RISC-V Bitmanip Extension

Document Version 0.94-draft

Editor: Claire Wolf
Symbiotic GmbH
`claire@symbioticeda.com`

January 20, 2021

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): 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, Avishai Tvila, Andrew Waterman, Thomas Wicki, and Claire Wolf.

Contents

1 Introduction **1**

1.1 ISA Extension Proposal Design Criteria . . . 1

1.2 B Extension Adoption Strategy . . . 2

1.3 Next steps . . . 2

2 RISC-V Bitmanip Extension **3**

2.1 Basic bit manipulation instructions . . . 5

2.1.1 Count Leading/Trailing Zeros (`clz, ctz`) . . . 5

2.1.2 Count Bits Set (`cpop`) . . . 6

2.1.3 Logic-with-negate (`andn`, `orn`, `xnor`) . . . 6

2.1.4 Pack two words in one register (`pack, packu, packh`) . . . 7

2.1.5 Min/max instructions (`min, max, minu, maxu`) . . . 8

2.1.6 Sign-extend instructions (`sext.b, sext.h`) . . . 9

2.1.7 Single-bit instructions (`bset, bclr, binv, bext`) . . . 10

2.1.8 Shift Ones (Left/Right) (`slo, sloi, sro, sroi`) . . . 11

2.2 Bit permutation instructions . . . 11

2.2.1 Rotate (Left/Right) (`rol, ror, rori`) . . . 13

2.2.2 Generalized Reverse (`grev`, `grevi`, `rev`) . . . 14

2.2.3 Generalized Shuffle (`shfl`, `unshfl`, `shfli`, `unshfli`, `zip`, `unzip`) . . . 18

2.2.4 Crossbar Permutation Instructions (`xperm.[nbhw]`) . . . 25

2.3 Generalized OR-Combine (`gorc, gorci`) . . . 27

2.4 Bit-Field Place (`bfp`) 28

2.5 Bit Compress/Decompress (`bcompress, bdecompress`) 29

2.6 Carry-Less Multiply (`clmul, clmulh, clmulr`) 31

2.7 CRC Instructions (`crc32.[bhwd], crc32c.[bhwd]`) 32

2.8 Bit-Matrix Instructions (`bmatxor, bmator, bmatflip`, RV64 only) 34

2.9 Ternary Bit-Manipulation Instructions 36

2.9.1 Conditional Mix (`cmix`) 36

2.9.2 Conditional Move (`cmov`) 37

2.9.3 Funnel Shift (`fsl`, `fsr`, `fsri`) 38

2.10 Address calculation instructions 39

2.11 Add/shift with prefix zero-extend (`add.uw`, `slli.uw`) 40

2.12 Opcode Encodings 41

2.13 Future compressed instructions 46

2.14 Macro-op fusion patterns 47

2.14.1 Fused `*-bfp` sequences 48

2.14.2 Load-immediate 48

2.14.3 Fused shift sequences 48

2.15 Other micro-architectural considerations 49

2.15.1 Unaligned memory access 49

2.15.2 Fast multiply 50

2.16 C intrinsics via `<rvintrin.h>` 50

3 Reference Implementations **53**

3.1 Verilog reference implementations 53

3.2 Fast C reference implementations 55

3.3 Bit permutation instructions as bit-index operations 57

4 Example Applications **59**

4.1 Basic Bitmanipulation 59

4.1.1 Loading constants . . . 59
4.1.2 Bitfield extract . . . 60
4.1.3 Packing bit-fields . . . 61
4.1.4 Parity check . . . 62
4.1.5 Average of two integers . . . 62
4.1.6 Detecting integer overflow . . . 63
4.1.7 Fuse-able sequences for logic operations . . . 63
4.1.8 Rotate shift of bytes and half-words . . . 64
4.1.9 Rank and select . . . 64
4.1.10 OR/AND/XOR-reduce in byte vectors . . . 65
4.1.11 Counting trailing non-zero bytes . . . 65
4.1.12 Finding bytes of certain values . . . 66
4.1.13 Fill right of most significant set bit . . . 66
4.1.14 Round to next power of two . . . 67
4.2 Packed vectors . . . 68
4.2.1 Packing bytes . . . 68
4.2.2 Permuting bytes . . . 68
4.2.3 Widening and narrowing . . . 68
4.2.4 Shifting packed vector elements . . . 69
4.2.5 Adding packed vectors . . . 70
4.3 Funnel shifts . . . 70
4.3.1 Bigint shift . . . 70
4.3.2 Parsing bit-streams of 27-bit words . . . 71
4.3.3 Parsing bit-streams of 6-bit words . . . 72
4.3.4 Fixed-point multiply . . . 74
4.4 Arbitrary bit permutations . . . 74
4.4.1 Using butterfly operations . . . 74
4.4.2 Using omega-flip networks . . . 75

4.4.3 Using baseline networks 76
4.4.4 Using sheep-and-goats 76
4.4.5 Using bit-matrix multiply 77
4.5 Mirroring and rotating bitboards 77
4.5.1 Mirroring bitboards 77
4.5.2 Rotating bitboards 78
4.5.3 Explanation 79
4.5.4 Rotating Bitcubes 79
4.6 Manipulating 64x64 Bit Matrices 80
4.7 Inverting Xorshift RNGs 81
4.8 Cyclic redundency checks (CRC) 82
4.9 Decoding RISC-V Immediates 85

Change History **87**

Bibliography **89**

Chapter 1

Introduction

This is the RISC-V Bitmanip Extension draft spec.

1.1 ISA Extension Proposal Design Criteria

Any proposed changes to the ISA should be evaluated according to the following criteria.

- Architecture Consistency: Decisions must be consistent with RISC-V philosophy. ISA changes should deviate as little as possible from existing RISC-V standards (such as instruction encodings), and should not re-implement features that are already found in the base specification or other extensions.

- Threshold Metric: The proposal should provide *significant* savings in terms of clocks or instructions. As a heuristic, any proposal should replace at least three instructions. An instruction that only replaces two may be considered, but only if the frequency of use is very high and/or the implementation very cheap.

- Data-Driven Value: Usage in real world applications, and corresponding benchmarks showing a performance increase, will contribute to the score of a proposal. A proposal will not be accepted on the merits of its *theoretical* value alone, unless it is used in the real world.

- Hardware Simplicity: Though instructions saved is the primary benefit, proposals that dramatically increase the hardware complexity and area, or are difficult to implement, should be penalized and given extra scrutiny. The final proposals should only be made if a test implementation can be produced.

- Compiler Support: ISA changes that can be natively detected by the compiler, or are already used as intrinsics, will score higher than instructions which do not fit that criteria.

1.2 B Extension Adoption Strategy

The overall goal of this extension is pervasive adoption by minimizing potential barriers and ensuring the instructions can be mapped to the largest number of ops, either direct or pseudo, that are supported by the most popular processors and compilers. By adding generic instructions and taking advantage of the RISC-V base instructions that already operate on bits, the minimal set of instructions need to be added while at the same time enabling a rich of operations.

The instructions cover the four major categories of bit manipulation: Count, Extract, Insert, Swap. The spec supports RV32, RV64, and RV128. "Clever" obscure and/or overly specific instructions are avoided in favor of more straightforward, fast, generic ones. Coordination with other emerging RISC-V ISA extensions groups is required to ensure our instruction sets are architecturally consistent.

1.3 Next steps

- Assign concrete instruction encodings so that we can start implementing the extension in processor cores and compilers.
- Add support for this extension to processor cores and compilers so we can run quantitative evaluations on the instructions.
- Create assembler snippets for common operations that do not map 1:1 to any instruction in this spec, but can be implemented easily using clever combinations of the instructions. Add support for those snippets to compilers.

Chapter 2

RISC-V Bitmanip Extension

In the proposals provided in this chapter, the C code examples are for illustration purposes only. They are not optimal implementations, but are intended to specify the desired functionality.

The final standard will likely define a range of Z-extensions for different bit manipulation instructions, with the "B" extension itself being a mix of instructions from those Z-extensions. It is unclear as of yet what this will look like exactly, but it will probably look something like table 2.

The main open questions are:

- Which of the "Zb*" extensions should be included in "B"?
- Which "Zbp" pseudo-ops should be included in "Zbb"?

These decisions will be informed in big part by evaluations of the cost and added value for the individual instructions.

For the purpose of tool-chain development "B" is currently everything.

For extensions that only implement certain pseudo-instructions (e.g., "Zbb" provides `rev8`, `orc.b`, and `zext.h`, which are pseudoinstructions for `grevi`, `gorci`, and `pack` (RV32) or `packw` (RV64), respectively), the same binary encoding is used for those instructions as are used on a core with full support for the `grev[i]` instruction.

