RISC-V Bit-Manipulation ISA-extensions

Version 1.0.0-38-g865e7a7, 2021-06-28: Release candidate

Table of Contents

Colophon

It describes the BitManip Zba, Zbb, Zbc and Zbs extensions being submitted for public review.

Acknowledgments

Contributors to this specification (in alphabetical order) include:
Jacob Bachmeyer, Allen Baum, Ari Ben, Alex Bradbury, Steven Braeger, Rogier Brussee, Michael Clark, Ken Dockser, Paul Donahue, Dennis Ferguson, Fabian Giesen, John Hauser, Robert Henry, Bruce Hoult, Po-wei Huang, Ben Marshall, Rex McCrary, Lee Moore, Jiří Moravec, Samuel Neves, Markus Oberhumer, Christopher Olson, Nils Pipenbrinck, Joseph Rahmeh, Xue Saw, Tommy Thorn, Philipp Tomsich, Avishai Tvila, Andrew Waterman, Thomas Wicki, and Claire Wolf.

We express our gratitude to everyone that contributed to, reviewed or improved this specification through their comments and questions.

Bit-manipulation a, b, c and s extensions grouped for public review and ratification

The bit-manipulation (bitmanip) extension collection is comprised of several component extensions to the base RISC-V architecture that are intended to provide some combination of code size reduction, performance improvement, and energy reduction. While the instructions are intended to have general use, some instructions are more useful in some domains than others. Hence, several smaller bitmanip extensions are provided, rather than one large extension. Each of these smaller extensions is grouped by common function and use case, and each has its own Zb*-extension name.

Each bitmanip extension includes a group of several bitmanip instructions that have similar purposes and that can often share the same logic. Some instructions are available in only one extension while others are available in several. The instructions have mnemonics and encodings that are independent of the extensions in which they appear. Thus, when implementing extensions with overlapping instructions, there is no redundancy in logic or encoding.

The bitmanip extensions are defined for RV32 and RV64. Most of the instructions are expected to be forward compatible with RV128. While the shift-immediate instructions are defined to have at most a 6-bit immediate field, a 7th bit is available in the encoding space should this be needed for RV128.

Word Instructions

The bitmanip extension follows the convention in RV64 that *w*-suffixed instructions (without a dot before the *w*) ignore the upper 32 bits of their inputs, operate on the least-significant 32-bits as signed values and produce a 32-bit signed result that is sign-extended to XLEN.

Bitmanip instructions with the suffix *.uw* have one operand that is an unsigned 32-bit value that is extracted from the least significant 32 bits of the specified register. Other than that, these perform full XLEN operations.

Bitmanip instructions with the suffix *.b*, *.h* and *.w* only look at the least significant 8-bits, 16-bits and 32-bits of the input (respectively) and produce an XLEN-wide result that is sign-extended or zero-extended, based on the specific instruction.

Pseudocode for instruction semantics

The semantics of each instruction in Instructions (in alphabetical order) is expressed in a SAIL-like syntax.

Chapter 1. Extensions

The first group of bitmanip extensions to be released for Public Review are:

- Address generation instructions
- Basic bit-manipulation
- Carry-less multiplication
- Single-bit instructions

Below is a list of all of the instructions (and pseudoinstructions) that are included in these extensions along with their specific mapping:

RV32	RV64	Mnemonic	Instruction	Zba	Zbb	Zbc	Zbs
	✓	add.uw *rd*, *rs1*, *rs2*	Add unsigned word	✓			
✓	✓	andn *rd*, *rs1*, *rs2*	AND with inverted operand		✓		
✓	✓	clmul *rd*, *rs1*, *rs2*	Carry-less multiply (low-part)			✓	
✓	✓	clmulh *rd*, *rs1*, *rs2*	Carry-less multiply (high-part)			✓	
✓	✓	clmulr *rd*, *rs1*, *rs2*	Carry-less multiply (reversed)			✓	
✓	✓	clz *rd*, *rs*	Count leading zero bits		✓		
	✓	clzw *rd*, *rs*	Count leading zero bits in word		✓		
✓	✓	cpop *rd*, *rs*	Count set bits		✓		
	✓	cpopw *rd*, *rs*	Count set bits in word		✓		
✓	✓	ctz *rd*, *rs*	Count trailing zero bits		✓		
	✓	ctzw *rd*, *rs*	Count trailing zero bits in word		✓		
✓	✓	max *rd*, *rs1*, *rs2*	Maximum		✓		
✓	✓	maxu *rd*, *rs1*, *rs2*	Unsigned maximum		✓		
✓	✓	min *rd*, *rs1*, *rs2*	Minimum		✓		
✓	✓	minu *rd*, *rs1*, *rs2*	Unsigned minimum		✓		
✓	✓	orc.b *rd*, *rs1*, *rs2*	Bitwise OR-Combine, byte granule		✓		
✓	✓	orn *rd*, *rs1*, *rs2*	OR with inverted operand		✓		
✓	✓	rev8 *rd*, *rs*	Byte-reverse register		✓		
✓	✓	rol *rd*, *rs1*, *rs2*	Rotate left (Register)		✓		
	✓	rolw *rd*, *rs1*, *rs2*	Rotate Left Word (Register)		✓		
✓	✓	ror *rd*, *rs1*, *rs2*	Rotate right (Register)		✓		
✓	✓	rori *rd*, *rs1*, *shamt*	Rotate right (Immediate)		✓		
	✓	roriw *rd*, *rs1*, *shamt*	Rotate right Word (Immediate)		✓		
	✓	rorw *rd*, *rs1*, *rs2*	Rotate right Word (Register)		✓		

