RISC-V BF16 Extensions

Version 1.0, 05 July 2024: Ratified

Table of Contents

Preamble

This document describes the BF16 format and instruction extensions to the RISC-V Instruction Set Architecture.

This document is in the Frozen State

Change is extremely unlikely. A high threshold will be used, and a change will only occur because of some truly critical issue being identified during the public review cycle. Any other desired or needed changes can be the subject of a follow-on new extension.

Document Version Information:

See github.com/riscv/riscv-bfloat16 for more information.

Acknowledgments

Contributors to all versions of the specification (in alphabetical order) include:

- GouYue
- Ken Dockser (Editor)
- Kenneth Rovers
- Nick Knight
- Nicolas Brunie

We are grateful to the other people who have helped to improve this specification through their comments, reviews, feedback and questions.

Chapter 1. Introduction

When FP16 (officially called binary16) was first introduced by the IEEE-754 standard, it was just an interchange format. It was intended as a space/bandwidth efficient encoding that would be used to transfer information. This is in line with the Zfhmin extension.

However, there were some applications (notably graphics) that found that the smaller precision and dynamic range was sufficient for their space. So, FP16 started to see some widespread adoption as an arithmetic format. This is in line with the Zfh extension.

While it was not the intention of '754 to have FP16 be an arithmetic format, it is supported by the standard. Even though the '754 committee recognized that FP16 was gaining popularity, the committee decided to hold off on making it a basic format in the 2019 release. This means that a '754 compliant implementation of binary floating point, which needs to support at least one basic format, cannot support only FP16 - it needs to support at least one of binary32, binary64, and binary128.

Experts working in machine learning noticed that FP16 was a much more compact way of storing operands and often provided sufficient precision for them. However, they also found that intermediate values were much better when accumulated into a higher precision. The final computations were then typically converted back into the more compact FP16 encoding. This approach has become very common in machine learning (ML) inference where the weights and activations are stored in FP16 encodings. There was the added benefit that smaller multiplication blocks could be created for the FP16's smaller number of significant bits. At this point, widening multiply-accumulate instructions became much more common. Also, more complicated dot product instructions started to show up including those that packed two FP16 numbers in a 32-bit register, multiplied these by another pair of FP16 numbers in another register, added these two products to an FP32 accumulate value in a 3rd register and returned an FP32 result.

Experts working in machine learning at Google who continued to work with FP32 values noted that the least significant 16 bits of their mantissas were not always needed for good results, even in training. They proposed a truncated version of FP32, which was the 16 most significant bits of the FP32 encoding. This format was named BFloat16 (or BF16). The B in BF16, stands for Brain since it was initially introduced by the Google Brain team. Not only did they find that the number of significant bits in BF16 tended to be sufficient for their work (despite being fewer than in FP16), but it was very easy for them to reuse their existing data; FP32 numbers could be readily rounded to BF16 with a minimal amount of work. Furthermore, the even smaller number of the BF16 significant bits enabled even smaller multiplication blocks to be built. Similar to FP16, BF16 multiply-accumulate widening and dot-product instructions started to proliferate.

1.1. Intended Audience

Floating-point arithmetic is a specialized subject, requiring people with many different backgrounds to cooperate in its correct and efficient implementation. Where possible, we have written this specification to be understandable by all, though we recognize that the motivations and references to algorithms or other specifications and standards may be unfamiliar to those who are not domain experts.

This specification anticipates being read and acted on by various people with different backgrounds. We have tried to capture these backgrounds here, with a brief explanation of what we expect them to know, and how it relates to the specification. We hope this aids people's understanding of which aspects of the specification are particularly relevant to them, and which they may (safely!) ignore or pass to a colleague.

Software developers

These are the people we expect to write code using the instructions in this specification. They should understand the motivations for the instructions we include, and be familiar with most of the algorithms and outside standards to which we refer.

Computer architects

We expect architects to have some basic floating-point background. Furthermore, we expect architects to be able to examine our instructions for implementation issues, understand how the instructions will be used in context, and advise on how they best to fit the functionality.

