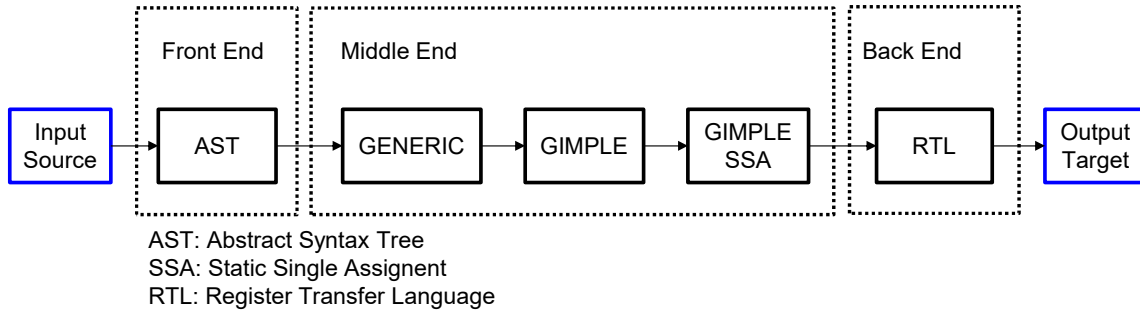


Figure 1 Structural diagram of the GCC architecture.

posit numbers in deep neural networks yields better accuracy and performance compared to using IEEE-754 numbers.<sup>5,6,7,8,9,10</sup>. However, the proposed posit format either supersedes or complements the current IEEE-754 format, depending on a number of factors, including software and hardware support.

Several studies<sup>11,12,13,14,15,16,17</sup> have implemented hardware for posit-based operations. The work in<sup>14</sup> modified GNU assembler and used inline-assembly codes in C level for supporting posit instructions. The work in<sup>16</sup> modified LLVM backend to generate proper posit instructions. The closest to our work is the compiler modification in<sup>13</sup>. They supported the conversion of floating-point literal to posit format in the modified GCC compiler. These works do not give a detailed description of modifying a compiler to support posit format. This article focuses on compiler issues for supporting posit-based arithmetic. The software library is used to support posit-related operations. This strategy helps developers evaluate the benefits of the accuracy of using posit at the early stage, even without posit-featured hardware. If worthwhile, developers can choose the appropriate hardware platform to meet performance requirement.

## 1.1 | GNU compiler collection

GCC (GNU Compiler Collection)<sup>18</sup> is a popular open-source compiler because it generates high-quality code and is portable. The GCC compiler consists of three parts: a front end, a middle end, and a back end. Figure 1 shows a structural diagram of the GCC architecture. The source code is processed in turn via these three parts during the compilation process. The front end reads the source code, and then parses and converts it into a standard abstract syntax tree (AST). Each language corresponds to a dedicated front end, with the resulting AST also being slightly different due to the different languages. The consistency step is executed after the AST is generated, whereby the AST is converted into a unified format that is called GENERIC. The middle-end stage lowers the GENERIC into another form called GIMPLE, and then the SSA form.

The compilation process then goes into the back end, where the GIMPLE SSA is transformed into the register-transfer language (RTL) representation, which is similar to a pseudo assembly code. The RTL representation is then optimized, and the proper target assembly code is generated according to the RTL and target hardware-related description.

Here we enhance the GCC compiler to support a posit number system. We focus on the C compiler, `float` data type, and 32-bit posit number system. The target machine is an x86. The operations of the posit numbers are computed using the software library. A program with a `float`-related computation can be compiled by selecting an additional `-fposit` option, which uses posit numbers to perform the computation. This design allows the user to seamlessly leverage the benefits of the posit format. We first study the GCC compilation flow, structure of the GCC back end, and related built-in software libraries. We then add a component to translate the GCC internal real number representation to the corresponding posit format. We also modify the GCC back end to support posit operations using built-in software libraries. We finally conduct an experiment to verify the correctness of the posit representation generated by the proposed compiler using tested benchmark programs, and also evaluate the results obtained from the IEEE-754 and posit representations. The experimental results show that the proposed compiler can correctly generate posit numbers. The application that subsequently uses these posit numbers can then obtain more accurate results compared with those using IEEE-754 numbers.

## 1.2 | Contributions

This paper makes the following contributions:

1. **Supporting the posit format in GCC internals.** To the best of our knowledge, this work is the first to detail how to automatically switch from floating-point to posit arithmetic in the most commonly-used C compiler, GCC. It transforms the GCC internal real number representation

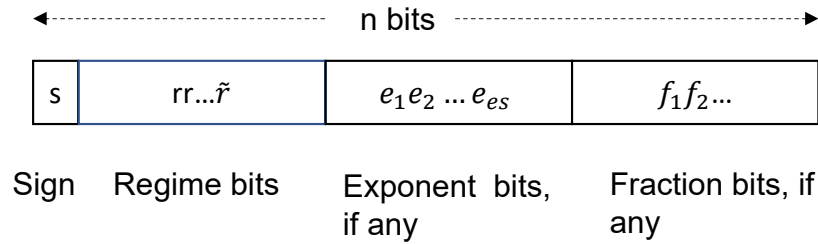


Figure 2 Fields in an  $n$ -bit posit number.

called `REAL VALUE TYPE` to the corresponding posit format. A new compiler option that can switch to either IEEE-754 or posit format is added to allow users to select their preferred number representation.

2. **Experiment.** Experimental results are presented to demonstrate the correctness of the generated posit numbers and their associated benefits compared with IEEE-754 numbers. The results show that our compiler correctly generates posit numbers. The posit format has obvious advantages in terms of accuracy compared with the IEEE-754 format for the tested benchmark programs.

### 1.3 | Organization

The remainder of this paper is organized as follows. Section 2 introduces the posit number format, and section 3 describes the proposed implementation that supports the posit format in GCC. Section 4 presents the experimental results. Finally, concluding remarks are given in Section 5.

## 2 | POSIT NUMBERS

John L. Gustafson<sup>4</sup> proposed the posit number, a Type III unum (universal number) that extends the IEEE standard on floating-point arithmetic, to address many of the shortcomings of IEEE-754 floating-point arithmetic. The format of the posit number is defined as  $\text{posit}_{(n,es)}$ , where  $n$  refers to the total number of bits and  $es$  indicates the number of exponent bits. The format is a fixed-length format with up to four fields, as shown in Figure 2. The primary difference between the posit and IEEE-754 floating-point number representations is the regime field of posit, which has a dynamic width like a unary number.

- **Sign:** Sign bit; 0 for positive numbers, 1 for negative numbers.
- **Regime:** Variable length; from 1 to  $n-1$  regime bits.
- **Exponent:** If any bits remain, the exponent bits are next in line, with at most  $es$  bits.
- **Fraction:** If any bits remain after the sign, regime, and exponent bits, they all belong to the fraction. The fraction includes a leading bit, which is always 1 and not stored as part of the fraction.