RV32	RV64	Mnemonic	Instruction	Zba	Zbb	Zbc	Zbs
✓	✓	bclr *rd*, *rs1*, *rs2*	Single-Bit Clear (Register)				✓
✓	✓	bclri *rd*, *rs1*, *imm*	Single-Bit Clear (Immediate)				✓
✓	✓	bext *rd*, *rs1*, *rs2*	Single-Bit Extract (Register)				✓
✓	✓	bexti *rd*, *rs1*, *imm*	Single-Bit Extract (Immediate)				✓
✓	✓	binv *rd*, *rs1*, *rs2*	Single-Bit Invert (Register)				✓
✓	✓	binvi *rd*, *rs1*, *imm*	Single-Bit Invert (Immediate)				✓
✓	✓	bset *rd*, *rs1*, *rs2*	Single-Bit Set (Register)				✓
✓	✓	bseti *rd*, *rs1*, *imm*	Single-Bit Set (Immediate)				✓
✓	✓	sext.b *rd*, *rs*	Sign-extend byte		✓		
✓	✓	sext.h *rd*, *rs*	Sign-extend halfword		✓		
✓	✓	sh1add *rd*, *rs1*, *rs2*	Shift left by 1 and add	✓			
	✓	sh1add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 1 and add	✓			
✓	✓	sh2add *rd*, *rs1*, *rs2*	Shift left by 2 and add	✓			
	✓	sh2add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 2 and add	✓			
✓	✓	sh3add *rd*, *rs2*, *rs2*	Shift left by 3 and add	✓			
	✓	sh3add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 3 and add	✓			
	✓	slli.uw *rd*, *rs1*, *imm*	Shift-left unsigned word (Immediate)	✓			
✓	✓	xnor *rd*, *rs1*, *rs2*	Exclusive NOR		✓		
✓	✓	zext.h *rd*, *rs*	Zero-extend halfword		✓		

1.1. Zba extension

The Zba extension is frozen.

The Zba instructions can be used to accelerate the generation of addresses that index into arrays of basic types (halfword, word, doubleword) using both unsigned word-sized and XLEN-sized indices: a shifted index is added to a base address.

The shift and add instructions do a left shift of 1, 2, or 3 because these are commonly found in real-world code and because they can be implemented with a minimal amount of additional hardware beyond that of the simple adder. This avoids lengthening the critical path in implementations.

While the shift and add instructions are limited to a maximum left shift of 3, the slli instruction (from the base ISA) can be used to perform similar shifts for indexing into arrays of wider elements. The slli.uw — added in this extension — can be used when the index is to be interpreted as an unsigned word.

The following instructions comprise the Zba extension:

RV32	RV64	Mnemonic	Instruction
	✓	add.uw *rd*, *rs1*, *rs2*	Add unsigned word
✓	✓	sh1add *rd*, *rs1*, *rs2*	Shift left by 1 and add
	✓	sh1add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 1 and add
✓	✓	sh2add *rd*, *rs1*, *rs2*	Shift left by 2 and add
	✓	sh2add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 2 and add
✓	✓	sh3add *rd*, *rs2*, *rs2*	Shift left by 3 and add
	✓	sh3add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 3 and add
	✓	slli.uw *rd*, *rs1*, *imm*	Shift-left unsigned word (Immediate)

1.2. Zbb: Basic bit-manipulation

The Zbb extension is frozen.

1.2.1. Logical with negate

RV32	RV64	Mnemonic	Instruction
✓	✓	andn *rd*, *rs1*, *rs2*	AND with inverted operand
✓	✓	orn *rd*, *rs1*, *rs2*	OR with inverted operand
✓	✓	xnor *rd*, *rs1*, *rs2*	Exclusive NOR

Implementation Hint

The Logical with Negate instructions can be implemented by inverting the *rs2* inputs to the base-required AND, OR, and XOR logic instructions. In some implementations, the inverter on rs2 used for subtraction can be reused for this purpose.

1.2.2. Count leading/trailing zero bits

RV32	RV64	Mnemonic	Instruction
✓	✓	clz *rd*, *rs*	Count leading zero bits
	✓	clzw *rd*, *rs*	Count leading zero bits in word
✓	✓	ctz *rd*, *rs*	Count trailing zero bits
	✓	ctzw *rd*, *rs*	Count trailing zero bits in word

1.2.3. Count population

These instructions count the number of set bits (1-bits). This is also commonly referred to as population count.

RV32	RV64	Mnemonic	Instruction
✓	✓	cpop *rd*, *rs*	Count set bits

RV32	RV64	Mnemonic	Instruction
	✓	cpopw *rd*, *rs*	Count set bits in word

1.2.4. Integer minimum/maximum

The integer minimum/maximum instructions are arithmetic R-type instructions that return the smaller/larger of two operands.

RV32	RV64	Mnemonic	Instruction
✓	✓	max *rd*, *rs1*, *rs2*	Maximum
✓	✓	maxu *rd*, *rs1*, *rs2*	Unsigned maximum
✓	✓	min *rd*, *rs1*, *rs2*	Minimum
✓	✓	minu *rd*, *rs1*, *rs2*	Unsigned minimum

1.2.5. Sign- and zero-extension

These instructions perform the sign-extension or zero-extension of the least significant 8 bits, 16 bits or 32 bits of the source register.

These instructions replace the generalized idioms `slli rD,rS,(XLEN-<size>) + srli` (for zero-extension) or `slli + srai` (for sign-extension) for the sign-extension of 8-bit and 16-bit quantities, and for the zero-extension of 16-bit and 32-bit quantities.

RV32	RV64	Mnemonic	Instruction
✓	✓	sext.b *rd*, *rs*	Sign-extend byte
✓	✓	sext.h *rd*, *rs*	Sign-extend halfword
✓	✓	zext.h *rd*, *rs*	Zero-extend halfword

1.2.6. Bitwise rotation

Bitwise rotation instructions are similar to the shift-logical operations from the base spec. However, where the shift-logical instructions shift in zeros, the rotate instructions shift in the bits that were shifted out of the other side of the value. Such operations are also referred to as 'circular shifts'.

RV32	RV64	Mnemonic	Instruction
✓	✓	rol *rd*, *rs1*, *rs2*	Rotate left (Register)
	✓	rolw *rd*, *rs1*, *rs2*	Rotate Left Word (Register)
✓	✓	ror *rd*, *rs1*, *rs2*	Rotate right (Register)
✓	✓	rori *rd*, *rs1*, *shamt*	Rotate right (Immediate)
	✓	roriw *rd*, *rs1*, *shamt*	Rotate right Word (Immediate)
	✓	rorw *rd*, *rs1*, *rs2*	Rotate right Word (Register)

Architecture Explanation

The rotate instructions were included to replace a common four-instruction sequence to achieve the same effect (neg; sll/srl; srl/sll; or)

1.2.7. OR Combine

orc.b sets the bits of each byte in the result *rd* to all zeros if no bit within the respective byte of *rs* is set, or to all ones if any bit within the respective byte of *rs* is set.