Like in the base ISA, we add `*W` instruction variants on RV64 with the semantic of the matching RV32 instruction. Those instructions ignore the upper 32 bit of their input and sign-extend their 32 bit output values. The `*W` instruction is omitted when the 64-bit instruction produces the same result as the `*W` instruction would when the 64-bit instruction is fed sign-extended 32 bit values.

Extension	RV32/RV64	RV64 only
Zbb (*)	`clz, ctz, cpop`	`clzw, ctzw, cpopw`
	`min, minu, max, maxu`	
	`sext.b, sext.h, zext.h`	
	`andn, orn, xnor`	
	`rol, ror, rori`	`rolw, rorw, roriw`
	`rev8, orc.b`	
Zbp	`andn, orn, xnor`	
	`pack, packu, packh`	`packw, packuw`
	`rol, ror, rori`	`rolw, rorw, roriw`
	`grev, grevi`	`grevw, greviw`
	`gorc, gorci`	`gorcw, gorciw`
	`shfl, shfli`	`shflw`
	`unshfl, unshfli`	`unshflw`
	`xperm.n, xperm.b, xperm.h`	`xperm.w`
Zbs	`bset, bseti`	
	`bclr, bclri`	
	`binv, binvi`	
	`bext, bexti`	
Zba (*)	`sh1add`	`sh1add.uw`
	`sh2add`	`sh2add.uw`
	`sh3add`	`sh3add.uw`
		`add.uw, slli.uw`
Zbe	`bcompress, bdecompress`	`bcompressw, bdecompressw`
	`pack, packh`	`packw`
Zbf	`bfp`	`bfpw`
	`pack, packh`	`packw`
Zbc (*)	`clmul, clmulh, clmulr`	
Zbm		`bmator, bmatxor, bmatflip`
		`unzip16, unzip8`
		`pack, packu`
Zbr	`crc32.b, crc32c.b`	
	`crc32.h, crc32c.h`	
	`crc32.w, crc32c.w`	
		`crc32.d, crc32c.d`
Zbt	`cmov, cmix`	
	`fsl, fsr, fsri`	`fslw, fsrw, fsriw`
B	All of the above except Zbr and Zbt	
Notes:		
- * means the extensions are expected to be unchanged in the official version.		

Table 2.1: `Zb*` extensions instruction listings

2.1 Basic bit manipulation instructions

2.1.1 Count Leading/Trailing Zeros (clz, ctz)

```
RISC-V Bitmanip ISA
RV32, RV64:
  clz rd, rs
  ctz rd, rs

RV64 only:
  clzw rd, rs
  ctzw rd, rs
```

The `clz` operation counts the number of 0 bits at the MSB end of the argument. That is, the number of 0 bits before the first 1 bit counting from the most significant bit. If the input is 0, the output is XLEN. If the input is -1, the output is 0.

The `ctz` operation counts the number of 0 bits at the LSB end of the argument. If the input is 0, the output is XLEN. If the input is -1, the output is 0.

```
uint_xlen_t clz(uint_xlen_t rs1)
{
    for (int count = 0; count < XLEN; count++)
        if ((rs1 << count) >> (XLEN - 1))
            return count;
    return XLEN;
}

uint_xlen_t ctz(uint_xlen_t rs1)
{
    for (int count = 0; count < XLEN; count++)
        if ((rs1 >> count) & 1)
            return count;
    return XLEN;
}
```

The expression `XLEN-1-clz(x)` evaluates to the index of the most significant set bit, also known as integer base-2 logarithm, or -1 if `x` is zero.

These instructions are commonly used for scanning bitmaps for set bits, for example in `malloc()`, in binary GCD, or in priority queues such as the `sched_find_first_bit()` function used in the Linux kernel real-time scheduler.

Another common applications include normalization in fixed-point code and soft float libraries, null suppression in data compression.

2.1.2 Count Bits Set (cpop)

```
RISC-V Bitmanip ISA
RV32, RV64:
  cpop rd, rs

RV64 only:
  cpopw rd, rs
```

This instruction counts the number of 1 bits in a register. This operations is known as population count, popcount, sideways sum, bit summation, or Hamming weight. [23, 21]

```
uint_xlen_t cpop(uint_xlen_t rs1)
{
    int count = 0;
    for (int index = 0; index < XLEN; index++)
        count += (rs1 >> index) & 1;
    return count;
}
```

2.1.3 Logic-with-negate (andn, orn, xnor)

```
RISC-V Bitmanip ISA
RV32, RV64:
  andn rd, rs1, rs2
  orn  rd, rs1, rs2
  xnor rd, rs1, rs2
```

This instructions implement AND, OR, and XOR with the 2nd arument inverted.

```
uint_xlen_t andn(uint_xlen_t rs1, uint_xlen_t rs2)
{
    return rs1 & ~rs2;
}

uint_xlen_t orn(uint_xlen_t rs1, uint_xlen_t rs2)
{
    return rs1 | ~rs2;
}

uint_xlen_t xnor(uint_xlen_t rs1, uint_xlen_t rs2)
{
    return rs1 ^ ~rs2;
}
```

This can use the existing inverter on rs2 in the ALU that's already there to implement subtract.

Among other things, those instructions allow implementing the "trailing bit manipulation" code patterns in two instructions each. For example, `(x - 1) & ~x` produces a mask from trailing zero bits in `x`.

2.1.4 Pack two words in one register (`pack`, `packu`, `packh`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  pack  rd, rs1, rs2
  packu rd, rs1, rs2
  packh rd, rs1, rs2

RV64 only:
  packw  rd, rs1, rs2
  packuw rd, rs1, rs2
```

The `pack` instruction packs the XLEN/2-bit lower halves of rs1 and rs2 into rd, with rs1 in the lower half and rs2 in the upper half.

```
uint_xlen_t pack(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t lower = (rs1 << XLEN/2) >> XLEN/2;
    uint_xlen_t upper = rs2 << XLEN/2;
    return lower | upper;
}
```

The `packu` instruction packs the upper halves of rs1 and rs2 into rd.

```
uint_xlen_t packu(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t lower = rs1 >> XLEN/2;
    uint_xlen_t upper = (rs2 >> XLEN/2) << XLEN/2;
    return lower | upper;
}
```

And the `packh` instruction packs the LSB bytes of rs1 and rs2 into the 16 LSB bits of rd, zero extending the rest of rd.

```
uint_xlen_t packh(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t lower = rs1 & 255;
    uint_xlen_t upper = (rs2 & 255) << 8;
    return lower | upper;
}
```

Applications include XLEN/2-bit funnel shifts, zero-extend XLEN/2 bit values, duplicate the lower XLEN/2 bits (e.g. for mask creation), loading unsigned 32 constants on RV64, and packing C structs that fit in a register and are therefore passed in a register according to the RISC-V calling convention.

```
; Constructing a 32-bit int from four bytes (RV32)
packh a0, a0, a1
packh a1, a2, a3
pack a0, a0, a1
```

```
; Load 0xffff0000ffff0000 on RV64
lui rd, 0xffff0
pack rd, rd, rd
```

```
; Same as FSLW on RV64
pack rd, rs1, rs3
rol rd, rd, rs2
addiw rd, rd, 0
```

```
; Clear the upper half of rd
pack rd, rd, zero
```

Paired with `shfli/unshfli` and the other bit permutation instructions, pack can interleave arbitrary power-of-two chunks of `rs1` and `rs2`. For example, interleaving the bytes in the lower halves of `rs1` and `rs2`:

```
pack rd, rs1, rs2
zip8 rd, rd
```

2.1.5 Min/max instructions (`min, max, minu, maxu`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  min  rd, rs1, rs2
  max  rd, rs1, rs2
  minu rd, rs1, rs2
  maxu rd, rs1, rs2
```

We define 4 R-type instructions `min, max, minu, maxu` with the following semantics:

```
uint_xlen_t min(uint_xlen_t rs1, uint_xlen_t rs2)
{
    return (int_xlen_t)rs1 < (int_xlen_t)rs2 ? rs1 : rs2;
}

uint_xlen_t max(uint_xlen_t rs1, uint_xlen_t rs2)
{
    return (int_xlen_t)rs1 > (int_xlen_t)rs2 ? rs1 : rs2;
}

uint_xlen_t minu(uint_xlen_t rs1, uint_xlen_t rs2)
{
    return rs1 < rs2 ? rs1 : rs2;
}
```

```
uint_xlen_t maxu(uint_xlen_t rs1, uint_xlen_t rs2)
{
    return rs1 > rs2 ? rs1 : rs2;
}
```

Code that performs saturated arithmetic on a word size < `XLEN` needs to perform min/max operations frequently. A simple way of performing those operations without branching can benefit those programs.