Digital design engineers & micro-architects

These are the people who will implement the specification inside a core. Floating-point expertise is assumed as not all of the corner cases are pointed out in the specification.

Verification engineers

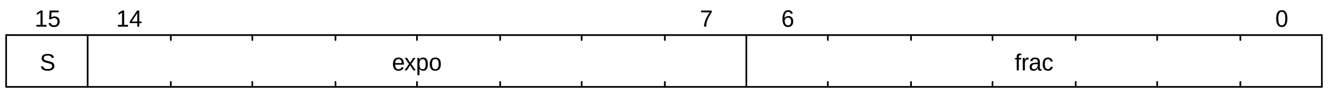
Responsible for ensuring the correct implementation of the extension in hardware. These people are expected to have some floating-point expertise so that they can identify and generate the interesting corner cases --- include exceptions --- that are common in floating-point architectures and implementations.

These are by no means the only people concerned with the specification, but they are the ones we considered most while writing it.

Chapter 2. Number Format

2.1. BF16 Operand Format

BF16 bits



IEEE Compliance: While BF16 (also known as BFloat16) is not an IEEE-754 *standard* format, it is a valid floating-point format as defined by IEEE-754. There are three parameters that specify a format: radix (b), number of digits in the significand (p), and maximum exponent (emax). For BF16 these values are:

Table 1. BF16 parameters

Parameter	**Value**
radix (b)	2
significand (p)	8
emax	127

Table 2. Obligatory Floating Point Format Table

Format	**Sign Bits**	**Expo Bits**	**fraction bits**	**padded 0s**	**encoding bits**	**expo max/bias**	**expo min**
FP16	1	5	10	0	16	15	-14
BF16	1	8	7	0	16	127	-126
TF32	1	8	10	13	32	127	-126
FP32	1	8	23	0	32	127	-126
FP64	1	11	52	0	64	1023	-1022
FP128	1	15	112	0	128	16,383	-16,382

2.2. BF16 Behavior

For these BF16 extensions, instruction behavior on BF16 operands is the same as for other floating-point instructions in the RISC-V ISA. For easy reference, some of this behavior is repeated here.

2.2.1. Subnormal Numbers:

Floating-point values that are too small to be represented as normal numbers, but can still be expressed by the format's smallest exponent value with a "0" integer bit and at least one "1" bit in the trailing fractional bits are called subnormal numbers. Basically, the idea is there is a trade off of precision to support *gradual underflow*.

All of the BF16 instructions in the extensions defined in this specification (i.e., Zfbfmin, Zvfbfmin and Zvfbfwma) fully support subnormal numbers. That is, instructions are able to accept subnormal values as inputs and they can produce subnormal results.

> Future floating-point extensions, including those that operate on BF16 values, may chose not to support subnormal numbers. The comments about supporting subnormal BF16 values are limited to those instructions defined in this specification.

2.2.2. Infinities:

Infinities are used to represent values that are too large to be represented by the target format. These are usually produced as a result of overflows (depending on the rounding mode), but can also be provided as inputs. Infinities have a sign associated with them: there are positive infinities and negative infinities.

Infinities are important for keeping meaningless results from being operated upon.

2.2.3. NaNs

NaN stands for Not a Number.

There are two types of NaNs: signalling (sNaN) and quiet (qNaN). No computational instruction will ever produce an sNaN; These are only provided as input data. Operating on an sNaN will cause an invalid operation exception. Operating on a Quiet NaN usually does not cause an exception.

QNaNs are provided as the result of an operation when it cannot be represented as a number or infinity. For example, performing the square root of -1 will result in a qNaN because there is no real number that can represent the result. NaNs can also be used as inputs.

NaNs include a sign bit, but the bit has no meaning.

NaNs are important for keeping meaningless results from being operated upon.

Except where otherwise explicitly stated, when the result of a floating-point operation is a qNaN, it is the RISC-V canonical NaN. For BF16, the RISC-V canonical NaN corresponds to the pattern of *0x7fc0* which is the most significant 16 bits of the RISC-V single-precision canonical NaN.