The value of a real number  $x$  can be represented by Equation 1, where  $k$  is the number of regime bits, which depends on whether the leading bit is one or zero.

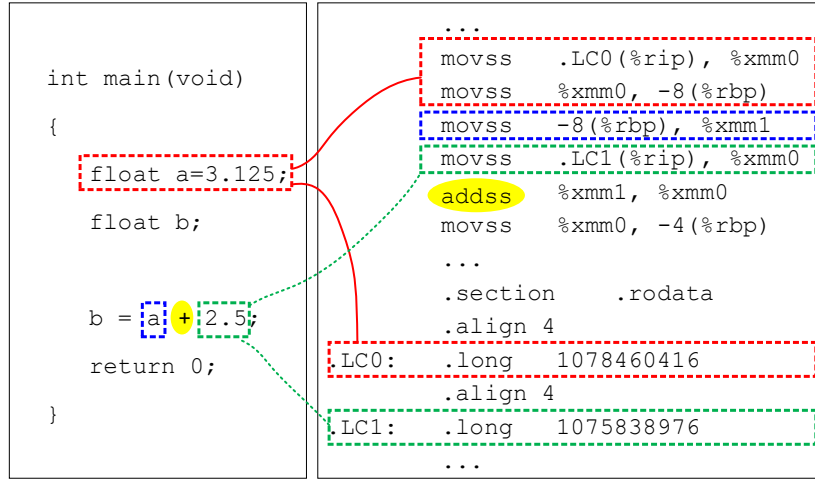
$$x = \begin{cases} 0 & b = 000\dots 0, \\ \text{Not a real number (NaR)} & b = 100\dots 0, \\ (-1)^s \times (2^{2^{es}})^k \times 2^e \times 1.f & \text{otherwise.} \end{cases} \quad (1)$$

$$k = \begin{cases} -m & \text{if regime has } m \text{ 0's} \\ m-1 & \text{if regime has } m \text{ 1's} \end{cases}$$

For example, a decimal number 575 that is expressed in  $\text{posit}_{(16,3)}$  format is represented by  $\text{sign} = 0$ ,  $k = 1$ ,  $es = 3$ ,  $e = 1$ , and  $f = \frac{63}{512}$ , as shown in Table 1. The corresponding equation formula is:  $(-1)^0 \times (2^{2^3})^1 \times 2^1 \times (1 + \frac{63}{512}) = 575$ .

**Table 1** Binary representation of 575, which is expressed in  $\text{posit}_{(16,3)}$  format.

S	Regime	Exponent	Fraction
0	110	001	000111111

**Figure 3** Example code with `float` arithmetic.

### 3 | IMPLEMENTATION

The `float` data type in C is represented and operated in IEEE-754 single-precision format. We first observe a program with `float` arithmetic and its corresponding assembly language. Figure 3 shows an example code that includes floating-point operations and the corresponding x86 assembly language that was generated by GCC 9.3.0. The dashed red rectangles show that the variable `a` is declared as a `float` data type and given an initial value of `3.125`. `3.125` is internally converted into the corresponding IEEE-754 format inside the compiler, and the decimal representation of this 32-bit number is `1078460416`. The dashed blue rectangles show that the variable `a` is loaded into the `xmm1` register. The dashed green rectangles show that `2.5` is converted into IEEE-754 single-precision format inside the compiler, and then read into the `xmm0` register. The decimal representation of this 32-bit number is `1075838976`. The variables `a` and `2.5` need to be added, and the appropriate hardware instruction `addss` is selected for the operation. We need to understand how floating-point numbers are stored and encoded inside GCC to effectively handle the `float` data type in C in a 32-bit `posit` format. Furthermore, we use the software library<sup>19</sup> to address the issue of any hardware instructions that do not support `posit` operations and perform `posit` arithmetic.

Figure 4 shows the proposed compilation flow. There are two paths for compiling a program with floating-point operations. One path employs the IEEE-754 floating-point format, as shown by the red lines. This path is the original flow, whereby the GCC handles floating-point operations. Even if there are hardware instructions to support a given data format, users may force the GCC to use the software library through the `-msoft-float` option. The other path employs the `posit` format, which is a newly added path, as shown by the blue lines. The path uses built-in software routines to handle `posit`-related operations. The built-in software routines constructed by SoftPosit library<sup>19</sup> are archived as `libposit.a`. A new compiler option, `-fposit`, is added to select the proper path.

Here we follow the `posit` standard document<sup>20</sup>, whereby the supported 32-bit `posit` number is `posit32`, which is represented as `posit(32,2)`. Our implementation can be divided into three parts. First, we understand the GCC-internal processing flow for handling a real number. Second, we design and implement an algorithm that transforms the GCC internal floating-point representation into the corresponding `posit` format. Finally, we support a low-level runtime library for `posit` operations.

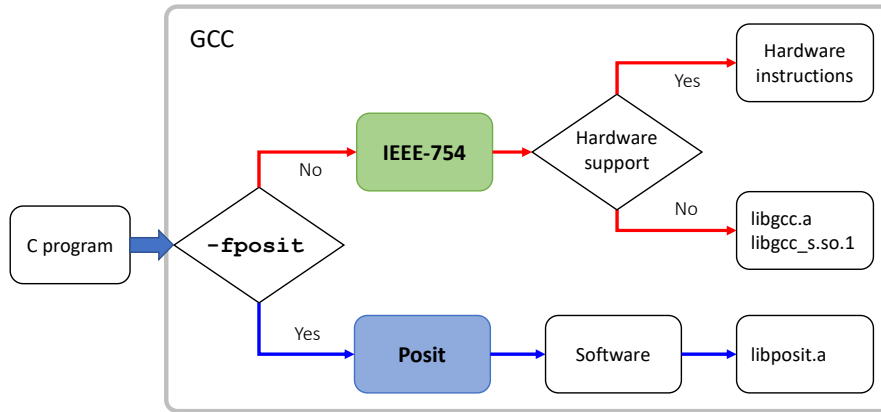


Figure 4 Processing flow for the proposed compilation.

### 3.1 | Floating-point representation in GCC internals

When GCC compiles a program containing real numbers, it converts the real numbers into a `REAL_VALUE_TYPE` internal representation after parsing and analyzing the semantics. The `REAL_VALUE_TYPE` data type is used internally in the GCC, which is defined in the file `gcc-x.x.x/gcc/real.h`. The file `gcc-x.x.x/gcc/real.c` contains the related functions for floating point emulation. The internal representation should describe a real value more accurately than the supported target floating-point representation for proper rounding. `REAL_VALUE_TYPE` is conceptually similar to the IEEE-754 format. However, its significand has greater precision, which is 160 bits on 32-bit systems and 192 bits on 64-bit systems for supporting all target machines. In addition, the entire significand of the intermediate representation is fractional. Normalized significands are in the range  $[0.5, 1.0)$  whereas those in IEEE-754 format are in the range  $[1.0, 2.0)$ . The exponent part is 26 bits, which does not have a bias representation.