Some applications spend a lot of time on calculating the absolute values of signed integers. One example of that would be SAT solvers, due to the way CNF literals are commonly encoded [11]. With `max` (or `minu`) this is a two-instruction operation:

```
neg a1, a0
max a0, a0, a1
```

2.1.6 Sign-extend instructions (`sext.b, sext.h`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  sext.b rd, rs
  sext.h rd, rs
```

For sign-extending a byte or half-word we define two unary instructions:

```
uint_xlen_t sextb(uint_xlen_t x)
{
    return int_xlen_t(x << (XLEN-8)) >> (XLEN-8);
}

uint_xlen_t sexth(uint_xlen_t x)
{
    return int_xlen_t(x << (XLEN-16)) >> (XLEN-16);
}
```

Additionally, we define pseudo-instructions for zero extending bytes or half-words and for zero- and sign-extending words:

```
RV32:
  zext.b rd, rs   ->   andi rd, rs, 255
  zext.h rd, rs   ->   pack rd, rs, zero

RV64:
  zext.b rd, rs   ->   andi rd, rs, 255
  zext.h rd, rs   ->   packw rd, rs, zero
  zext.w rd, rs   ->   add.uw rd, rs, zero
  sext.w rd, rs   ->   addiw rd, rs, 0
```

Sign extending 8-bit and 16-bit values is needed when calling a function that accepts an 8-bit or 16-bit signed argument, because the RISC-V calling conventions dictates that such an argument must be passed in sign-extended form.

2.1.7 Single-bit instructions (`bset`, `bclr`, `binv`, `bext`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  bset  rd, rs1, rs2
  bclr  rd, rs1, rs2
  binv  rd, rs1, rs2
  bext  rd, rs1, rs2
  bseti rd, rs1, imm
  bclri rd, rs1, imm
  binvi rd, rs1, imm
  bexti rd, rs1, imm
```

We define 4 single-bit instructions `bset` (set), `bclr` (clear), `binv` (invert), and `bext` (extract), and their immediate-variants, with the following semantics:

```
uint_xlen_t bset(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int shamt = rs2 & (XLEN - 1);
    return rs1 | (uint_xlen_t(1) << shamt);
}

uint_xlen_t bclr(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int shamt = rs2 & (XLEN - 1);
    return rs1 & ~(uint_xlen_t(1) << shamt);
}

uint_xlen_t binv(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int shamt = rs2 & (XLEN - 1);
    return rs1 ^ (uint_xlen_t(1) << shamt);
}

uint_xlen_t bext(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int shamt = rs2 & (XLEN - 1);
    return 1 & (rs1 >> shamt);
}
```

2.1.8 Shift Ones (Left/Right) (`slo, sloi, sro, sroi`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  slo  rd, rs1, rs2
  sro  rd, rs1, rs2
  sloi rd, rs1, imm
  sroi rd, rs1, imm

RV64 only:
  slow  rd, rs1, rs2
  srow  rd, rs1, rs2
  sloiw rd, rs1, imm
  sroiw rd, rs1, imm
```

These instructions are similar to shift-logical operations from the base spec, except instead of shifting in zeros, they shift in ones.

```
uint_xlen_t slo(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int shamt = rs2 & (XLEN - 1);
    return ~(~rs1 << shamt);
}

uint_xlen_t sro(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int shamt = rs2 & (XLEN - 1);
    return ~(~rs1 >> shamt);
}
```

ISAs with flag registers often have a "Shift in Carry" or "Rotate through Carry" instruction. Arguably a "Shift Ones" is an equivalent on an ISA like RISC-V that avoids such flag registers.

The main application for the Shift Ones instruction is mask generation.

When implementing this circuit, the only change in the ALU over a standard logical shift is that the value shifted in is not zero, but is a 1-bit register value that has been forwarded from the high bit of the instruction decode. This creates the desired behavior on both logical zero-shifts and logical ones-shifts.

2.2 Bit permutation instructions

The following sections describe 3 types of bit permutation instructions: Rotate shift, generalized reverse, and generalized shuffle.

A bit permutation essentially is applying an invertible function to the bit addresses. (Bit addresses are 5 bit values on RV32 and 6 bit values on RV64.)

Rotate shift by k is simply addition (`rol`) or subtraction (`ror`) modulo XLEN.

$$i'_{\text{rot}} := i \pm k \mod \text{XLEN}$$

Generalized reverse with control argument k is simply XOR-ing the bit addresses with k:

$$i'_{\text{grev}} := i \oplus k$$

And generalized shuffle is performing a bit permutation on the bits of the bit addresses:

$$i'_{\text{shfl}} := \text{perm}_k(i)$$

With the caveat that a single `shfl`/`unshfl` instruction can only perform a certain sub-set of bit address permutations, but a sequence of 4 `shfl`/`unshfl` instructions can perform any of the 120 such permutations on RV32, and a sequence of 5 `shfl`/`unshfl` instructions can perform any of the 720 such permutations on RV64.

Combining those three types of operations makes a vast number of bit permutations accessible within only a few instructions [26] (see Table 2.2).

N	ROT-only	GREV-only	SHFL-only	ROT+GREV	ROT+GREV+SHFL
0	1	1	1	1	1
1	32	32	24	62	85
2	—	—	86	864	3 030
3	—	—	119	4 640	78 659
4	—	—	120	23 312	2 002 167
5	—	—	—	92 192	50 106 844
6	—	—	—	294 992	1 234 579 963
7	—	—	—	703 744	est. 30 000 000 000
8	—	—	—	1 012 856	est. 700 000 000 000
9	—	—	—	1 046 224	est. 15 000 000 000 000
10	—	—	—	1 048 576	··· ··· ··· ··· ···
11	—	—	—	—	··· ··· ··· ··· ···

Table 2.2: Number of permutations reachable with N permutation instructions on RV32. "—" indicates that additional instructions don't increase the space of reachable permutations.

Sequences of `ror`, `grev`, and `[un]shfl` instructions can generate any arbitrary bit permutation. Often in surprising ways. For example, the following sequence swaps the two LSB bits of `a0`:

```
rori a0, a0, 2
unshfli a0, a0, -1
roli a0, a0, 1
shfli a0, a0, -1
```

The mechanics of this sequence is closely related to the fact that `rol(ror(x-2)+1)` is a function that maps 1 to 0 and 0 to 1 and every other number to itself. (With `rol` and `ror` denoting 1-bit rotate left and right shifts respectively.) See Table 2.3 for details.

Instruction	State (XLEN=8)	Bit-Index Op
initial value	7 6 5 4 3 2 1 0	—
`rori a0, a0, 2`	1 0 7 6 5 4 3 2	$i' := i - 2$
`unshfli a0, a0, -1`	1 7 5 3 0 6 4 2	$i' := \mathrm{ror}(i)$
`roli a0, a0, 1`	7 5 3 0 6 4 2 1	$i' := i + 1$
`shfli a0, a0, -1`	7 6 5 4 3 2 0 1	$i' := \mathrm{rol}(i)$

Table 2.3: Breakdown of the `ror+[un]shfl` sequence for swapping the two LSB bits of a word, using XLEN=8 for simplicity.

The numbers in the right column of Table 2.2 might appear large, but they are tiny in comparison to the total number of 32-bit permutations ($2.63 \cdot 10^{35}$). Fortunately the `[un]shfl` instruction "explores" permutations that involve fields that have a size that's a power-of-two, and/or moves that are a power-of-two first, which means we get shorter sequences for permutations we more often care about in real-world applications.

For example, there are 24 ways of arranging the four bytes in a 32-bit word. `ror`, `grev`, and `[un]shfl` can perform any of those permutations in at most 3 instructions. See Table 4.1 for a list of those 24 sequences.

There are 40320 ways of arranging the eight bytes in a 64-bit word or the eight nibbles in a 32-bit word. `ror`, `grev`, and `[un]shfl` can perform any of those permutations in at most 9 instructions. [26]

Besides the more-or-less arbitrary permutations we get from combining `ror`, `grev`, and `[un]shfl` in long sequences, there are of course many cases where just one of these instructions does exactly what we need. For `grev` and `[un]shfl` we define `rev*` and `[un]zip*` pseudo instructions for the most common use-cases.

2.2.1 Rotate (Left/Right) (`rol, ror, rori`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  ror  rd, rs1, rs2
  rol  rd, rs1, rs2
  rori rd, rs1, imm

RV64 only:
  rorw  rd, rs1, rs2
  rolw  rd, rs1, rs2
  roriw rd, rs1, imm
```

These instructions are similar to shift-logical operations from the base spec, except they shift in the values from the opposite side of the register, in order. This is also called 'circular shift'.