2.2.4. Scalar NaN Boxing

RISC-V applies NaN boxing to scalar results and checks for NaN boxing when a floating-point operation --- even a vector-scalar operation --- consumes a value from a scalar floating-point register. If the value is properly NaN-boxed, its least significant bits are used as the operand, otherwise it is treated as if it were the canonical QNaN.

NaN boxing is nothing more than putting the smaller encoding in the least significant bits of a register and setting all of the more significant bits to “1”. This matches the encoding of a qNaN (although not the canonical NaN) in the larger precision.

Nan-boxing never affects the value of the operand itself, it just changes the bits of the register that are more significant than the operand’s most significant bit.

2.2.5. Rounding Modes:

As is the case with other floating-point instructions, the BF16 instructions support all 5 RISC-V Floating-point rounding modes. These modes can be specified in the `rm` field of scalar instructions as well as in the `frm` CSR

Table 3. RISC-V Floating Point Rounding Modes

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards −∞)
011	RUP	Round Up (towards +∞)
100	RMM	Round to Nearest, ties to Max Magnitude

As with other scalar floating-point instructions, the rounding mode field `rm` can also take on the `DYN` encoding, which indicates that the instruction uses the rounding mode specified in the `frm` CSR.

Table 4. Additional encoding for the `rm` field of scalar instructions

Rounding Mode	Mnemonic	Meaning
111	DYN	select dynamic rounding mode

In practice, the default IEEE rounding mode (round to nearest, ties to even) is generally used for arithmetic.

2.2.6. Handling exceptions

RISC-V supports IEEE-defined default exception handling. BF16 is no exception.

Default exception handling, as defined by IEEE, is a simple and effective approach to producing results in exceptional cases. For the coder to be able to see what has happened, and take further action if needed, BF16 instructions set floating-point exception flags the same way as all other floating-point instructions in RISC-V.

Underflow

The IEEE-defined underflow exception requires that a result be inexact and tiny, where tininess can be detected before or after rounding. In RISC-V, tininess is detected after rounding.

It is important to note that the detection of tininess after rounding requires its own rounding that is different from the final result rounding. This tininess detection requires rounding as if the exponent were unbounded. This means that the input to the rounder is always a normal number. This is different from the final result rounding where the input to the rounder is a subnormal number when the value is too small to be represented as a normal number in the target format. The two different roundings can result in underflow being signalled for results that are rounded back to the normal range.

As is defined in '754, under default exception handling, underflow is only signalled when the result is tiny and inexact. In such a case, both the underflow and inexact flags are raised.

Chapter 3. Extensions

The group of extensions introduced by the BF16 Instruction Set Extensions is listed here.

Detection of individual BF16 extensions uses the unified software-based RISC-V discovery method.

At the time of writing, these discovery mechanisms are still a work in progress.

The BF16 extensions defined in this specification (i.e., `Zfbfmin`, `Zvfbfmin`, and `Zvfbfwma`) depend on the single-precision floating-point extension `F`. Furthermore, the vector BF16 extensions (i.e.,`Zvfbfmin`, and `Zvfbfwma`) depend on the `"V"` Vector Extension for Application Processors or the `Zve32f` Vector Extension for Embedded Processors.

As stated later in this specification, there exists a dependency between the newly defined extensions: `Zvfbfwma` depends on `Zfbfmin` and `Zvfbfmin`.

This initial set of BF16 extensions provides very basic functionality including scalar and vector conversion between BF16 and single-precision values, and vector widening multiply-accumulate instructions.

3.1. `Zfbfmin` - Scalar BF16 Converts

This extension provides the minimal set of instructions needed to enable scalar support of the BF16 format. It enables BF16 as an interchange format as it provides conversion between BF16 values and FP32 values.

This extension requires the single-precision floating-point extension `F`, and the `FLH`, `FSH`, `FMV.X.H`, and `FMV.H.X` instructions as defined in the `Zfh` extension.