GCC is designed to be easily ported to divergent back-end platforms. In order to explain the GCC internal operations and the subsequent algorithms more clearly, hereafter, we will assume that the host machine is a 64-bit system and `long` data type is 64 bits in length. Figure 5 shows the structure of significand in `REAL_VALUE_TYPE`. A three-element array `sig` is used to store the information.

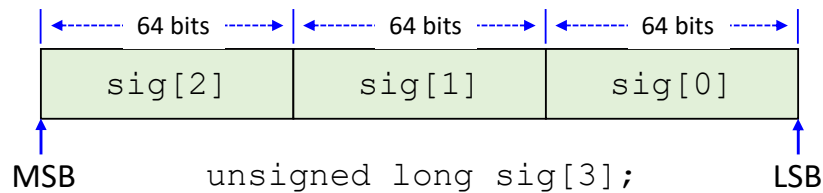


Figure 5 Significand part in `REAL_VALUE_TYPE`.

Consider the statement `float a=3.125;` in Example 3. Figure 6 shows the critical calling sequence during GCC parsing the floating-point constant. A top-level function named `interpret_float` is called to handle the work. The function `excess_precision_type` is used to select a proper significand size to store the floating-point constant. Here, the size selected is 64 bits. The function `real_from_string` leverages GNU MPFR (a library for reliable multiple precision floating-point arithmetic<sup>21</sup>) to convert the string “3.125” to the corresponding `REAL_VALUE_TYPE` value. The value from `real_from_string` has the significand with 192 bits. Next, the function `real_convert` is called to round the value to 64-bit significand, decided by `excess_precision_type`. Refer to Figure 5. That is `sig[0]` and `sig[1]` are zero. `sig[2]` has the significand. Finally, the function `build_real` constructs a proper tree node which contains the converted value. In the GIMPLE and RTL representation, “3.125” is stored as a `REAL_VALUE_TYPE` value with 64-bit significand.

The real value of a real number that is assigned to a `float` variable type is properly converted from the `REAL_VALUE_TYPE` data type. During the code generation of the back end, the function `encode_ieee_single()` reads the internal representation to obtain the information from each bit field and then composes a bit pattern in IEEE-754 single-precision format. The real values 3.125 and 2.5 in Example 3 are transformed to the IEEE-754 single-precision numbers, with bit patterns (in decimal) of 1078460416 and 1075838976, respectively.

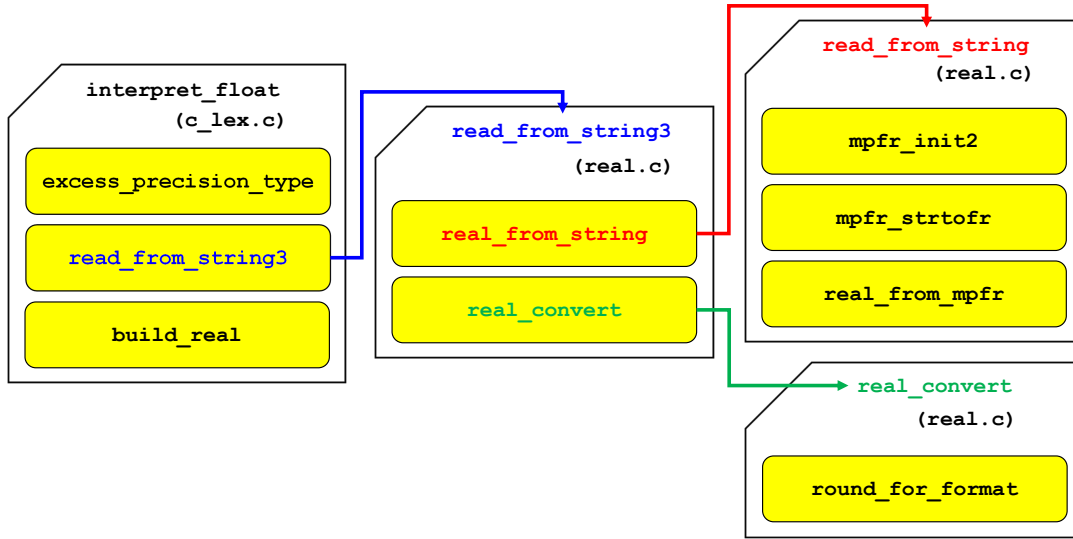


Figure 6 Critical calling sequence during parsing floating-point numbers in Example 3.

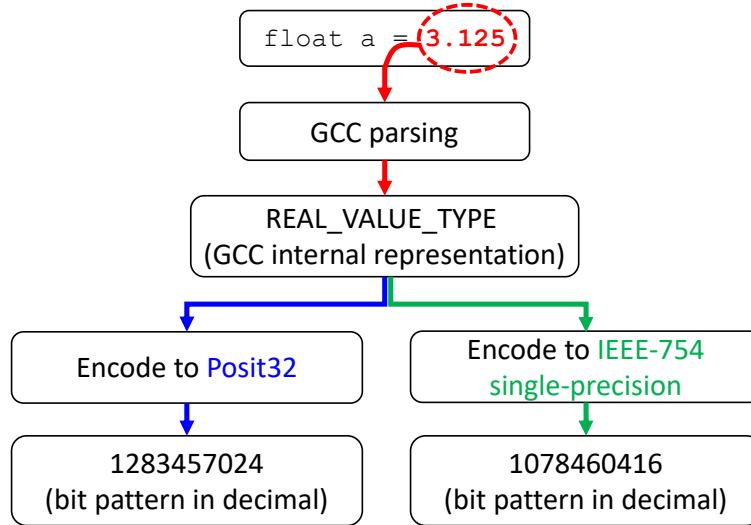


Figure 7 Encoding for the IEEE-754 single-precision and posit32 formats.

### 3.2 | Transformation of the posit number

We need to transform the GCC internal representation to the corresponding posit format to support posit32. The idea for this transformation is illustrated in Figure 7. Similar to `encode_ieee_single()`, a proper function should be added to perform the transformation. The function is internally called by `real_to_target()`.

We referred to the SoftPosit library<sup>19</sup> and proposed an algorithm to transform the GCC internal representation to posit32 format, as shown in Algorithm 1. The involved notations are described in Table 2. The algorithm reads a `REAL_VALUE_TYPE` variable type  $x$ , and then outputs the corresponding posit32 value  $px$ , whose bit pattern is represented as a 32-bit integer data type. Equation 2 shows the relationship between  $x$  and  $px$ . The bit patterns of  $px$  comprise special cases (i.e., maximum, minimum, not a number, infinity, and subnormal) and a normal case. We first extract the parts of the sign, exponent, and fraction via the GCC built-in macros. `REAL_EXP()` returns the exponent value. Recall that for `float` data type, the normalized significand of the internal representation is rounded to 64 bits, which is stored at `sig[2]`. `sig[1]` and `sig[0]` are zero. Therefore, we use `SIG2` to represent the significand of  $x$ . Due to that the significand is in the range  $[0.5, 1.0)$ , the value of `REAL_EXP()` needs to subtract 1 before assigning to  $exp$ . We then employ a switch construct to handle the special cases and normal case individually. Specific bit patterns are directly assigned to  $px$  for the special cases. The normal case indicates that the value of  $x$  is exactly expressible. Algorithm 2 lists the function

Table 2 Notation for Algorithms 1 and 2.