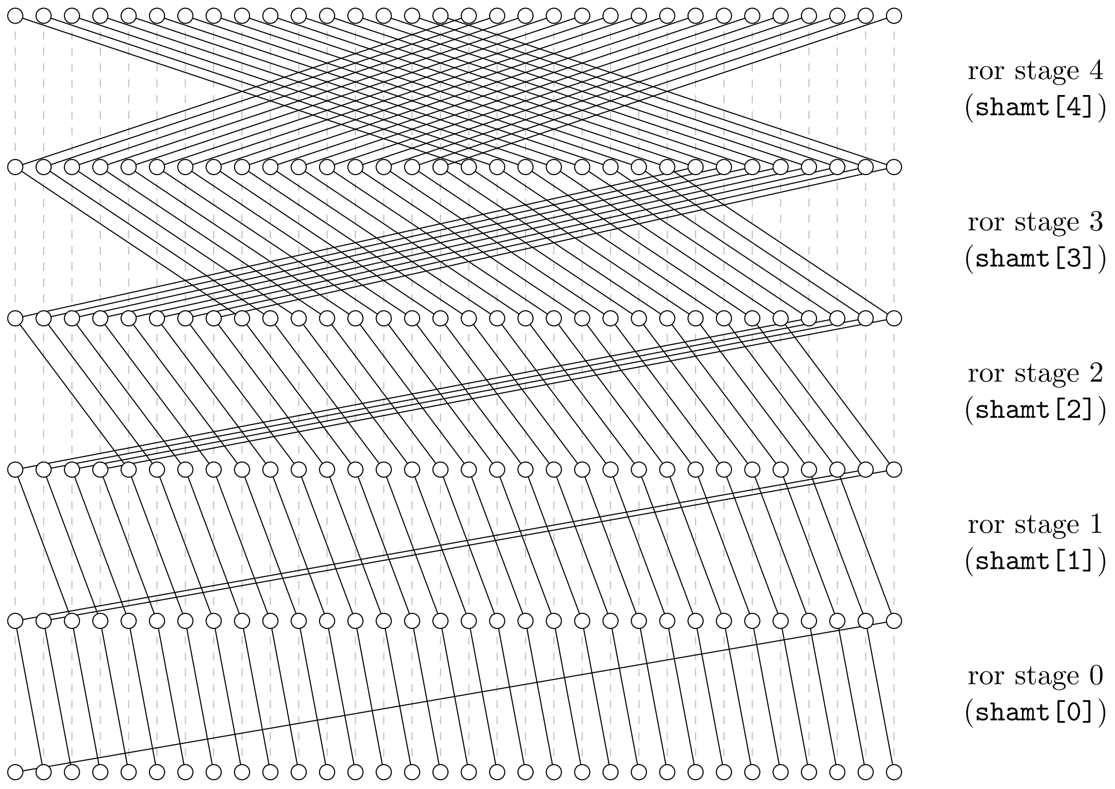


Figure 2.1: `ror` permutation network

```
uint_xlen_t rol(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int shamt = rs2 & (XLEN - 1);
    return (rs1 << shamt) | (rs1 >> ((XLEN - shamt) & (XLEN - 1)));
}
```

```
uint_xlen_t ror(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int shamt = rs2 & (XLEN - 1);
    return (rs1 >> shamt) | (rs1 << ((XLEN - shamt) & (XLEN - 1)));
}
```

2.2.2 Generalized Reverse (grev, grevi, rev)

```
RISC-V Bitmanip ISA
RV32, RV64:
  grev  rd, rs1, rs2
  grevi rd, rs1, imm

RV64 only:
  grevw  rd, rs1, rs2
  greviw rd, rs1, imm
```

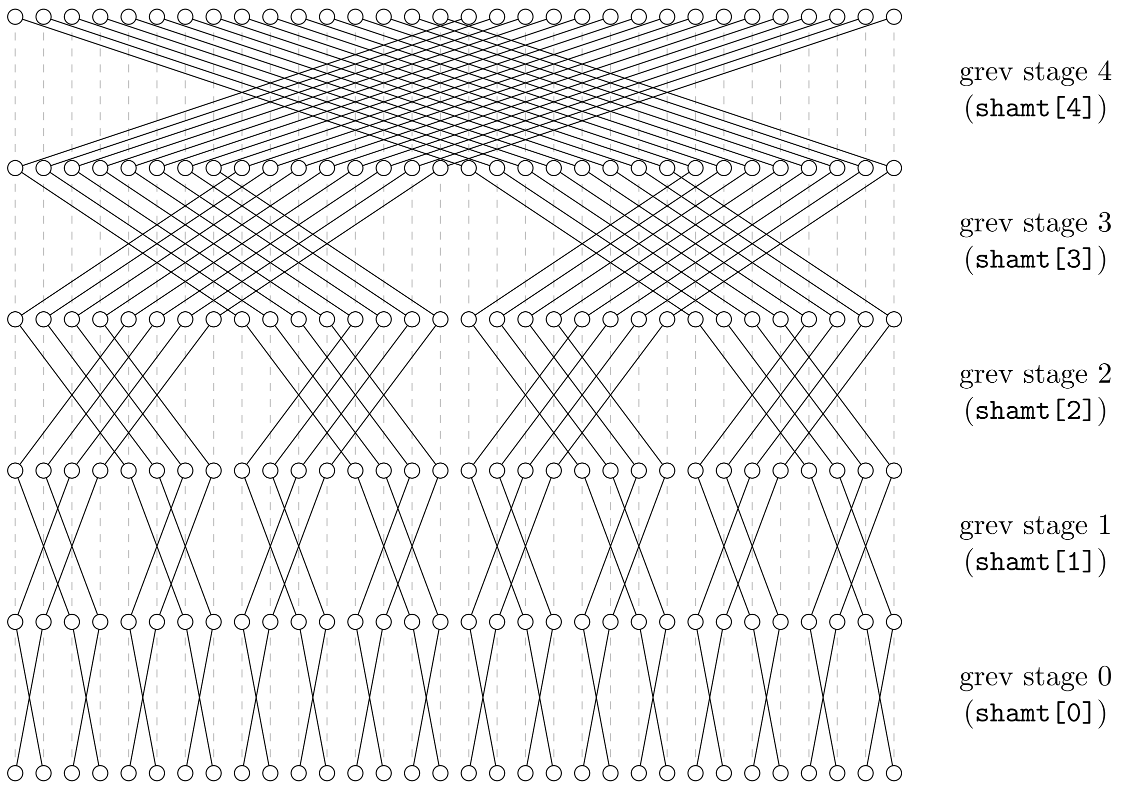


Figure 2.2: `grev` permutation network

This instruction provides a single hardware instruction that can implement all of byte-order swap, bitwise reversal, short-order-swap, word-order-swap (RV64), nibble-order swap, bitwise reversal in a byte, etc, all from a single hardware instruction.

The Generalized Reverse (GREV) operation iteratively checks each bit i in the 2nd argument from $i = 0$ to $log_2(\text{XLEN}) - 1$, and if the corresponding bit is set, swaps each adjacent pair of 2^i bits.

```
uint32_t grev32(uint32_t rs1, uint32_t rs2)
{
    uint32_t x = rs1;
    int shamt = rs2 & 31;
    if (shamt &  1) x = ((x & 0x55555555) <<  1) | ((x & 0xAAAAAAAA) >>  1);
    if (shamt &  2) x = ((x & 0x33333333) <<  2) | ((x & 0xCCCCCCCC) >>  2);
    if (shamt &  4) x = ((x & 0x0F0F0F0F) <<  4) | ((x & 0xF0F0F0F0) >>  4);
    if (shamt &  8) x = ((x & 0x00FF00FF) <<  8) | ((x & 0xFF00FF00) >>  8);
    if (shamt & 16) x = ((x & 0x0000FFFF) << 16) | ((x & 0xFFFF0000) >> 16);
    return x;
}
```

```
uint64_t grev64(uint64_t rs1, uint64_t rs2)
{
    uint64_t x = rs1;
    int shamt = rs2 & 63;
    if (shamt &  1) x = ((x & 0x5555555555555555LL) <<  1) |
                        ((x & 0xAAAAAAAAAAAAAAAALL) >>  1);
    if (shamt &  2) x = ((x & 0x3333333333333333LL) <<  2) |
                        ((x & 0xCCCCCCCCCCCCCCCCLL) >>  2);
    if (shamt &  4) x = ((x & 0x0F0F0F0F0F0F0F0FLL) <<  4) |
                        ((x & 0xF0F0F0F0F0F0F0F0LL) >>  4);
    if (shamt &  8) x = ((x & 0x00FF00FF00FF00FFLL) <<  8) |
                        ((x & 0xFF00FF00FF00FF00LL) >>  8);
    if (shamt & 16) x = ((x & 0x0000FFFF0000FFFFLL) << 16) |
                        ((x & 0xFFFF0000FFFF0000LL) >> 16);
    if (shamt & 32) x = ((x & 0x00000000FFFFFFFFLL) << 32) |
                        ((x & 0xFFFFFFFF00000000LL) >> 32);
    return x;
}
```

The above pattern should be intuitive to understand in order to extend this definition in an obvious manner for RV128.