One use-case is string-processing functions, such as **strlen** and **strcpy**, which can use **orc.b** to test for the terminating zero byte by counting the set bits in leading non-zero bytes in a word.

RV32	RV64	Mnemonic	Instruction
✓	✓	orc.b *rd*, *rs*	Bitwise OR-Combine, byte granule

1.2.8. Byte-reverse

rev8 reverses the byte-ordering of *rs*.

RV32	RV64	Mnemonic	Instruction
✓	✓	rev8 *rd*, *rs*	Byte-reverse register

1.3. Zbc: Carry-less multiplication

The Zbc extension is frozen.

Carry-less multiplication is the multiplication in the polynomial ring over GF(2).

clmul produces the lower half of the carry-less product and **clmulh** produces the upper half of the 2×XLEN carry-less product.

clmulr produces bits 2×XLEN−2:XLEN-1 of the 2×XLEN carry-less product.

RV32	RV64	Mnemonic	Instruction
✓	✓	clmul *rd*, *rs1*, *rs2*	Carry-less multiply (low-part)
✓	✓	clmulh *rd*, *rs1*, *rs2*	Carry-less multiply (high-part)
✓	✓	clmulr *rd*, *rs1*, *rs2*	Carry-less multiply (reversed)

1.4. Zbs: Single-bit instructions

The Zbs extension is frozen.

The single-bit instructions provide a mechanism to set, clear, invert, or extract a single bit in a register. The bit is specified by its index.

RV32	RV64	Mnemonic	Instruction
✓	✓	bclr *rd*, *rs1*, *rs2*	Single-Bit Clear (Register)
✓	✓	bclri *rd*, *rs1*, *imm*	Single-Bit Clear (Immediate)
✓	✓	bext *rd*, *rs1*, *rs2*	Single-Bit Extract (Register)
✓	✓	bexti *rd*, *rs1*, *imm*	Single-Bit Extract (Immediate)
✓	✓	binv *rd*, *rs1*, *rs2*	Single-Bit Invert (Register)
✓	✓	binvi *rd*, *rs1*, *imm*	Single-Bit Invert (Immediate)
✓	✓	bset *rd*, *rs1*, *rs2*	Single-Bit Set (Register)
✓	✓	bseti *rd*, *rs1*, *imm*	Single-Bit Set (Immediate)

Chapter 2. Instructions (in alphabetical order)

2.1. add.uw

Synopsis

Add unsigned word

Mnemonic

add.uw *rd*, *rs1*, *rs2*

Pseudoinstructions

zext.w *rd*, *rs1* → add.uw *rd*, *rs1*, zero

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 0 0 0 1 0 0	rs2	rs1	0 0 0	rd	0 1 1 1 0 1 1
ADD.UW			ADD.UW		OP-32

Description

This instruction performs an XLEN-wide addition between *rs2* and the zero-extended least-significant word of *rs1*.

Operation

```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);

X(rd) = base + index;
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Frozen

2.2. andn

Synopsis

AND with inverted operand

Mnemonic

andn *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1
ANDN			ANDN		OP

Description

This instruction performs the bitwise logical AND operation between *rs1* and the bitwise inversion of *rs2*.

Operation

```
X(rd) = X(rs1) & ~X(rs2);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.3. bclr

Synopsis

Single-Bit Clear (Register)

Mnemonic

bclr *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 0 0 1 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1
BCLR/BEXT			BCLR		OP

Description

This instruction returns *rs1* with a single bit cleared at the index specified in *rs2*. The index is read from the lower log2(XLEN) bits of *rs2*.

Operation

```
let index = X(rs2) & (XLEN - 1);
X(rd) = X(rs1) & ~(1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	0.93	Frozen

2.4. bclri

Synopsis

Single-Bit Clear (Immediate)

Mnemonic

bclri *rd*, *rs1*, *shamt*

Encoding (RV32)

31–25	24–20	19–15	14–12	11–7	6–0
0 1 0 0 1 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1
BCLRI			BCLRI		OP-IMM

Encoding (RV64)

31–26	25–20	19–15	14–12	11–7	6–0
0 1 0 0 1 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1
BCLRI			BCLRI		OP-IMM

Description

This instruction returns *rs1* with a single bit cleared at the index specified in *shamt*. The index is read from the lower log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Operation

```
let index = shamt & (XLEN - 1);
X(rd) = X(rs1) & ~(1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	0.93	Frozen

2.5. bext

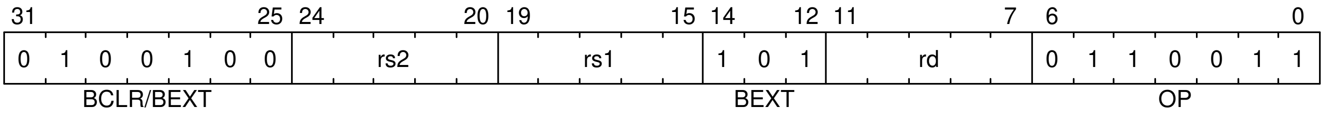
Synopsis

Single-Bit Extract (Register)

Mnemonic

bext *rd*, *rs1*, *rs2*

Encoding



Description

This instruction returns a single bit extracted from *rs1* at the index specified in *rs2*. The index is read from the lower log2(XLEN) bits of *rs2*.

Operation

```
let index = X(rs2) & (XLEN - 1);
X(rd) = (X(rs1) >> index) & 1;
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	0.93	Frozen

2.6. bexti

Synopsis

Single-Bit Extract (Immediate)

Mnemonic

bexti *rd*, *rs1*, *shamt*

Encoding (RV32)

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0 1 0 0 1 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1
BEXTI/BCLRI			BEXTI		OP-IMM

Encoding (RV64)

31 … 26	25 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0 1 0 0 1 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1
BEXTI/BCLRI			BEXTI		OP-IMM

Description

This instruction returns a single bit extracted from *rs1* at the index specified in *rs2*. The index is read from the lower log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Operation

```
let index = shamt & (XLEN - 1);
X(rd) = (X(rs1) >> index) & 1;
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	0.93	Frozen

2.7. binv

Synopsis

Single-Bit Invert (Register)

Mnemonic

binv *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 1 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1
BINV			BINV		OP

Description

This instruction returns *rs1* with a single bit inverted at the index specified in *rs2*. The index is read from the lower log2(XLEN) bits of *rs2*.