Name	Description
<i>zero</i>	value 0 (ignoring sign)
<i>Maxpos</i>	largest representable posit value
<i>Minpos</i>	smallest representable posit value
subnormal	A number is too small to normalize within the format
Inf	infinity or negative infinity
NaN	not a number
<i>s</i>	sign of <i>x</i>
<i>exp</i>	effective exponent value of <i>x</i>
<i>sig</i>	fraction value of <i>x</i>
<i>num<sub>r</sub></i>	regime length
<i>regime</i>	regime value of <i>px</i>
<i>f</i>	fraction value of <i>px</i> after shifting
<i>fLen</i>	fraction length
<i>bitNPlusOne</i>	sticky bit of fraction
<i>bitsMore</i>	sticky bit of <i>bitNPlusOne</i>

used for handling the normal case. These normal values are separated into two parts depending on whether the exponent value is greater than zero or not. Both parts use the exponent value to calculate the number of regime bits. The regime part need to add an opposite bit if there is still space. The variable *bitNPlusOne* is the LSB (least significant bit) if there is an n+1-bit representation. Here, n is 32. The variable *bitsMore* indicates whether more bits are required beyond *bitNPlusOne* in an exact representation. The fraction part is adjusted and then rounded according to *bitNPlusOne* and *bitsMore*. Next, the exponent part is computed. Finally, we return the posit result *px*, which is obtained by combining these parts.

$$\begin{cases}
 x = \pm 0 & px = 0x00000000 \\
 x = \text{NaN/Inf} & px = 0x80000000 \\
 x \geq \text{Maxpos} & px = 0x7FFFFFFF \\
 x \leq -\text{Maxpos} & px = 0x80000001 \\
 0 < x \leq \text{Minpos} & px = 0x00000001 \\
 -\text{Minpos} \leq x < 0 & px = 0xFFFFFFFF \\
 x = \text{subnormal} & px = 0x00000001 \\
 x = -\text{subnormal} & px = 0xFFFFFFFF \\
 x = (-1)^s \times (2^{2^e})^k \times 2^e \times (1.f) & px = \text{Other bit patterns.}
 \end{cases} \quad (2)$$

### 3.2.1 | Posit Arithmetic Computation

Whenever a target needs to perform an operation that is too complicated to emit proper target instructions, GCC will generate a call instruction to a subroutine that performs the operation. Such low-level subroutines are archived as a runtime library, which is called either `libgcc_s.so.1` or `libgcc.a`. This strategy makes the code generation flexible and simple. Most of the subroutines in `libgcc` are used to handle arithmetic operations when a target processor does not support hardware operations directly<sup>22</sup>. This design is suitable for addressing the problem that arises when the current hardware cannot support posit operations well. Here we focus on single-precision floating-point operations. Most of these subroutines can be defined using the machine-independent C language. The remainder must be hand-written in a target assembly language. Our work uses a soft-float library called `SoftPosit`<sup>19</sup> to support the posit operations, as shown by the blue-lined path in Figure 4. For the implementation, we need to properly define the configuration files that are located in the `gcc-x.x.x/libgcc` folder, with `config.host` being the `libgcc` host-specific configuration where a configuration type is mapped to different system-specific definitions and files. The `libgcc/config/target` directory contains extra files that are needed for the target architecture. One of the files is `sfp-machine.h`, which contains the default settings. `t-softfp` contains machine description-specific makefile fragments that are used for software floating-point emulation. These two files affect the library routines in the `libgcc/soft-fp` directory to be built up. The `libgcc/soft-fp` directory contains the subroutines for the floating-point emulation (e.g., addition, subtraction, multiplication, and division). There are about 77 routines in total that GCC provides for use on machines that

---

**Algorithm 1** Posit32 encoding from REAL\_VALUE\_TYPE.

---

**Input:** REAL\_VALUE\_TYPE-typed  $x$ 
**Output:** Posit32 compliance  $px$ 

```

1  $s \leftarrow$  sign bit of  $x$ 

   /* posit's fraction has a hidden bit. */
2  $exp \leftarrow \text{REAL\_EXP}(x) - 1$   $frac \leftarrow (\text{SIG2} \gg (64 - 29)) \& 0\text{FFFFFF}$  switch getClass( $x$ ) do
3   case zero do  $px \leftarrow 0\text{x}00000000$ ;
4   case Inf do  $px \leftarrow 0\text{x}80000000$ ;
5   case NaN do  $px \leftarrow 0\text{x}80000000$ ;
6   otherwise do
7     if  $x$  is  $\pm$ subnormal then
8        $px \leftarrow \pm Minpos$ 
9     else if ( $exp \geq 120$ )  $\wedge$  ( $s = 0$ ) then
10       $px \leftarrow Maxpos$ 
11     else if ( $exp \geq 120$ )  $\wedge$  ( $s = 1$ ) then
12       $px \leftarrow -Maxpos$ 
13     else if ( $exp \leq -120$ )  $\wedge$  ( $s = 0$ ) then
14       $px \leftarrow Minpos$ 
15     else if ( $exp \leq -120$ )  $\wedge$  ( $s = 1$ ) then
16       $px \leftarrow -Minpos$ 
17     else if ( $s = 0$ )  $\wedge$  ( $exp = 0$ )  $\wedge$  ( $frac = 0$ ) then
18       $px \leftarrow 0\text{x}40000000$ 
19     else if ( $s = 1$ )  $\wedge$  ( $exp = 0$ )  $\wedge$  ( $frac = 0$ ) then
20       $px \leftarrow 0\text{x}C0000000$ 
21     else
22       /* handle the normal cases */
23        $px \leftarrow \text{normal\_case}(s, exp, frac)$ 
24     end
25 end

```

---

do not have hardware-supported floating-point operations. Here we use the `-msoft-float` compiler option to enable and issue call instructions to these soft floating-point subroutines. Figure 8 shows an example with two IEEE-754-encoded real numbers that need to be added together. GCC would call the `addsf3` function, whereby the two numbers are emulated to unsigned integers and added according to their respective fields. Finally, these results are combined into IEEE-754 format and then returned. We modify 23 single-precision floating-point functions to support posit32 operations. Figure 9 shows an example `addsf3` function, whereby we use the `p32_add` function, which is provided in the SoftPosit library, to perform the addition operation for the posit format.