The `grev` operation can easily be implemented using a permutation network with log_2(XLEN) stages. Figure 2.1 shows the permutation network for `ror` for reference. Figure 2.2 shows the permutation network for `grev`.

Pseudo-instructions are provided for the most common GREVI use-cases. Their names consist of a prefix and and optional suffix. Each prefix and suffix corresponds to a bit mask (see Table 2.5). The GREVI control word is obtained by AND-ing the two masks together.

In other words, the prefix controls the number of zero bits at the LSB end of the control word, and the suffix controls the number of zeros at the MSB end of the control word.

`rev8` reverses the order of bytes in a word, thus performs endianness conversion. This is equivalent to the ARM `REV` instructions or `BSWAP` on x86. ARM also has instructions for swapping the bytes in 16-bit and 32-bit words, and reversing the bit order (see table 2.6).

RV32		RV64			
shamt	Instruction	shamt	Instruction	shamt	Instruction
0: 00000	—	0: 000000	—	32: 100000	`rev32`
1: 00001	`rev.p`	1: 000001	`rev.p`	33: 100001	—
2: 00010	`rev2.n`	2: 000010	`rev2.n`	34: 100010	—
3: 00011	`rev.n`	3: 000011	`rev.n`	35: 100011	—
4: 00100	`rev4.b`	4: 000100	`rev4.b`	36: 100100	—
5: 00101	—	5: 000101	—	37: 100101	—
6: 00110	`rev2.b`	6: 000110	`rev2.b`	38: 100110	—
7: 00111	`rev.b`	7: 000111	`rev.b`	39: 100111	—
8: 01000	`rev8.h`	8: 001000	`rev8.h`	40: 101000	—
9: 01001	—	9: 001001	—	41: 101001	—
10: 01010	—	10: 001010	—	42: 101010	—
11: 01011	—	11: 001011	—	43: 101011	—
12: 01100	`rev4.h`	12: 001100	`rev4.h`	44: 101100	—
13: 01101	—	13: 001101	—	45: 101101	—
14: 01110	`rev2.h`	14: 001110	`rev2.h`	46: 101110	—
15: 01111	`rev.h`	15: 001111	`rev.h`	47: 101111	—
16: 10000	`rev16`	16: 010000	`rev16.w`	48: 110000	`rev16`
17: 10001	—	17: 010001	—	49: 110001	—
18: 10010	—	18: 010010	—	50: 110010	—
19: 10011	—	19: 010011	—	51: 110011	—
20: 10100	—	20: 010100	—	52: 110100	—
21: 10101	—	21: 010101	—	53: 110101	—
22: 10110	—	22: 010110	—	54: 110110	—
23: 10111	—	23: 010111	—	55: 110111	—
24: 11000	`rev8`	24: 011000	`rev8.w`	56: 111000	`rev8`
25: 11001	—	25: 011001	—	57: 111001	—
26: 11010	—	26: 011010	—	58: 111010	—
27: 11011	—	27: 011011	—	59: 111011	—
28: 11100	`rev4`	28: 011100	`rev4.w`	60: 111100	`rev4`
29: 11101	—	29: 011101	—	61: 111101	—
30: 11110	`rev2`	30: 011110	`rev2.w`	62: 111110	`rev2`
31: 11111	`rev`	31: 011111	`rev.w`	63: 111111	`rev`

Table 2.4: Pseudo-instructions for `grevi` instruction

Prefix	Mask	Suffix	Mask	
`rev`	111111	—	111111	
`rev2`	111110	`.w`	011111	(`w` = word)
`rev4`	111100	`.h`	001111	(`h` = half word)
`rev8`	111000	`.b`	000111	(`b` = byte)
`rev16`	110000	`.n`	000011	(`n` = nibble)
`rev32`	100000	`.p`	000001	(`p` = pair)

Table 2.5: Naming scheme for `grevi` pseudo-instructions. The prefix and suffix masks are ANDed to compute the immediate argument.

RISC-V	ARM	X86
`rev`	RBIT	—
`rev8.h`	REV16	—
`rev8.w`	REV32	—
`rev8`	REV	BSWAP

Table 2.6: Comparison of bit/byte reversal instructions

2.2.3 Generalized Shuffle (`shfl`, `unshfl`, `shfli`, `unshfli`, `zip`, `unzip`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  shfl    rd, rs1, rs2
  unshfl  rd, rs1, rs2
  shfli   rd, rs1, imm
  unshfli rd, rs1, imm

RV64 only:
  shflw    rd, rs1, rs2
  unshflw  rd, rs1, rs2
```

Shuffle is the third bit permutation instruction in the RISC-V Bitmanip extension, after rotary shift and generalized reverse. It implements a generalization of the operation commonly known as perfect outer shuffle and its inverse (shuffle/unshuffle), also known as zip/unzip or interlace/uninterlace.

Bit permutations can be understood as reversible functions on bit indices (i.e. 5 bit functions on RV32 and 6 bit functions on RV64).

Operation	Corresponding function on bit indices
Rotate shift	Addition modulo XLEN
Generalized reverse	XOR with bitmask
Generalized shuffle	Bitpermutation

A generalized (un)shuffle operation has $log_2(\text{XLEN}) - 1$ control bits, one for each pair of neighbouring bits in a bit index. When the bit is set, generalized shuffle will swap the two index bits. The `shfl` operation performs this swaps in MSB-to-LSB order (performing a rotate left shift on contiguous regions of set control bits), and the `unshfl` operation performs the swaps in LSB-to-MSB order (performing a rotate right shift on contiguous regions of set control bits). Combining up to $log_2(\text{XLEN})$ of those `shfl`/`unshfl` operations can implement any bitpermutation on the bit indices.

The most common type of shuffle/unshuffle operation is one on an immediate control value that only contains one contiguous region of set bits. We call those operations zip/unzip and provide pseudo-instructions for them. The naming scheme for those pseudo-instructions is similar to the naming scheme for the `grevi` pseudo-instructions (see Tables 2.4 and 2.5), except that the LSB bit of the masks in Table 2.5 is not used for zip/unzip.

Shuffle/unshuffle operations that only have individual bits set (not a contiguous region of two or more bits) are their own inverse.

shamt	inv	Bit index rotations	Pseudo-Instruction
0: 0000	0	no-op	—
0000	1	no-op	—
1: 0001	0	`i[1] -> i[0]`	`zip.n, unzip.n`
0001	1	*equivalent to 0001 0*	—
2: 0010	0	`i[2] -> i[1]`	`zip2.b, unzip2.b`
0010	1	*equivalent to 0010 0*	—
3: 0011	0	`i[2] -> i[0]`	`zip.b`
0011	1	`i[2] <- i[0]`	`unzip.b`
4: 0100	0	`i[3] -> i[2]`	`zip4.h, unzip4.h`
0100	1	*equivalent to 0100 0*	—
5: 0101	0	`i[3] -> i[2], i[1] -> i[0]`	—
0101	1	*equivalent to 0101 0*	—
6: 0110	0	`i[3] -> i[1]`	`zip2.h`
0110	1	`i[3] <- i[1]`	`unzip2.h`
7: 0111	0	`i[3] -> i[0]`	`zip.h`
0111	1	`i[3] <- i[0]`	`unzip.h`
8: 1000	0	`i[4] -> i[3]`	`zip8, unzip8`
1000	1	*equivalent to 1000 0*	—
9: 1001	0	`i[4] -> i[3], i[1] -> i[0]`	—
1001	1	*equivalent to 1001 0*	—
10: 1010	0	`i[4] -> i[3], i[2] -> i[1]`	—
1010	1	*equivalent to 1010 0*	—
11: 1011	0	`i[4] -> i[3], i[2] -> i[0]`	—
1011	1	`i[4] <- i[3], i[2] <- i[0]`	—
12: 1100	0	`i[4] -> i[2]`	`zip4`
1100	1	`i[4] <- i[2]`	`unzip4`
13: 1101	0	`i[4] -> i[2], i[1] -> i[0]`	—
1101	1	`i[4] <- i[2], i[1] <- i[0]`	—
14: 1110	0	`i[4] -> i[1]`	`zip2`
1110	1	`i[4] <- i[1]`	`unzip2`
15: 1111	0	`i[4] -> i[0]`	`zip`
1111	1	`i[4] <- i[0]`	`unzip`

Table 2.7: RV32 modes and pseudo-instructions for `shfli`/`unshfli` instruction

Like GREV and rotate shift, the (un)shuffle instruction can be implemented using a short sequence of elementary permutations, that are enabled or disabled by the shamt bits. But (un)shuffle has one stage fewer than GREV. Thus shfli+unshfli together require the same amount of encoding space as grevi.