Operation

```
let index = X(rs2) & (XLEN - 1);
X(rd) = X(rs1) ^ (1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	0.93	Frozen

2.8. binvi

Synopsis

Single-Bit Invert (Immediate)

Mnemonic

binvi *rd*, *rs1*, *shamt*

Encoding (RV32)

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 1 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1
BINVI			BINV		OP-IMM

Encoding (RV64)

31–26	25–20	19–15	14–12	11–7	6–0
0 1 1 0 1 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1
BINVI			BINV		OP-IMM

Description

This instruction returns *rs1* with a single bit inverted at the index specified in *shamt*. The index is read from the lower log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Operation

```
let index = shamt & (XLEN - 1);
X(rd) = X(rs1) ^ (1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	0.93	Frozen

2.9. bset

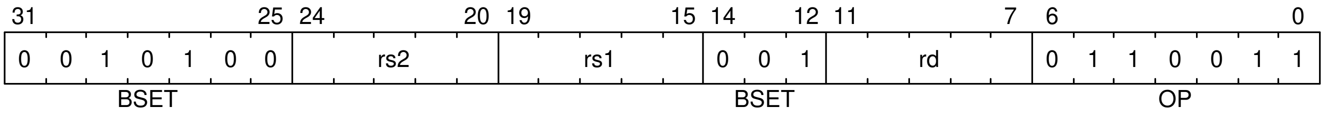
Synopsis

Single-Bit Set (Register)

Mnemonic

bset *rd*, *rs1*,*rs2*

Encoding



Description

This instruction returns *rs1* with a single bit set at the index specified in *rs2*. The index is read from the lower log2(XLEN) bits of *rs2*.

Operation

```
let index = X(rs2) & (XLEN - 1);
X(rd) = X(rs1) | (1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	0.93	Frozen

2.10. bseti

Synopsis

Single-Bit Set (Immediate)

Mnemonic

bseti *rd*, *rs1*,*shamt*

Encoding (RV32)

31–25	24–20	19–15	14–12	11–7	6–0
0 0 1 0 1 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1
BSETI			BSETI		OP-IMM

Encoding (RV64)

31–26	25–20	19–15	14–12	11–7	6–0
0 0 1 0 1 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1
BSETI			BSETI		OP-IMM

Description

This instruction returns *rs1* with a single bit set at the index specified in *shamt*. The index is read from the lower log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Operation

```
let index = shamt & (XLEN - 1);
X(rd) = X(rs1) | (1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	0.93	Frozen

2.11. clmul

Synopsis

Carry-less multiply (low-part)

Mnemonic

clmul *rd*, *rs1*, *rs2*

Encoding

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
0 0 0 0 1 0 1	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1
MINMAX/CLMUL			CLMUL		OP

Description

clmul produces the lower half of the 2·XLEN carry-less product.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);
let output : xlenbits = 0;

foreach (i from 0 to xlen by 1) {
   output = if   ((rs2_val >> i) & 1)
            then output ^ (rs1_val << i);
        else output;
}

X[rd] = output
```

Included in

Extension	Minimum version	Lifecycle state
Zbc (Carry-less multiplication)	0.93	Frozen

2.12. clmulh

Synopsis

Carry-less multiply (high-part)

Mnemonic

clmulh *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 0 0 0 1 0 1	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1
MINMAX/CLMUL			CLMULH		OP

Description

clmulh produces the upper half of the 2·XLEN carry-less product.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);
let output : xlenbits = 0;

foreach (i from 1 to xlen by 1) {
   output = if   ((rs2_val >> i) & 1)
            then output ^ (rs1_val >> (xlen - i));
        else output;
}

X[rd] = output
```

Included in

Extension	Minimum version	Lifecycle state
Zbc (Carry-less multiplication)	0.93	Frozen

2.13. clmulr

Synopsis

Carry-less multiply (reversed)

Mnemonic

clmulr *rd*, *rs1*, *rs2*

Encoding

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0 0 0 0 1 0 1	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1
MINMAX/CLMUL			CLMULR		OP

Description

clmulr produces bits 2·XLEN—2:XLEN-1 of the 2·XLEN carry-less product.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);
let output : xlenbits = 0;

foreach (i from 0 to (xlen - 1) by 1) {
   output = if   ((rs2_val >> i) & 1)
            then output ^ (rs1_val >> (xlen - i - 1));
        else output;
}

X[rd] = output
```

Note

The **clmulr** instruction is used to accelerate CRC calculations. The **r** in the instruction's mnemonic stands for *reversed*, as the instruction is equivalent to bit-reversing the inputs, performing a **clmul**, then bit-reversing the output.

Included in

Extension	Minimum version	Lifecycle state
Zbc (Carry-less multiplication)	0.93	Frozen

2.14. clz

Synopsis

Count leading zero bits

Mnemonic

clz *rd*, *rs*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0 0	0 0 0 0 0	rs1	0 0 1	rd	0 0 1 0 0 1 1
CLZ	CLZ		CLZ		OP-IMM

Description

This instruction counts the number of 0's before the first 1, starting at the most-significant bit (i.e., XLEN-1) and progressing to bit 0. Accordingly, if the input is 0, the output is XLEN, and if the most-significant bit of the input is a 1, the output is 0.

Operation

```
val HighestSetBit : forall ('N : Int), 'N >= 0. bits('N) -> int

function HighestSetBit x = {
  foreach (i from (xlen - 1) to 0 by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return -1;
}

let rs = X(rs);
X[rd] = (xlen - 1) - HighestSetBit(rs);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.15. clzw

Synopsis

Count leading zero bits in word

Mnemonic

clzw *rd*, *rs*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0 0	0 0 0 0 0	rs1	0 0 1	rd	0 0 1 1 0 1 1
CLZW	CLZW		CLZW		OP-IMM-32

Description

This instruction counts the number of 0's before the first 1 starting at bit 31 and progressing to bit 0. Accordingly, if the least-significant word is 0, the output is 32, and if the most-significant bit of the word (i.e., bit 31) is a 1, the output is 0.