## 4 | EXPERIMENT

The posit32-enabled GCC compiler was implemented using GCC version 9.2.0. Tables 3 and 4 list the experimental environment and compiler options used in the evaluation, respectively. The test platform is x86\_64-linux-gnu. We used the `-fposit` option to switch to the posit floating-point format, and `-msoft-float -m32` to enable soft floating-point operations. The option `-ffloat-store` inhibits other options that might change whether a floating point value is taken from a register or memory. It prevents undesirable excess precision. The entire experiment contains two parts: to verify the correctness of the proposed compiler, and to evaluate the benefits obtained via the posit32 format compared with the IEEE-754 single-precision format using the proposed compiler.



---

**Algorithm 2** Function `normal_case()` that handles the normal case during the encoding.

---

**Input:**  $s$ ,  $exp$ , and  $frac$  of `REAL_VALUE_TYPE`-typed  $x$

**Output:** Posit32 compliance  $px$

```

26 if  $exp \geq 0$  then
27    $num_r \leftarrow 1$  while  $exp \geq 4$  do
28      $exp \leftarrow exp - 4$   $num_r \leftarrow num_r + 1$ 
29   end
30    $regime \leftarrow ((1 \ll num_r) - 1) \ll 1$ 
31 else
32    $num_r \leftarrow 0$  while  $exp < 0$  do
33      $exp \leftarrow exp + 4$   $num_r \leftarrow num_r + 1$ 
34   end
35    $regime \leftarrow 1$ 
36 end

  /* length of regime is  $num_r + 1$ . */
37  $fracPLen \leftarrow 28 - num_r$ 
38 if  $fracPLen < 0$  then
39    $fracP \leftarrow 0$  if  $num_r = 29$  then
40      $bitNPlusOne \leftarrow exp \& 0x1$   $bitsMore \leftarrow 0$   $exp \leftarrow exp \gg 1$ 
41   else
42      $bitNPlusOne \leftarrow exp \gg 1$   $bitsMore \leftarrow exp \& 0x1$   $exp \leftarrow 0$ 
43   end
44   if  $frac \neq 0$  then
45      $bitsMore \leftarrow 1$ 
46   end
47 else
48    $exp \leftarrow exp \ll (28 - num_r)$   $fracP \leftarrow frac \gg (28 - fracPLen)$   $bitNPlusOne \leftarrow (SIG2 \gg (64 - 2 - fracLen)) \& 1$ 
49   if  $(SIG2 \& ((1 \ll (63 - fracLen)) - 1)) > 0$  then
50      $bitsMore \leftarrow 1$ 
51   else
52      $bitsMore \leftarrow 0$ 
53   end
54 end

55  $px \leftarrow ((regime \ll (30 - num_r)) + exp + fracP$  /* Rounding. */
56 if  $(bitNPlusOne \& (px \& 1)) \mid (bitNPlusOne \& bitsMore)$  then
57    $px \leftarrow px + 1$ 
58 end
59 if  $s = 1$  then /*  $x$  is negative. */
60    $px \leftarrow 2$ 's complement of  $px$ 
61 end
62 return  $px$ 

```

---

#### 4.1 | Verification of the proposed posit compiler

We checked the bit patterns of the constant values encoded by the proposed compiler according to the posit standard document<sup>20</sup> to verify our implementation. The bit patterns of the posit numbers contain special cases (i.e.,  $\pm Maxpos$ ,  $\pm Minpos$ , Inf, NaN, zero) and a normal case. We tested the special cases and 5000 randomly generated floating-point values to check whether they meet the corresponding format, as shown in Figure 10.

We also evaluated the primitive operations to verify the effectiveness of the proposed compiler. The addition, subtraction, multiplication, and division operations are separately computed for the same input data using the SoftPosit library and proposed compiler. We randomly generate 5000 sets of input data, and then evaluate whether the results are identical, as shown in Figure 11. The experimental results show that the proposed

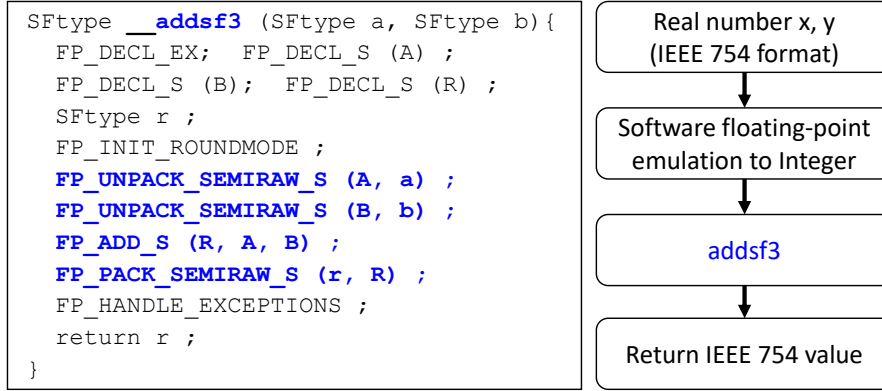


Figure 8 Example of the addition operation (IEEE-754 single-precision format).

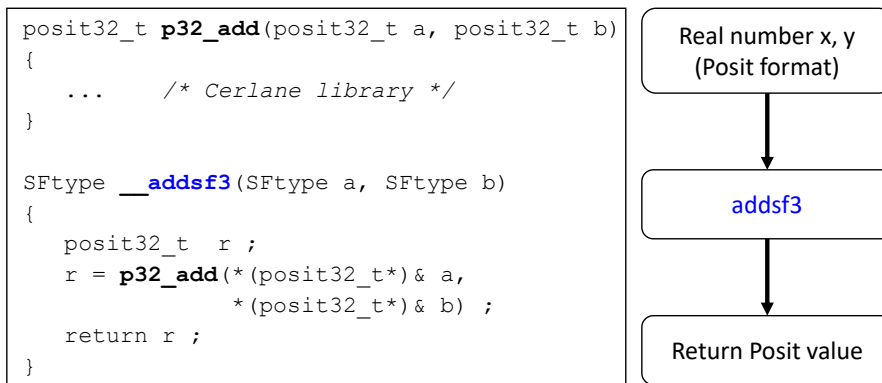


Figure 9 Example of the addition operation (posit32 format).

Table 3 Experimental environment.