While conversion instructions tend to include all supported formats, in these extensions we only support conversion between BF16 and FP32 as we are targeting a special use case. These extensions are intended to support the case where BF16 values are used as reduced precision versions of FP32 values, where use of BF16 provides a two-fold advantage for storage, bandwidth, and computation. In this use case, the BF16 values are typically multiplied by each other and accumulated into FP32 sums. These sums are typically converted to BF16 and then used as subsequent inputs. The operations on the BF16 values can be performed on the CPU or a loosely coupled coprocessor.

Subsequent extensions might provide support for native BF16 arithmetic. Such extensions could add additional conversion instructions to allow all supported formats to be converted to and from BF16.

BF16 addition, subtraction, multiplication, division, and square-root operations can be faithfully emulated by converting the BF16 operands to single-precision, performing the operation using single-precision arithmetic, and then converting back to BF16. Performing BF16 fused multiply-addition using this method can produce results that differ by 1-ulp on some inputs for the RNE and RMM rounding modes.

Conversions between BF16 and formats larger than FP32 can be emulated. Exact widening conversions from BF16 can be synthesized by first converting to FP32 and then converting from FP32 to the target precision. Conversions narrowing to BF16 can be synthesized by first converting to FP32 through a series of halving steps and then converting from FP32 to the target precision. As with the fused multiply-addition instruction described above, this method of converting values to BF16 can be off by 1-ulp on some inputs for the RNE and RMM rounding modes.

Mnemonic	Instruction
FCVT.BF16.S	Convert FP32 to BF16
FCVT.S.BF16	Convert BF16 to FP32
FLH	
FSH	
FMV.H.X	
FMV.X.H	

3.2. `Zvfbfmin` - Vector BF16 Converts

This extension provides the minimal set of instructions needed to enable vector support of the BF16 format. It enables BF16 as an interchange format as it provides conversion between BF16 values and FP32 values.

This extension requires either the "V" extension or the `Zve32f` embedded vector extension.

While conversion instructions tend to include all supported formats, in these extensions we only support conversion between BF16 and FP32 as we are targeting a special use case. These extensions are intended to support the case where BF16 values are used as reduced precision versions of FP32 values, where use of BF16 provides a two-fold advantage for storage, bandwidth, and computation. In this use case, the BF16 values are typically multiplied by each other and accumulated into FP32 sums. These sums are typically converted to BF16 and then used as subsequent inputs. The operations on the BF16 values can be performed on the CPU or a loosely coupled coprocessor.

Subsequent extensions might provide support for native BF16 arithmetic. Such extensions could add additional conversion instructions to allow all supported formats to be converted to and from BF16.

BF16 addition, subtraction, multiplication, division, and square-root operations can be faithfully emulated by converting the BF16 operands to single-precision, performing the operation using single-precision arithmetic, and then converting back to BF16. Performing BF16 fused multiply-addition using this method can produce results that differ by 1-ulp on some inputs for the RNE and RMM rounding modes.

Conversions between BF16 and formats larger than FP32 can be faithfully emulated. Exact widening conversions from BF16 can be synthesized by first converting to FP32 and then converting from FP32 to the target precision. Conversions narrowing to BF16 can be synthesized by first converting to FP32 through a series of halving steps using vector round-towards-odd narrowing conversion instructions (*vfncvt.rod.f.f.w*). The final convert from FP32 to BF16 would use the desired rounding mode.

Mnemonic	Instruction
vfncvtbf16.f.f.w	Vector convert FP32 to BF16
vfwcvtbf16.f.f.v	Vector convert BF16 to FP32

3.3. `Zvfbfwma` - Vector BF16 widening mul-add

This extension provides a vector widening BF16 mul-add instruction that accumulates into FP32.

This extension requires the `Zvfbfmin` extension and the `Zfbfmin` extension.