Operation

```
val HighestSetBit32 : forall ('N : Int), 'N >= 0. bits('N) -> int

function HighestSetBit32 x = {
  foreach (i from 31 to 0 by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return -1;
}

let rs = X(rs);
X[rd] = 31 - HighestSetBit(rs);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.16. cpop

Synopsis

Count set bits

Mnemonic

cpop *rd*, *rs*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0 0	0 0 0 1 0	rs1	0 0 1	rd	0 0 1 0 0 1 1
CPOP	CPOP		CPOP		OP-IMM

Description

This instructions counts the number of 1's (i.e., set bits) in the source register.

Operation

```
let bitcount = 0;
let rs = X(rs);

foreach (i from 0 to (xlen - 1) in inc)
    if rs[i] == 0b1 then bitcount = bitcount + 1 else ();

X[rd] = bitcount
```

Software Hint

This operations is known as population count, popcount, sideways sum, bit summation, or Hamming weight.

The GCC builtin function `__builtin_popcount (unsigned int x)` is implemented by cpop on RV32 and by **cpopw** on RV64. The GCC builtin function `__builtin_popcountl (unsigned long x)` for LP64 is implemented by **cpop** on RV64.

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.17. cpopw

Synopsis

Count set bits in word

Mnemonic

cpopw *rd*, *rs*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0 0	0 0 0 1 0	rs	0 0 1	rd	0 0 1 1 0 1 1
CPOPW	CPOPW		CPOPW		OP-IMM-32

Description

This instructions counts the number of 1's (i.e., set bits) in the least-significant word of the source register.

Operation

```
let bitcount = 0;
let val = X(rs);

foreach (i from 0 to 31 in inc)
    if val[i] == 0b1 then bitcount = bitcount + 1 else ();

X[rd] = bitcount
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.18. ctz

Synopsis

Count trailing zeros

Mnemonic

ctz *rd*, *rs*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0 0	0 0 0 0 1	rs1	0 0 1	rd	0 0 1 0 0 1 1
CTZ/CTZW	CTZ/CTZW		CTZ/CTZW		OP-IMM

Description

This instruction counts the number of 0's before the first 1, starting at the least-significant bit (i.e., 0) and progressing to the most-significant bit (i.e., XLEN-1). Accordingly, if the input is 0, the output is XLEN, and if the least-significant bit of the input is a 1, the output is 0.

Operation

```
val LowestSetBit : forall ('N : Int), 'N >= 0. bits('N) -> int

function LowestSetBit x = {
  foreach (i from 0 to (xlen - 1) by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return xlen;
}

let rs = X(rs);
X[rd] = LowestSetBit(rs);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.19. ctzw

Synopsis

Count trailing zero bits in word

Mnemonic

ctzw *rd*, *rs*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0 0	0 0 0 0 1	rs1	0 0 1	rd	0 0 1 1 0 1 1
CTZ/CTZW	CTZ/CTZW		CTZ/CTZW		OP-IMM-32

Description

This instruction counts the number of 0's before the first 1, starting at the least-significant bit (i.e., 0) and progressing to the most-significant bit of the least-significant word (i.e., 31). Accordingly, if the least-significant word is 0, the output is 32, and if the least-significant bit of the input is a 1, the output is 0.

Operation

```
val LowestSetBit32 : forall ('N : Int), 'N >= 0. bits('N) -> int

function LowestSetBit32 x = {
  foreach (i from 0 to 31 by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return 32;
}

let rs = X(rs);
X[rd] = LowestSetBit32(rs);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.20. max

Synopsis

Maximum

Mnemonic

max *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 0 0 0 1 0 1	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1
MINMAX/CLMUL			MAX		OP

Description

This instruction returns the larger of two signed integers.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_s rs2_val
             then rs2_val
         else rs1_val;

X(rd) = result;
```

Software Hint

Calculating the absolute value of a signed integer can be performed using the following sequence: **neg rD,rS** followed by **max rD,rS,rD**. When using this common sequence, it is suggested that they are scheduled with no intervening instructions so that implementations that are so optimized can fuse them together.

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.21. maxu

Synopsis

Unsigned maximum

Mnemonic

maxu *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 0 0 0 1 0 1	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1
MINMAX/CLMUL			MAXU		OP

Description

This instruction returns the larger of two unsigned integers.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_u rs2_val
             then rs2_val
         else rs1_val;

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.22. min

Synopsis

Minimum

Mnemonic

min *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 0 0 0 1 0 1	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1
MINMAX/CLMUL			MIN		OP

Description

This instruction returns the smaller of two signed integers.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_s rs2_val
             then rs1_val
         else rs2_val;

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.23. minu

Synopsis

Unsigned minimum

Mnemonic

minu *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 0 0 0 1 0 1	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1
MINMAX/CLMUL			MINU		OP

Description

This instruction returns the smaller of two unsigned integers.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_u rs2_val
             then rs1_val
         else rs2_val;

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.24. orc.b

Synopsis

Bitwise OR-Combine, byte granule

Mnemonic

orc.b *rd*, *rs*

Encoding

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0 0 1 0 1 0 0 0 0 1 1 1	rs	1 0 1	rd	0 0 1 0 0 1 1
				OP-IMM

Description

Combines the bits within each byte using bitwise logical OR. This sets the bits of each byte in the result *rd* to all zeros if no bit within the respective byte of *rs* is set, or to all ones if any bit within the respective byte of *rs* is set.

Operation

```
let input = X(rs);
let output : xlenbits = 0;

foreach (i from 0 to (xlen - 8) by 8) {
   output[(i + 7)..i] = if   input[(i + 7)..i] == 0
                        then 0b00000000
                        else 0b11111111;
}

X[rd] = output;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.25. orn

Synopsis

OR with inverted operand

Mnemonic

orn *rd*, *rs1*, *rs2*

Encoding

31 25	24 20	19 15	14 12	11 7	6 0
0 1 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1
ORN			ORN		OP

Description

This instruction performs the bitwise logical AND operation between *rs1* and the bitwise inversion of *rs2*.