Hardware	Processor	Intel Core-i7 7700 3.6 GHz
	DRAM	8 GB
Software	Operating system	Ubuntu Linux 20.04
	System type	x86_64-linux-gnu
	Linux kernel version	Linux ubuntu 5.4.0-72-generic
	Compiler	GNU GCC 9.2.0
	Data type float	posit <sub>(32,2)</sub> / IEEE 754 single-precision

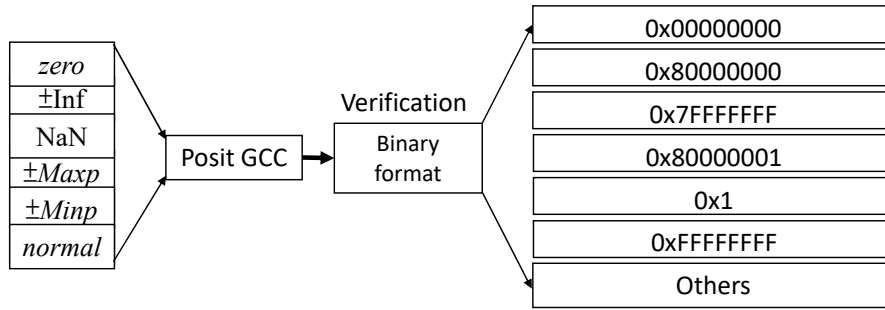
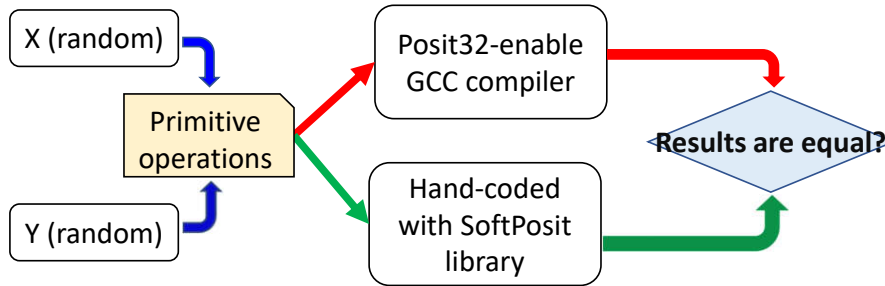
compiler can correctly encode floating-point numbers to the corresponding posit format, and accurately perform the primitive operations in posit format.

## 4.2 | Evaluation of real programs

Here we evaluate the benefits obtained from the posit32 format compared with the IEEE-754 single-precision format via the proposed compiler. Two programs are selected in the evaluation. The tested programs are compiled using the IEEE-754 single-precision and posit32 formats. The execution results for the two formats are compared to elucidate the accuracy of the formats.

**Table 4** Compiler options.

Option	Description
-fposit	Enable posit format instead of IEEE-754 single-precision.
-msoft-float -m32	Generate output containing library calls for floating point.
-lsoft-fp	Link to posit library for library calls.
-mfpmath = sse	Uses scalar floating-point instructions present in the SSE instruction.
-msse2	Enables SSE extensions.
-ffp-contract=off	Turns off the fused multiple add option for floating-point computations.
-ffloat-store	Sets the compiler to not store floating points variables in registers.

**Figure 10** Tested bit patterns.**Figure 11** Concept of the evaluation of the primitive operations.

#### 4.2.1 | Computing $\pi$

We used the John Machin formula<sup>23</sup> to gradually approach  $\pi$  precision. The formula is shown in Equation 3, where  $\arctan$  is computed via Taylor Series expansion (Equation 4). We estimate that the floating-point value is accurate to 17 digits after the decimal point as the reference point. The absolute error is the absolute value of the baseline value minus the evaluated value. The result is shown in Table 5. The absolute errors for the posit32 and IEEE-754 single-precision results are about  $1.98 \times 10^{-9}$  and  $8.74 \times 10^{-8}$ , respectively, with the error obtained using IEEE-754 single-precision numbers being about 44 times larger than the result obtained using posit32 numbers.

$$\frac{\pi}{4} = 4 \arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right) \quad (3)$$

**Table 5** Evaluation results for  $\pi$ .


	posit32	IEEE-754 single-precision
Evaluated value	3.14159265557398015	3.14159274101257324
Absolute error	0.00000000198418704 $\sim 1.98 \times 10^{-9}$	0.00000008742278010 $\sim 8.74 \times 10^{-8}$

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots \quad (4)$$

### 4.3 | Machining with learning control

The CNC machine tool is an indispensable element in the manufacturing process. Here we evaluate an iterative-learning control (ILC) algorithm<sup>24</sup> for machining using the posit32 and IEEE-754 single-precision formats. ILC is a technique that uses the current machining information to improve the accuracy of the next machining process. The tested algorithm is run on a five-axis machine tool, with the details provided in Table 6. We implement the ILC algorithm and integrate it into LinuxCNC<sup>25</sup>, an open-source program for computer control, to drive the machine tool and enhance the machining accuracy. We hand-coded the required mathematical functions to support the posit format since the C standard library does not support the posit format. Circle and NURBS (Non-Uniform Rational Basis Spline)<sup>26</sup> curves are selected as tested machining paths for the evaluation, with the tested machining paths shown in Figures 12 and 13, respectively.

**Table 6** Tested five-axis machine tool.

Items	Description	
CPU	Intel Core-i7 2600 (3.4 GHz)	
RAM	3 GB	
OS	Ubuntu 10.04 (Linux kernel 2.6.32-122 RTAI)	
CNC software	LinuxCNC 2.5.4	
Analog output	Advantech PCI-1723	
Encoder	Advantech PCI-1784	
Driver	YASKAWA SGD7 resolution 0.6 $\mu$ m	
Motor	YASKAWA 3000 RPM, ax.5000 RPM	

The contour error  $\varepsilon_i$  of a point  $x_i$  is the minimum distance between  $x_i$  and the expected machining path. Figure 14 shows the concept of the contour error. We use the root mean square (RMS) of the contour errors to evaluate all of the actual points. The RMS formula is given by Equation 5, where  $\varepsilon_i$  is the contour error of each actual point.

$$RMS = \sqrt{\frac{\sum_{i=1}^n \varepsilon_i^2}{n}} = \sqrt{\frac{\varepsilon_1^2 + \varepsilon_2^2 + \dots + \varepsilon_n^2}{n}} \quad (5)$$

We record the RMS contour error after each learning iteration. The learning termination condition is used to stop learning when the contour error increases. Equation 6 shows the learning termination condition, where  $\varepsilon_{rms,j}$  and  $\varepsilon_{rms,j-1}$  are the RMS contour errors after the  $j$ -th and  $(j-1)$ -th learning iteration, respectively.  $\varepsilon_{rms,j-1}$  is termed the converged RMS contour error when the  $j$ -th learning iteration satisfies Equation 6. The convergence ratio (CR) is the expected ratio of  $\varepsilon_{rms,j}$  to  $\varepsilon_{rms,j-1}$ , whereby the contour errors at the next machining iteration are expected to be smaller than those at the present machining iteration. We tested CR values of 0.1, 0.5, and 0.9 during the evaluation. Figure 15 compares the converged RMS contour errors that were calculated from the posit32 and IEEE-754 single-precision versions, and Table 7 lists the details of the converged RMS contour errors. The results show that the posit32 version has a better accuracy than IEEE-754 single-precision version for all of the tested CR values.