```
uint32_t shuffle32_stage(uint32_t src, uint32_t maskL, uint32_t maskR, int N)
{
    uint32_t x = src & ~(maskL | maskR);
    x |= ((src <<  N) & maskL) | ((src >>  N) & maskR);
    return x;
}
```

shamt	inv	Pseudo-Instruction
0: 00000	0	—
00000	1	—
1: 00001	0	`zip.n, unzip.n`
00001	1	—
2: 00010	0	`zip2.b, unzip2.b`
00010	1	—
3: 00011	0	`zip.b`
00011	1	`unzip.b`
4: 00100	0	`zip4.h, unzip4.h`
00100	1	—
5: 00101	0	—
00101	1	—
6: 00110	0	`zip2.h`
00110	1	`unzip2.h`
7: 00111	0	`zip.h`
00111	1	`unzip.h`
8: 01000	0	`zip8.w, unzip8.w`
01000	1	—
9: 01001	0	—
01001	1	—
10: 01010	0	—
01010	1	—
11: 01011	0	—
01011	1	—
12: 01100	0	`zip4.w`
01100	1	`unzip4.w`
13: 01101	0	—
01101	1	—
14: 01110	0	`zip2.w`
01110	1	`unzip2.w`
15: 01111	0	`zip.w`
01111	1	`unzip.w`
16: 10000	0	`zip16, unzip16`
10000	1	—
17: 10001	0	—
10001	1	—
18: 10010	0	—
10010	1	—
19: 10011	0	—
10011	1	—
20: 10100	0	—
10100	1	—
21: 10101	0	—
10101	1	—
22: 10110	0	—
10110	1	—
23: 10111	0	—
10111	1	—
24: 11000	0	`zip8`
11000	1	`unzip8`
25: 11001	0	—
11001	1	—
26: 11010	0	—
11010	1	—
27: 11011	0	—
11011	1	—
28: 11100	0	`zip4`
11100	1	`unzip4`
29: 11101	0	—
11101	1	—
30: 11110	0	`zip2`
11110	1	`unzip2`
31: 11111	0	`zip`
11111	1	`unzip`

Table 2.8: RV64 modes and pseudo-instructions for `shfli`/`unshfli` instruction

```
uint32_t shfl32(uint32_t rs1, uint32_t rs2)
{
    uint32_t x = rs1;
    int shamt = rs2 & 15;

    if (shamt & 8) x = shuffle32_stage(x, 0x00ff0000, 0x0000ff00, 8);
    if (shamt & 4) x = shuffle32_stage(x, 0x0f000f00, 0x00f000f0, 4);
    if (shamt & 2) x = shuffle32_stage(x, 0x30303030, 0x0c0c0c0c, 2);
    if (shamt & 1) x = shuffle32_stage(x, 0x44444444, 0x22222222, 1);

    return x;
}
```

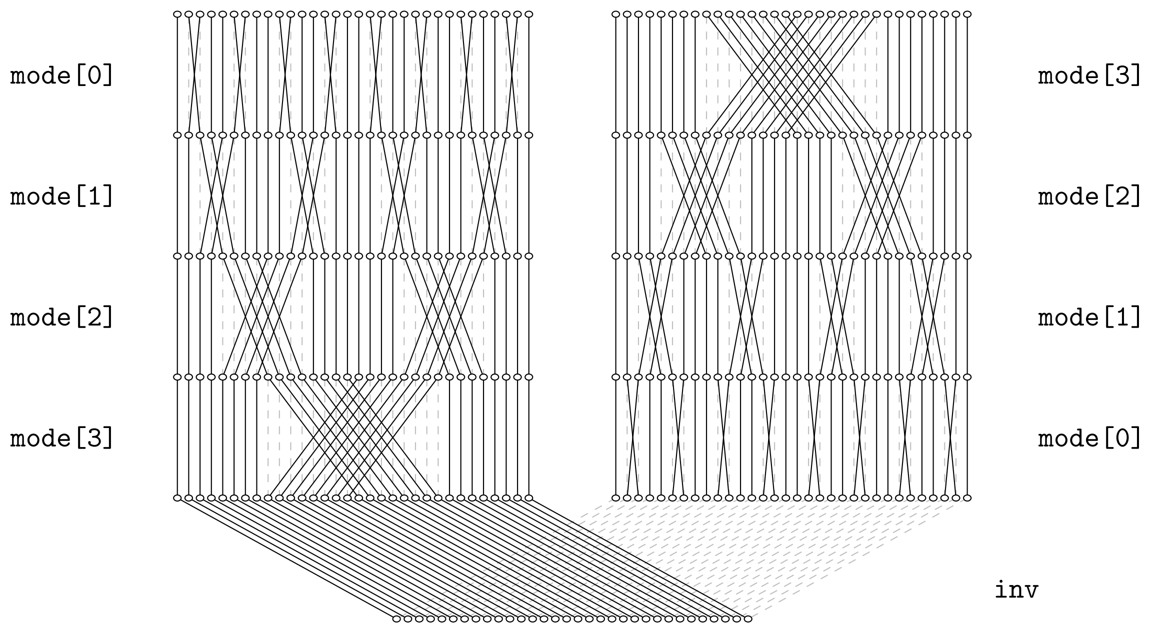


Figure 2.3: (un)shuffle permutation network without “flip” stages

```
uint32_t unshfl32(uint32_t rs1, uint32_t rs2)
{
    uint32_t x = rs1;
    int shamt = rs2 & 15;

    if (shamt & 1) x = shuffle32_stage(x, 0x44444444, 0x22222222, 1);
    if (shamt & 2) x = shuffle32_stage(x, 0x30303030, 0x0c0c0c0c, 2);
    if (shamt & 4) x = shuffle32_stage(x, 0x0f000f00, 0x00f000f0, 4);
    if (shamt & 8) x = shuffle32_stage(x, 0x00ff0000, 0x0000ff00, 8);

    return x;
}
```

Or for RV64:

```
uint64_t shuffle64_stage(uint64_t src, uint64_t maskL, uint64_t maskR, int N)
{
    uint64_t x = src & ~(maskL | maskR);
    x |= ((src <<  N) & maskL) | ((src >>  N) & maskR);
    return x;
}
```

```
uint64_t shfl64(uint64_t rs1, uint64_t rs2)
{
    uint64_t x = rs1;
    int shamt = rs2 & 31;

    if (shamt & 16) x = shuffle64_stage(x, 0x0000ffff00000000LL,
                                           0x00000000ffff0000LL, 16);
    if (shamt &  8) x = shuffle64_stage(x, 0x00ff000000ff0000LL,
                                           0x0000ff000000ff00LL, 8);
    if (shamt &  4) x = shuffle64_stage(x, 0x0f000f000f000f00LL,
                                           0x00f000f000f000f0LL, 4);
    if (shamt &  2) x = shuffle64_stage(x, 0x3030303030303030LL,
                                           0x0c0c0c0c0c0c0c0cLL, 2);
    if (shamt &  1) x = shuffle64_stage(x, 0x4444444444444444LL,
                                           0x2222222222222222LL, 1);

    return x;
}

uint64_t unshfl64(uint64_t rs1, uint64_t rs2)
{
    uint64_t x = rs1;
    int shamt = rs2 & 31;

    if (shamt &  1) x = shuffle64_stage(x, 0x4444444444444444LL,
                                           0x2222222222222222LL, 1);
    if (shamt &  2) x = shuffle64_stage(x, 0x3030303030303030LL,
                                           0x0c0c0c0c0c0c0c0cLL, 2);
    if (shamt &  4) x = shuffle64_stage(x, 0x0f000f000f000f00LL,
                                           0x00f000f000f000f0LL, 4);
    if (shamt &  8) x = shuffle64_stage(x, 0x00ff000000ff0000LL,
                                           0x0000ff000000ff00LL, 8);
    if (shamt & 16) x = shuffle64_stage(x, 0x0000ffff00000000LL,
                                           0x00000000ffff0000LL, 16);

    return x;
}
```

The above pattern should be intuitive to understand in order to extend this definition in an obvious manner for RV128.

Alternatively (un)shuffle can be implemented in a single network with one more stage than GREV, with the additional first and last stage executing a permutation that effectively reverses the order of the inner stages. However, since the inner stages only mux half of the bits in the word each, a hardware implementation using this additional “flip” stages might actually be more expensive than simply creating two networks.