Mnemonic	Instruction
VFWMACCBF16	Vector BF16 widening multiply-accumulate

Chapter 4. Instructions

4.1. fcvt.bf16.s

Synopsis

Convert FP32 value to a BF16 value

Mnemonic

fcvt.bf16.s rd, rs1

Encoding

31 27	26 25	24 20	19 15	14 12	11 7	6 0
01000	10	01000	rs1	rm	rd	1010011
fcvt	h	bf16.s				OP-FP

Encoding

While the mnemonic of this instruction is consistent with that of the other RISC-V floating-point convert instructions, a new encoding is used in bits 24:20.

`BF16.S` and `H` are used to signify that the source is FP32 and the destination is BF16.

Description

Narrowing convert FP32 value to a BF16 value. Round according to the RM field.

This instruction is similar to other narrowing floating-point-to-floating-point conversion instructions.

Exceptions: Overflow, Underflow, Inexact, Invalid

Included in: Zfbfmin

4.2. fcvt.s.bf16

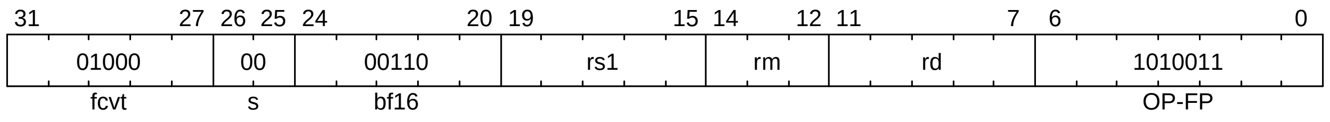
Synopsis

Convert BF16 value to an FP32 value

Mnemonic

fcvt.s.bf16 rd, rs1

Encoding



> *Encoding*
>
> While the mnemonic of this instruction is consistent with that of the other RISC-V floating-point convert instructions, a new encoding is used in bits 24:20 to indicate that the source is BF16.

Description

Converts a BF16 value to an FP32 value. The conversion is exact.

This instruction is similar to other widening floating-point-to-floating-point conversion instructions.

> If the input is normal or infinity, the BF16 encoded value is shifted to the left by 16 places and the least significant 16 bits are written with 0s.
>
> The result is NaN-boxed by writing the most significant `FLEN`-32 bits with 1s.

Exceptions: Invalid

Included in: Zfbfmin

4.3. vfncvtbf16.f.f.w

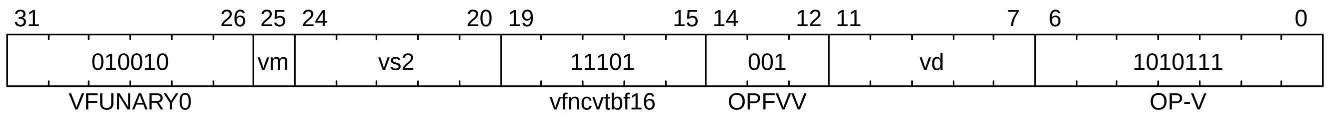
Synopsis

Vector convert FP32 to BF16

Mnemonic

vfncvtbf16.f.f.w vd, vs2, vm

Encoding



Reserved Encodings

- `SEW` is any value other than 16

Arguments

Register	Direction	EEW	Definition
Vs2	input	32	FP32 Source
Vd	output	16	BF16 Result

Description

Narrowing convert from FP32 to BF16. Round according to the *frm* register.

This instruction is similar to `vfncvt.f.f.w` which converts a floating-point value in a 2*SEW-width format into an SEW-width format. However, here the SEW-width format is limited to BF16.

Exceptions: Overflow, Underflow, Inexact, Invalid

Included in: Zvfbfmin

4.4. vfwcvtbf16.f.f.v

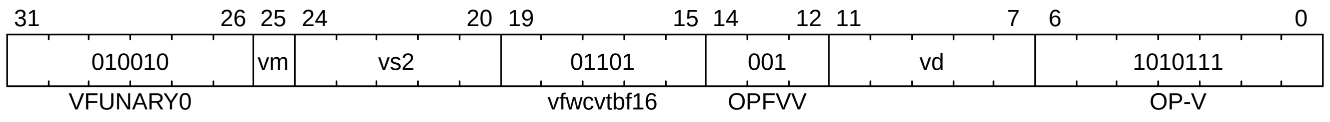
Synopsis

Vector convert BF16 to FP32

Mnemonic

vfwcvtbf16.f.f.v vd, vs2, vm

Encoding



Reserved Encodings

- `SEW` is any value other than 16

Arguments

Register	Direction	EEW	Definition
Vs2	input	16	BF16 Source
Vd	output	32	FP32 Result

Description

Widening convert from BF16 to FP32. The conversion is exact.