Operation

```
X(rd) = X(rs1) | ~X(rs2);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.26. rev8

Synopsis

Byte-reverse register

Mnemonic

rev8 *rd*, *rs*

Encoding (RV32)

31–20	19–15	14–12	11–7	6–0
0 1 1 0 1 0 0 1 1 0 0 0	rs	1 0 1	rd	0 0 1 0 0 1 1
				OP-IMM

Encoding (RV64)

31–20	19–15	14–12	11–7	6–0
0 1 1 0 1 0 1 1 1 0 0 0	rs	1 0 1	rd	0 0 1 0 0 1 1
				OP-IMM

Description

This instruction reverses the order of the bytes in *rs*.

Operation

```
let input = X(rs);
let output : xlenbits = 0;
let j = xlen - 1;

foreach (i from 0 to (xlen - 8) by 8) {
   output[i..(i + 7)] = input[(j - 7)..j];
   j = j - 8;
}

X[rd] = output
```

Note

The **rev8** mnemonic corresponds to different instruction encodings in RV32 and RV64.

Software Hint

The byte-reverse operation is only available for the full register width. To emulate word-sized and halfword-sized byte-reversal, perform a `rev8 rd,rs` followed by a `srai rd,rd`.

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.27. rol

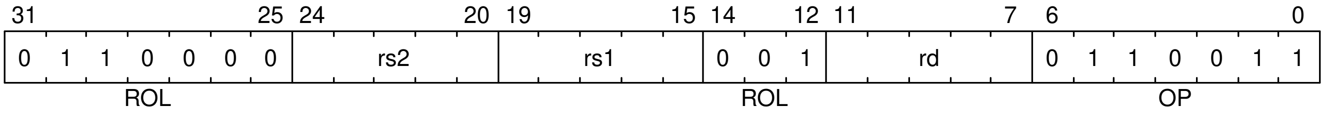
Synopsis

Rotate Left (Register)

Mnemonic

rol *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate left of *rs1* by the amount in least-significant log2(XLEN) bits of *rs2*.

Operation

```
let shamt = if   xlen == 32
            then X(rs2)[4..0]
        else X(rs2)[5..0];
let result = (X(rs1) << shamt) | (X(rs1) >> (xlen - shamt));

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.28. rolw

Synopsis

Rotate Left Word (Register)

Mnemonic

rolw *rd*, *rs1*, *rs2*

Encoding

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0 1 1 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 1 0 1 1
ROLW			ROLW		OP-32

Description

This instruction performs a rotate left on the least-significant word of *rs1* by the amount in least-significant 5 bits of *rs2*. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

Operation

```
let rs1 = EXTZ(X(rs1)[31..0])
let shamt = X(rs2)[4..0];
let result = (rs1 << shamt) | (rs1 >> (32 - shamt));
X(rd) = EXTS(result);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.29. ror

Synopsis

Rotate Right

Mnemonic

ror *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1
ROR			ROR		OP

Description

This instruction performs a rotate right of *rs1* by the amount in least-significant log2(XLEN) bits of *rs2*.

Operation

```
let shamt = if   xlen == 32
            then X(rs2)[4..0]
        else X(rs2)[5..0];
let result = (X(rs1) >> shamt) | (X(rs1) << (xlen - shamt));

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.30. rori

Synopsis

Rotate Right (Immediate)

Mnemonic

rori *rd*, *rs1*, *shamt*

Encoding (RV32)

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1
RORI			RORI		OP-IMM

Encoding (RV64)

31–26	25–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1
RORI			RORI		OP-IMM

Description

This instruction performs a rotate right of *rs1* by the amount in the least-significant log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Operation

```
let shamt = if   xlen == 32
            then shamt[4..0]
        else shamt[5..0];
let result = (X(rs1) >> shamt) | (X(rs1) << (xlen - shamt));

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.31. roriw

Synopsis

Rotate Right Word by Immediate

Mnemonic

roriw *rd*, *rs1*, *shamt*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 1 0 1 1
RORIW			RORIW		OP-IMM-32

Description

This instruction performs a rotate right on the least-significant word of *rs1* by the amount in the least-significant log2(XLEN) bits of *shamt*. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

Operation

```
let rs1_data = EXTZ(X(rs1)[31..0];
let result = (rs1_data >> shamt[4..0]) | (rs1_data << (32 - shamt[4..0]));
X(rd) = EXTS(result[31..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.32. rorw

Synopsis

Rotate Right Word (Register)

Mnemonic

rorw *rd*, *rs1*, *rs2*

Encoding

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
0 1 1 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 1 0 1 1
RORW			RORW		OP-32

Description

This instruction performs a rotate right on the least-significant word of *rs1* by the amount in least-significant 5 bits of *rs2*. The resultant word is sign-extended by copying bit 31 to all of the more-significant bits.

Operation

```
let rs1 = EXTZ(X(rs1)[31..0])
let shamt = X(rs2)[4..0];
let result = (rs1 >> shamt) | (rs1 << (32 - shamt));
X(rd) = EXTS(result);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.33. sext.b

Synopsis

Sign-extend byte

Mnemonic

sext.b *rd*, *rs*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0 0	0 0 1 0 0	rs1	0 0 1	rd	0 0 1 0 0 1 1
	SEXT.B		SEXT.B/SEXT.H		OP-IMM

Description

This instruction sign-extends the least-significant byte in the source to XLEN by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits.

Operation

```
X(rd) = EXTS(X(rs)[7..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.34. sext.h

Synopsis

Sign-extend halfword

Mnemonic

sext.h *rd*, *rs*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 1 0 0 0 0	0 0 1 0 1	rs1	0 0 1	rd	0 0 1 0 0 1 1
SEXT.H			SEXT.B/SEXT.H		OP-IMM

Description

This instruction sign-extends the least-significant halfword in *rs* to XLEN by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits.

Operation

```
X(rd) = EXTS(X(rs)[15..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.35. sh1add

Synopsis

Shift left by 1 and add

Mnemonic

sh1add *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 0 1 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1
SH1ADD			SH1ADD		OP

Description

This instruction shifts *rs1* to the left by 1 bit and adds it to *rs2*.

Operation

```
X(rd) = X(rs2) + (X(rs1) << 1);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Frozen

2.36. sh1add.uw

Synopsis

Shift unsigned word left by 1 and add

Mnemonic

sh1add.uw *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 0 1 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 1 0 1 1
SH1ADD.UW			SH1ADD.UW		OP-32

Description

This instruction performs an XLEN-wide addition of two addends. The first addend is *rs2*. The second addend is the unsigned value formed by extracting the least-significant word of *rs1* and shifting it left by 1 place.