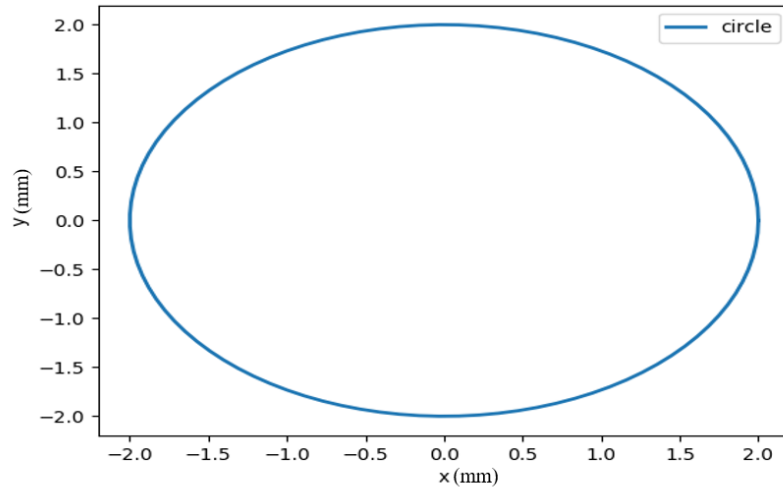


Figure 12 Tested machining path: circle.

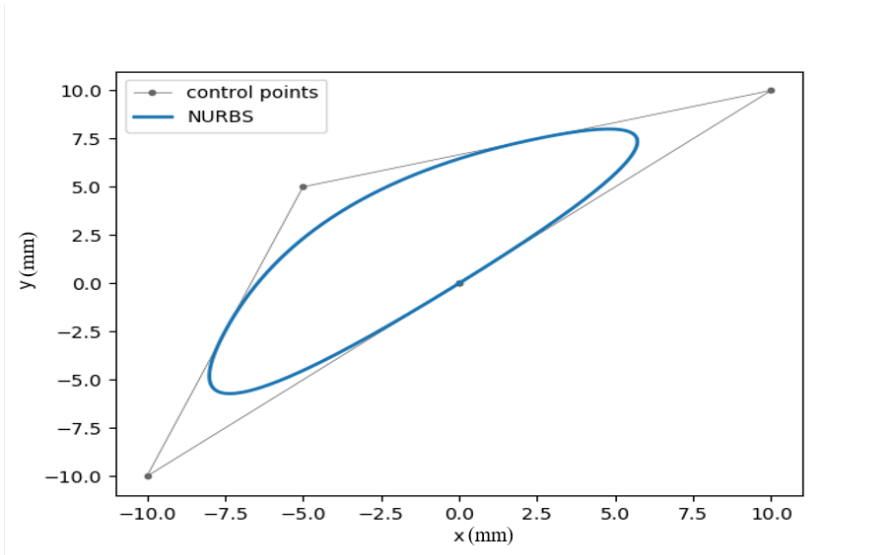


Figure 13 Tested machining path: NURBS curve.

The results have little improvement. The reason may come from two parts. One is that the input data actual positions are obtained from LinuxCNC. These position values may not be precise enough, so the difference between posit32 and single-precision representations is slight. The other is data conversion among the learning iterations. For each learning iteration, the command positions generated by ILC are converted to the double-precision format and then stored in a file. Next, the command positions in the file are loaded and rounded for the next learning.

$$\frac{\varepsilon_{rms,j}}{\varepsilon_{rms,j-1}} > 1 \quad (6)$$

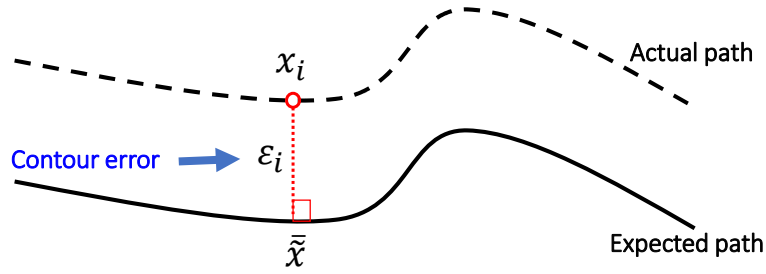


Figure 14 Concept of the contour error.

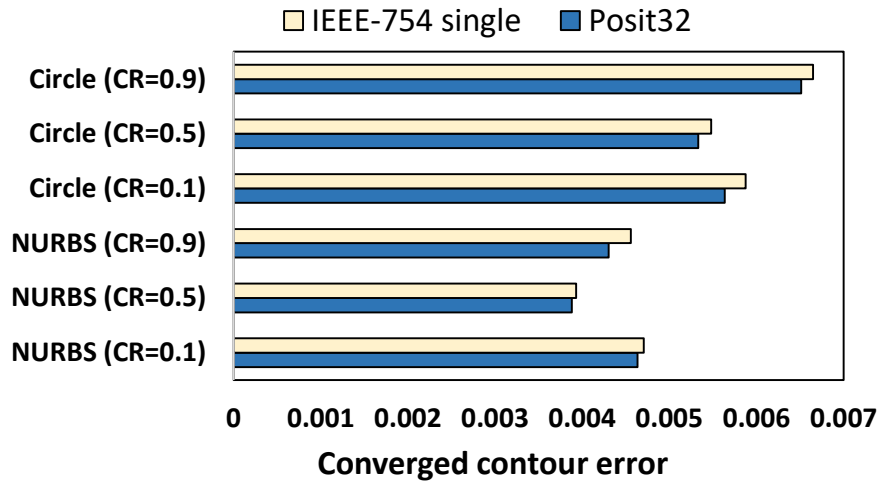


Figure 15 Evaluation of the converged RMS contour errors.

Table 7 Comparison of the converged RMS contour errors.

Items	posit32	IEEE-754 single-precision
NURBS (CR=0.1)	0.004635449	0.004705870
NURBS (CR=0.5)	0.003881841	0.003929080
NURBS (CR=0.9)	0.004305322	0.004558363
Circle (CR=0.1)	0.005636368	0.005875778
Circle (CR=0.5)	0.005330911	0.005479364
Circle (CR=0.9)	0.006513486	0.006649797

## 5 | CONCLUSION

Posit numbers closely resemble the distribution of values used in calculations involving real numbers. Therefore, it has some advantages over the IEEE-754 floating-point format, including a better accuracy and the ability to be customized for domain-specific applications. These advantages can be leveraged by emerging technologies (e.g., deep learning and automotive electronic systems) to provide additional benefits. Here we enhanced GCC compiler to support the posit floating-point format. We targeted the IEEE-754 single-precision and posit<sub>(32,2)</sub> formats, noting that other-sized formats could be supported using a similar approach. We first studied the process of how GCC handles floating-point operations. A conversion algorithm that transformed the GCC internal floating-point representation to the corresponding posit format was then implemented. We also added a compiler option that allows programmers to easily switch between the IEEE-754 single-precision and posit formats. The GCC low-level runtime library and third-party SoftPosit library were used for the posit operations. We finally conducted an experiment to verify the correctness of the

proposed compiler, and evaluated the benefits of the obtained results in posit32 format compared with those in IEEE-754 single-precision format. The experimental results highlighted that the tested programs compiled by the proposed compiler could be correctly executed. Furthermore, the tested programs in posit floating-point format could achieve higher accuracy than those in IEEE-754 single-precision format.