This instruction is similar to `vfwcvt.f.f.v` which converts a floating-point value in an SEW-width format into a 2*SEW-width format. However, here the SEW-width format is limited to BF16.

If the input is normal or infinity, the BF16 encoded value is shifted to the left by 16 places and the least significant 16 bits are written with 0s.

Exceptions: Invalid

Included in: Zvfbfmin

4.5. vfwmaccbf16

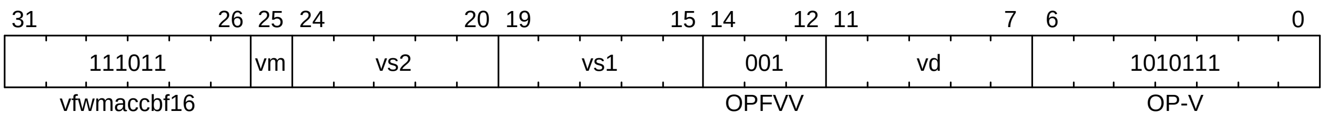
Synopsis

Vector BF16 widening multiply-accumulate

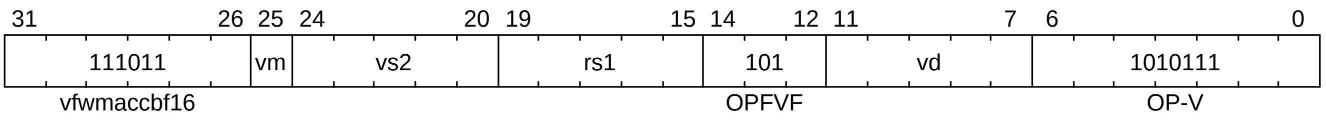
Mnemonic

vfwmaccbf16.vv vd, vs1, vs2, vm
vfwmaccbf16.vf vd, rs1, vs2, vm

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- `SEW` is any value other than 16

Arguments

Register	Direction	EEW	Definition
Vd	input	32	FP32 Accumulate
Vs1/rs1	input	16	BF16 Source
Vs2	input	16	BF16 Source
Vd	output	32	FP32 Result

Description

This instruction performs a widening fused multiply-accumulate operation, where each pair of BF16 values are multiplied and their unrounded product is added to the corresponding FP32 accumulate value. The sum is rounded according to the *frm* register.

In the vector-vector version, the BF16 elements are read from `vs1` and `vs2` and FP32 accumulate value is read from `vd`. The FP32 result is written to the destination register `vd`.

The vector-scalar version is similar, but instead of reading elements from `vs1`, a scalar BF16 value is read from the FPU register `rs1`.

Exceptions: Overflow, Underflow, Inexact, Invalid

Operation

This `vfwmaccbf16.vv` instruction is equivalent to widening each of the BF16 inputs to FP32 and then performing an FMACC as shown in the following instruction sequence:

```
vfwcvtbf16.f.f.v T1, vs1, vm
vfwcvtbf16.f.f.v T2, vs2, vm
vfmacc.vv        vd, T1, T2, vm
```

Likewise, `vfwmaccbf16.vf` is equivalent to the following instruction sequence:

```
fcvt.s.bf16      T1, rs1
vfwcvtbf16.f.f.v T2, vs2, vm
vfmacc.vf        vd, T1, T2, vm
```

Included in: Zvfbfwma

Bibliography

754-2019 - IEEE Standard for Floating-Point Arithmetic
754-2008 - IEEE Standard for Floating-Point Arithmetic