Operation

```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);

X(rd) = base + (index << 1);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Frozen

2.37. sh2add

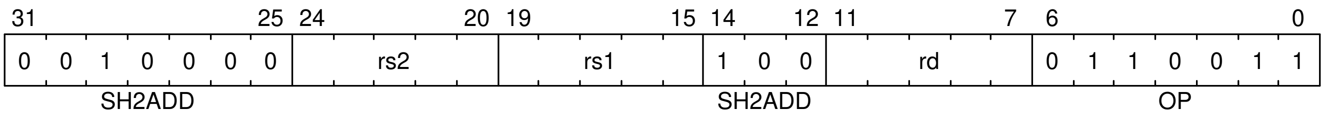
Synopsis

Shift left by 2 and add

Mnemonic

sh2add *rd*, *rs1*, *rs2*

Encoding



Description

This instruction shifts *rs1* to the left by 2 places and adds it to *rs2*.

Operation

```
X(rd) = X(rs2) + (X(rs1) << 2);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Frozen

2.38. sh2add.uw

Synopsis

Shift unsigned word left by 2 and add

Mnemonic

sh2add.uw *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 0 1 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 1 0 1 1
SH2ADD.UW			SH2ADD.UW		OP-32

Description

This instruction performs an XLEN-wide addition of two addends. The first addend is *rs2*. The second addend is the unsigned value formed by extracting the least-significant word of *rs1* and shifting it left by 2 places.

Operation

```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);

X(rd) = base + (index << 2);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Frozen

2.39. sh3add

Synopsis

Shift left by 3 and add

Mnemonic

sh3add *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 0 1 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1
SH3ADD			SH3ADD		OP

Description

This instruction shifts *rs1* to the left by 3 places and adds it to *rs2*.

Operation

```
X(rd) = X(rs2) + (X(rs1) << 3);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Frozen

2.40. sh3add.uw

Synopsis

Shift unsigned word left by 3 and add

Mnemonic

sh3add.uw *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 0 1 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 1 0 1 1
SH3ADD.UW			SH3ADD.UW		OP-32

Description

This instruction performs an XLEN-wide addition of two addends. The first addend is *rs2*. The second addend is the unsigned value formed by extracting the least-significant word of *rs1* and shifting it left by 3 places.

Operation

```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);

X(rd) = base + (index << 3);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Frozen

2.41. slli.uw

Synopsis

Shift-left unsigned word (Immediate)

Mnemonic

slli.uw *rd*, *rs1*, *shamt*

Encoding

31–26	25–20	19–15	14–12	11–7	6–0
0 0 0 0 1 0	shamt	rs1	0 0 1	rd	0 0 1 1 0 1 1
SLLI.UW			SLLI.UW		OP-IMM-32

Description

This instruction takes the least-significant word of *rs1*, zero-extends it, and shifts it left by the immediate.

Operation

```
X(rd) = (EXTZ(X(rs)[31..0]) << shamt);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Frozen

Architecture Explanation

This instruction is the same as **slli** with **zext.w** performed on *rs1* before shifting.

2.42. xnor

Synopsis

Exclusive NOR

Mnemonic

xnor *rd*, *rs1*, *rs2*

Encoding

31–25	24–20	19–15	14–12	11–7	6–0
0 1 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1
XNOR			XNOR		OP

Description

This instruction performs the bit-wise exclusive-NOR operation on *rs1* and *rs2*.

Operation

```
X(rd) = ~(X(rs1) ^ X(rs2));
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

2.43. zext.h

Synopsis

Zero-extend halfword

Mnemonic

zext.h *rd*, *rs*

Encoding (RV32)

31–25	24–20	19–15	14–12	11–7	6–0
0 0 0 0 1 0 0	0 0 0 0 0	rs	1 0 0	rd	0 1 1 0 0 1 1
			ZEXT.H		OP

Encoding (RV64)

31–25	24–20	19–15	14–12	11–7	6–0
0 0 0 0 1 0 0	0 0 0 0 0	rs	1 0 0	rd	0 1 1 1 0 1 1
			ZEXT.H		OP-32

Description

This instruction zero-extends the least-significant halfword of the source to XLEN by inserting 0's into all of the bits more significant than 15.

Operation