However, the posit format still has key disadvantages<sup>27</sup>, namely insufficient hardware and software support. Luc Forget et al.<sup>28</sup> compared the hardware costs of posit and IEEE-754. According to their study, the cost of posit hardware is high, and it remains an open question whether the accuracy improvement offered by posit is worth the extra hardware resources they require. On the other hand, additional system software (e.g., C standard library and compiler) is required to support the posit format and fully address these support issues. The proposed compiler in this paper provides a solution for the part of these issues. The main obstacle in promoting the posit format is its early stage of system development, which makes it difficult for developers to evaluate whether the posit format can bring benefits and if it should be adopted. The proposed compiler and presented research results will hopefully encourage users to employ the posit format without the need to modify the program, which will help to overcome this obstacle and provide more choices when implementing software that uses floating-point operations.

## ACKNOWLEDGEMENTS

This work is supported in part by the Ministry of Science and Technology, Taiwan, under the grant number MOST 110-2221-E-194-019-MY3. The authors are grateful to the National Center for High-Performance Computing for computer time and facilities.

## References

1. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* 2008: 1-70. doi: 10.1109/IEEESTD.2008.4610935
2. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* 2019: 1-84. doi: 10.1109/IEEESTD.2019.8766229
3. Fousse L, Hanrot G, Lefèvre V, Pélissier P, Zimmermann P. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Softw.* 2007; 33(2): 13-es. doi: 10.1145/1236463.1236468
4. Gustafson J, Yonemoto I. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations* 2017; 4(2). doi: 10.14529/jsf170206
5. Carmichael Z, Langroudi HF, Khazanov C, Lillie J, Gustafson JL, Kudithipudi D. Deep Positron: A Deep Neural Network Using the Posit Number System. In: ; 2019: 1421-1426
6. Fatemi Langroudi SH, Pandit T, Kudithipudi D. Deep Learning Inference on Embedded Devices: Fixed-Point vs Posit. In: ; 2018: 19-23
7. Lu J, Lu S, Wang Z, et al. Training Deep Neural Networks Using Posit Number System. In: ; 2019: 62-67
8. Klöwer M, Düben PD, Palmer TN. Posits as an Alternative to Floats for Weather and Climate Models. In: CoNGA'19. Association for Computing Machinery; 2019
9. Romanov AY, Stempkovsky AL, Lariushkin IV, et al. Analysis of Posit and Bfloat Arithmetic of Real Numbers for Machine Learning. *IEEE Access* 2021; 9: 82318-82324. doi: 10.1109/ACCESS.2021.3086669
10. Cococcioni M, Rossi F, Ruffaldi E, Saponara S. Small Reals Representations for Deep Learning at the Edge: A Comparison. In: Gustafson J, Dimitrov V., eds. *Next Generation Arithmetic* Springer International Publishing; 2022; Cham: 117-133.
11. Podobas A, Matsuoka S. Hardware Implementation of POSITs and Their Application in FPGAs. In: ; 2018: 138-145
12. Xiao F, Liang F, Wu B, Liang J, Cheng S, Zhang G. Posit Arithmetic Hardware Implementations with The Minimum Cost Divider and SquareRoot. *Electronics* 2020; 9(10). doi: 10.3390/electronics9101622
13. Tiwari S, Gala N, Rebeiro C, Kamakoti V. PERI: A Configurable Posit Enabled RISC-V Core. *ACM Trans. Archit. Code Optim.* 2021; 18(3). doi: 10.1145/3446210
14. Sharma N, Jain R, Mohan M, et al. CLARINET: A RISC-V based framework for posit arithmetic empiricism. *arXiv preprint arXiv:2006.00364* 2020.

15. Arunkumar M, Bhairathi SG, Hayatnagarkar HG. Perc: Posit enhanced rocket chip. In: ; 2020.
16. Mallasén D, Murillo R, Del Barrio AA, Botella G, Piñuel L, Prieto M. PERCIVAL: Open-Source Posit RISC-V Core with Quire Capability. *arXiv preprint arXiv:2111.15286* 2021.
17. Murillo R, Mallasén D, Del Barrio AA, Botella G. Comparing Different Decodings for Posit Arithmetic. In: Gustafson J, Dimitrov V., eds. *Next Generation Arithmetic* Springer International Publishing; 2022; Cham: 84–99.
18. Stallman RM, GCC Developer Community t. *Using the GNU Compiler Collection*. GNU Press . 2020. Online available at <http://gcc.gnu.org/onlinedocs>.
19. Leong S. SoftPosit. <https://gitlab.com/cerlane/SoftPosit>; 2017. [Online; accessed 1-June-2020].
20. Posit Standard Documentation Release 3.2-draft. [https://posithub.org/docs/posit\\_standard.pdf](https://posithub.org/docs/posit_standard.pdf); 2018. [Online; accessed 1-June-2020].
21. Website . The GNU MPFR Library. <https://www.mpfr.org/>; 2019.
22. GNU Compiler Collection (GCC) Internals:Routines for floating point emulation. <https://gcc.gnu.org/onlinedocs/gccint/Soft-float-library-routines.html#Soft-float-library-routines>; 2019. [Online; accessed 1-June-2020].
23. ProofWiki . Machin's Formula for Pi. [https://proofwiki.org/wiki/Machin%27s\\_Formula\\_for\\_Pi](https://proofwiki.org/wiki/Machin%27s_Formula_for_Pi); 2019. [Online; accessed 1-June-2020].
24. Chen SL, Chang YC. Notice of Removal: Contouring control of multi-axis machine tools with iterative learning control. In: ; 2015: 667-670
25. Website . LinuxCNC: Software for realtime control. [www.linuxcnc.org](http://www.linuxcnc.org); 2019.
26. Piegl L, Tiller W. *The NURBS Book*. Berlin/Heidelberg: Germany: Springer-Verlag. second ed. 1997.
27. Dinechin dF, Forget L, Muller JM, Uguen Y. Posits: The Good, the Bad and the Ugly. In: CoNGA'19. Association for Computing Machinery; 2019; New York, NY, USA
28. Luc Forget FdD. Comparing posit and IEEE-754 hardware costs. [https://hal.science/hal-03195756/file/2021\\_Posit\\_IEEE754\\_Hardware\\_Cost.pdf](https://hal.science/hal-03195756/file/2021_Posit_IEEE754_Hardware_Cost.pdf); 2021. [Online; accessed 1-June-2022].