```
uint32_t shuffle32_flip(uint32_t src)
{
    uint32_t x = src & 0x88224411;
    x |= ((src <<  6) & 0x22001100) | ((src >>  6) & 0x00880044);
    x |= ((src <<  9) & 0x00440000) | ((src >>  9) & 0x00002200);
    x |= ((src << 15) & 0x44110000) | ((src >> 15) & 0x00008822);
    x |= ((src << 21) & 0x11000000) | ((src >> 21) & 0x00000088);
    return x;
}

uint32_t unshfl32alt(uint32_t rs1, uint32_t rs2)
{
    uint32_t shfl_mode = 0;
    if (rs2 & 1) shfl_mode |= 8;
    if (rs2 & 2) shfl_mode |= 4;
    if (rs2 & 4) shfl_mode |= 2;
    if (rs2 & 8) shfl_mode |= 1;

    uint32_t x = rs1;
    x = shuffle32_flip(x);
    x = shfl32(x, shfl_mode);
    x = shuffle32_flip(x);

    return x;
}
```

Figure 2.4 shows the (un)shuffle permutation network with “flip” stages and Figure 2.3 shows the (un)shuffle permutation network without “flip” stages.

The `zip` instruction with the upper half of its input cleared performs the commonly needed “fan-out” operation. (Equivalent to `bdecompress` with a 0x55555555 mask.) The `zip` instruction applied twice fans out the bits in the lower quarter of the input word by a spacing of 4 bits.

For example, the following code calculates the bitwise prefix sum of the bits in the lower byte of a 32 bit word on RV32:

```
andi a0, a0, 0xff
zip a0, a0
zip a0, a0
slli a1, a0, 4
c.add a0, a1
slli a1, a0, 8
c.add a0, a1
slli a1, a0, 16
c.add a0, a1
```

The final prefix sum is stored in the 8 nibbles of the `a0` output word.

Similarly, the following code stores the indices of the set bits in the LSB nibbles of the output word (with the LSB bit having index 1), with the unused MSB nibbles in the output set to zero:

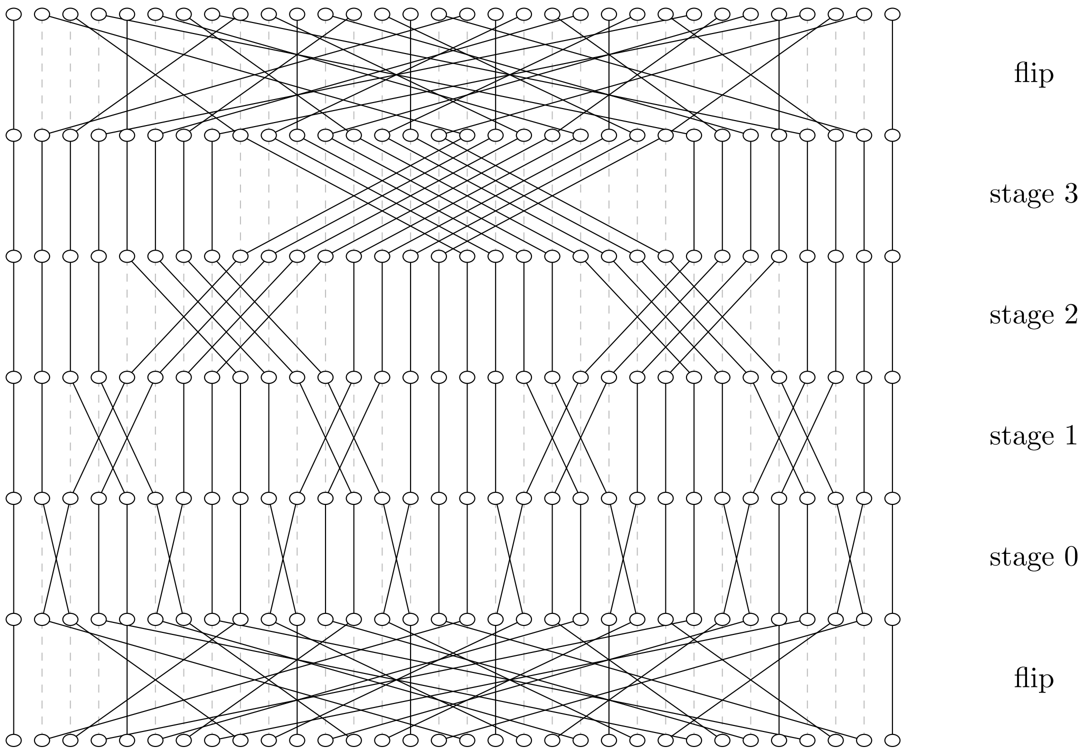


Figure 2.4: (un)shuffle permutation network with “flip” stages

```
andi a0, a0, 0xff
zip a0, a0
zip a0, a0
orc.n a0, a0
li a1, 0x87654321
and a1, a0, a1
bcompress a0, a1, a0
```

Other `zip` modes can be used to “fan-out” in blocks of 2, 4, 8, or 16 bit. `zip` can be combined with `grevi` to perform inner shuffles. For example on RV64:

```
li a0, 0x0000000012345678
zip4 t0, a0    ; <- 0x0102030405060708
rev4.b t1, t0  ; <- 0x1020304050607080
zip8 t2, a0    ; <- 0x0012003400560078
rev8.h t3, t2  ; <- 0x1200340056007800
zip16 t4, a0   ; <- 0x0000123400005678
rev16.w t5, t4 ; <- 0x1234000056780000
```

Another application for the zip instruction is generating Morton code [24].

The x86 `PUNPCK[LH]*` MMX/SSE/AVX instructions perform similar operations as `zip8` and `zip16`.

2.2.4 Crossbar Permutation Instructions (`xperm.[nbhw]`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  xperm.n rd, rs1, rs2
  xperm.b rd, rs1, rs2
  xperm.h rd, rs1, rs2

RV64 only:
  xperm.w rd, rs1, rs2
```

These instructions operate on nibbles/bytes/half-words/words. `rs1` is a vector of data words and `rs2` is a vector of indices into `rs1`. The result of the instruction is the vector `rs2` with each element replaced by the corresponding data word from `rs1`, or zero then the index in `rs2` is out of bounds.

```
uint_xlen_t xperm(uint_xlen_t rs1, uint_xlen_t rs2, int sz_log2)
{
    uint_xlen_t r = 0;
    uint_xlen_t sz = 1LL << sz_log2;
    uint_xlen_t mask = (1LL << sz) - 1;
    for (int i = 0; i < XLEN; i += sz) {
        uint_xlen_t pos = ((rs2 >> i) & mask) << sz_log2;
        if (pos < XLEN)
            r |= ((rs1 >> pos) & mask) << i;
    }
    return r;
}
```

```
uint_xlen_t xperm_n (uint_xlen_t rs1, uint_xlen_t rs2) { return xperm(rs1, rs2, 2); }
uint_xlen_t xperm_b (uint_xlen_t rs1, uint_xlen_t rs2) { return xperm(rs1, rs2, 3); }
uint_xlen_t xperm_h (uint_xlen_t rs1, uint_xlen_t rs2) { return xperm(rs1, rs2, 4); }
uint_xlen_t xperm_w (uint_xlen_t rs1, uint_xlen_t rs2) { return xperm(rs1, rs2, 5); }
```

The `xperm.[nbhw]` instructions can be implemented with an $XLEN/4$-lane nibble-wide crossbar switch.

The `xperm.n` instruction can be used to implement an arbitrary 64-bit bit permutation in 15 instructions, using 8 control words and 3 constant masks [25]:

```
uint64_t perm64_bitmanip_cmix(perm64t &p, uint64_t v)
{
    uint64_t v0 = _rv64_gorc(_rv64_xperm_n(v, p.ctrl[0]) & p.mask[0], 3);   //  3 insns
    uint64_t v1 = _rv64_gorc(_rv64_xperm_n(v, p.ctrl[1]) & p.mask[1], 3);   //  6 insns
    uint64_t v2 = _rv64_gorc(_rv64_xperm_n(v, p.ctrl[2]) & p.mask[2], 3);   //  9 insns
    uint64_t v3 = _rv64_gorc(_rv64_xperm_n(v, p.ctrl[3]) & p.mask[3], 3);   // 12 insns

    v0 = _rv_cmix(0x1111111111111111LL, v0, v1);                             // 13 insns
    v2 = _rv_cmix(0x4444444444444444LL, v2, v3);                             // 14 insns
    return _rv_cmix(0x3333333333333333LL, v0, v2);                           // 15 insns
}
```