```
X(rd) = EXTZ(X(rs)[15..0]);
```

Note

The **zext.h** mnemonic corresponds to different instruction encodings in RV32 and RV64.

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Frozen

Appendix A: Software optimization guide

A.1. strlen

The **orc.b** instruction allows for the efficient detection of **NUL** bytes in an XLEN-sized chunk of data:

- the result of **orc.b** on a chunk that does not contain any **NUL** bytes will be all-ones, and
- after a bitwise-negation of the result of **orc.b**, the number of data bytes before the first **NUL** byte (if any) can be detected by **ctz**/**clz** (depending on the endianness of data).

A full example of a **strlen** function, which uses these techniques and also demonstrates the use of it for unaligned/partial data, is the following:

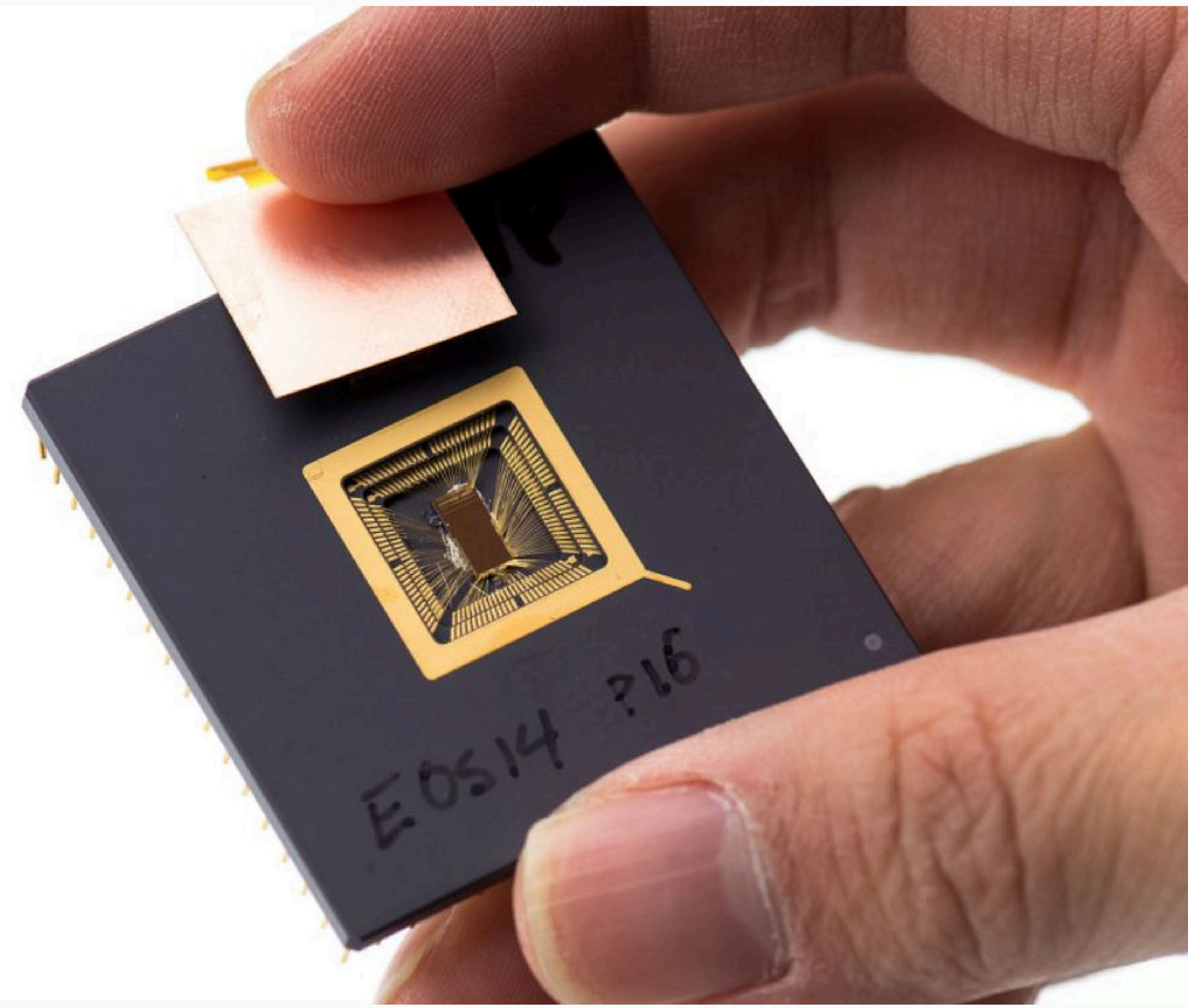
```
#include <sys/asm.h>

    .text
    .globl strlen
    .type  strlen, @function
strlen:
    andi    a3, a0, (SZREG-1)   // offset
    andi    a1, a0, -SZREG      // align pointer
.Lprologue:
    li      a4, SZREG
    sub     a4, a4, a3          // XLEN - offset
    slli    a3, a3, PTRLOG      // offset * 8
    REG_L   a2, 0(a1)           // chunk
    /*
     * Shift the partial/unaligned chunk we loaded to remove the bytes
     * from before the start of the string, adding NUL bytes at the end.
     */
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    srl a2, a2 ,a3          // chunk >> (offset * 8)
#else
    sll     a2, a2, a3
#endif
    orc.b   a2, a2
    not a2, a2
    /*
     * Non-NUL bytes in the string have been expanded to 0x00, while
     * NUL bytes have become 0xff.  Search for the first set bit
     * (corresponding to a NUL byte in the original chunk).
     */
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    ctz     a2, a2
#else
    clz     a2, a2
#endif
```

```
    /*
     * The first chunk is special: compare against the number of valid
     * bytes in this chunk.
     */
    srli    a0, a2, 3
    bgtu    a4, a0, .Ldone
    addi    a3, a1, SZREG
    li      a4, -1
    .align 2
    /*
     * Our critical loop is 4 instructions and processes data in 4 byte
     * or 8 byte chunks.
     */
.Lloop:
    REG_L   a2, SZREG(a1)
    addi    a1, a1, SZREG
    orc.b   a2, a2
    beq     a2, a4, .Lloop

.Lepilogue:
    not     a2, a2
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    ctz     a2, a2
#else
    clz     a2, a2
#endif
    sub     a1, a1, a3
    add a0, a0, a1
    srli    a2, a2, 3
    add     a0, a0, a2
.Ldone:
    ret
```

A.2. strcmp

```
#include <sys/asm.h>

  .text
  .globl strcmp
  .type  strcmp, @function
strcmp:
  or    a4, a0, a1
  li    t2, -1
  and   a4, a4, SZREG-1
  bnez  a4, .Lsimpleloop
```

```
  # Main loop for aligned strings
.Lloop:
  REG_L a2, 0(a0)
  REG_L a3, 0(a1)
  orc.b t0, a2
  bne   t0, t2, .Lfoundnull
  addi  a0, a0, SZREG
  addi  a1, a1, SZREG
  beq   a2, a3, .Lloop

  # Words don't match, and no null byte in first word.
  # Get bytes in big-endian order and compare.
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  rev8  a2, a2
  rev8  a3, a3
#endif
  # Synthesize (a2 >= a3) ? 1 : -1 in a branchless sequence.
  sltu a0, a2, a3
  neg  a0, a0
  ori  a0, a0, 1
  ret

.Lfoundnull:
  # Found a null byte.
  # If words don't match, fall back to simple loop.
  bne   a2, a3, .Lsimpleloop

  # Otherwise, strings are equal.
  li    a0, 0
  ret

  # Simple loop for misaligned strings
.Lsimpleloop:
  lbu   a2, 0(a0)
  lbu   a3, 0(a1)
  addi  a0, a0, 1
  addi  a1, a1, 1
  bne   a2, a3, 1f
  bnez  a2, .Lsimpleloop

1:
  sub   a0, a2, a3
  ret

.size   strcmp, .-strcmp
```


E0514