RV32		RV64			
shamt	Instruction	shamt	Instruction	shamt	Instruction
0: 00000	—	0: 000000	—	32: 100000	orc32
1: 00001	orc.p	1: 000001	orc.p	33: 100001	—
2: 00010	orc2.n	2: 000010	orc2.n	34: 100010	—
3: 00011	orc.n	3: 000011	orc.n	35: 100011	—
4: 00100	orc4.b	4: 000100	orc4.b	36: 100100	—
5: 00101	—	5: 000101	—	37: 100101	—
6: 00110	orc2.b	6: 000110	orc2.b	38: 100110	—
7: 00111	orc.b	7: 000111	orc.b	39: 100111	—
8: 01000	orc8.h	8: 001000	orc8.h	40: 101000	—
9: 01001	—	9: 001001	—	41: 101001	—
10: 01010	—	10: 001010	—	42: 101010	—
11: 01011	—	11: 001011	—	43: 101011	—
12: 01100	orc4.h	12: 001100	orc4.h	44: 101100	—
13: 01101	—	13: 001101	—	45: 101101	—
14: 01110	orc2.h	14: 001110	orc2.h	46: 101110	—
15: 01111	orc.h	15: 001111	orc.h	47: 101111	—
16: 10000	orc16	16: 010000	orc16.w	48: 110000	orc16
17: 10001	—	17: 010001	—	49: 110001	—
18: 10010	—	18: 010010	—	50: 110010	—
19: 10011	—	19: 010011	—	51: 110011	—
20: 10100	—	20: 010100	—	52: 110100	—
21: 10101	—	21: 010101	—	53: 110101	—
22: 10110	—	22: 010110	—	54: 110110	—
23: 10111	—	23: 010111	—	55: 110111	—
24: 11000	orc8	24: 011000	orc8.w	56: 111000	orc8
25: 11001	—	25: 011001	—	57: 111001	—
26: 11010	—	26: 011010	—	58: 111010	—
27: 11011	—	27: 011011	—	59: 111011	—
28: 11100	orc4	28: 011100	orc4.w	60: 111100	orc4
29: 11101	—	29: 011101	—	61: 111101	—
30: 11110	orc2	30: 011110	orc2.w	62: 111110	orc2
31: 11111	orc	31: 011111	orc.w	63: 111111	orc

Table 2.9: Pseudo-instructions for `gorci` instruction

2.3 Generalized OR-Combine (gorc, gorci)

```
RISC-V Bitmanip ISA
RV32, RV64:
  gorc  rd, rs1, rs2
  gorci rd, rs1, imm

RV64 only:
  gorcw  rd, rs1, rs2
  gorciw rd, rs1, imm
```

The GORC operation is similar to GREV, except that instead of swapping pairs of bits, GORC ORs them together, and writes the new value in both positions.

```
uint32_t gorc32(uint32_t rs1, uint32_t rs2)
{
    uint32_t x = rs1;
    int shamt = rs2 & 31;
    if (shamt &  1) x |= ((x & 0x55555555) <<  1) | ((x & 0xAAAAAAAA) >>  1);
    if (shamt &  2) x |= ((x & 0x33333333) <<  2) | ((x & 0xCCCCCCCC) >>  2);
    if (shamt &  4) x |= ((x & 0x0F0F0F0F) <<  4) | ((x & 0xF0F0F0F0) >>  4);
    if (shamt &  8) x |= ((x & 0x00FF00FF) <<  8) | ((x & 0xFF00FF00) >>  8);
    if (shamt & 16) x |= ((x & 0x0000FFFF) << 16) | ((x & 0xFFFF0000) >> 16);
    return x;
}

uint64_t gorc64(uint64_t rs1, uint64_t rs2)
{
    uint64_t x = rs1;
    int shamt = rs2 & 63;
    if (shamt &  1) x |= ((x & 0x5555555555555555LL) <<  1) |
                         ((x & 0xAAAAAAAAAAAAAAAALL) >>  1);
    if (shamt &  2) x |= ((x & 0x3333333333333333LL) <<  2) |
                         ((x & 0xCCCCCCCCCCCCCCCCLL) >>  2);
    if (shamt &  4) x |= ((x & 0x0F0F0F0F0F0F0F0FLL) <<  4) |
                         ((x & 0xF0F0F0F0F0F0F0F0LL) >>  4);
    if (shamt &  8) x |= ((x & 0x00FF00FF00FF00FFLL) <<  8) |
                         ((x & 0xFF00FF00FF00FF00LL) >>  8);
    if (shamt & 16) x |= ((x & 0x0000FFFF0000FFFFLL) << 16) |
                         ((x & 0xFFFF0000FFFF0000LL) >> 16);
    if (shamt & 32) x |= ((x & 0x00000000FFFFFFFFLL) << 32) |
                         ((x & 0xFFFFFFFF00000000LL) >> 32);
    return x;
}
```

GORC can be useful for copying naturally-aligned fields in a word, and testing such fields for being equal to zero.

`gorci` pseudo-instructions follow the same naming scheme as `grevi` pseudo-instructions (see Tables 2.4 and 2.5), except the prefix `orc` is used instead of `rev`. See Table 2.9 for a full list of `gorci` pseudo-instructions.

An important use-case is `strlen()` and `strcpy()`, which can utilize `orc.b` for testing for zero bytes, and counting trailing non-zero bytes in a word.

2.4 Bit-Field Place (`bfp`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  bfp rd, rs1, rs2

RV64 only:
  bfpw rd, rs1, rs2
```

The bit field place (`bfp`) instruction places up to XLEN/2 LSB bits from `rs2` into the value in `rs1`. The upper bits of `rs2` control the length of the bit field and target position. The layout of `rs2` is chosen in a way that makes it possible to construct `rs2` easily using `pack[h]` instructions and/or `andi`/`lui`.

```
uint_xlen_t bfp(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t cfg = rs2 >> (XLEN/2);
    if ((cfg >> 30) == 2)
        cfg = cfg >> 16;
    int len = (cfg >> 8) & (XLEN/2-1);
    int off = cfg & (XLEN-1);
    len = len ? len : XLEN/2;
    uint_xlen_t mask = slo(0, len) << off;
    uint_xlen_t data = rs2 << off;
    return (data & mask) | (rs1 & ~mask);
}
```

The layout of the control word in rs2 is as follows for RV32. `LEN=0` encodes for `LEN=16`.

```
|  3                   2                   1                   |
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|
|---------------|---------------|---------------|---------------|
|       |  LEN  |     |   OFF   |             DATA              |
|---------------|---------------|---------------|---------------|
```

And on RV64 (with `LEN=0` encoding for `LEN=32`):

```
|      6                   5                   4                   3           |
|3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 .... 2 1 0|
|---------------|---------------|---------------|---------------|------ -- ------|
|SEL| |   LEN   |   |    OFF    |     |  LEN'   |   |   OFF'    |      DATA      |
|---------------|---------------|---------------|---------------|------ -- ------|
```

When `SEL=10` then `LEN` and `OFF` are used, otherwise `LEN'` and `OFF'` are used.

Placing bits from `a0` in `a1`, with results in `t0` on RV32:

```
addi t0, zero, {length[3:0], offset[7:0]}
pack t0, a0, t0
bfp t0, a1, t0
```

And on RV64:

```
lui t0, zero, {3'b 100, length[4:0], offset[7:0], 4'b 0000}
pack t0, a0, t0
bfp t0, a1, t0
```

Or with `a2=length` and `a3=offset`:

```
packh t0, a3, a2
pack t0, a0, t0
bfp t0, a1, t0
```

Placing up to 16 constant bits in any contiguous region:

```
lui t0, ...
addi t0, t0, ...
bfp[w] t0, a1, t0
```

Note that above sequences only modify one register (`t0`), which makes them fuse-able sequences.

2.5 Bit Compress/Decompress (`bcompress, bdecompress`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  bcompress rd, rs1, rs2
  bdecompress rd, rs1, rs2

RV64 only:
  bcompressw rd, rs1, rs2
  bdecompressw rd, rs1, rs2
```

This instructions implement the generic bit extract and bit deposit functions. This operation is also referred to as bit gather/scatter, bit pack/unpack, parallel extract/deposit, compress/expand, or right_compress/right_expand.

`bcompress` collects LSB justified bits to rd from rs1 using extract mask in rs2.

`bdecompress` writes LSB justified bits from rs1 to rd using deposit mask in rs2.

```
uint_xlen_t bcompress(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t r = 0;
    for (int i = 0, j = 0; i < XLEN; i++)
        if ((rs2 >> i) & 1) {
            if ((rs1 >> i) & 1)
                r |= uint_xlen_t(1) << j;
            j++;
        }
    return r;
}

uint_xlen_t bdecompress(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t r = 0;
    for (int i = 0, j = 0; i < XLEN; i++)
        if ((rs2 >> i) & 1) {
            if ((rs1 >> j) & 1)
                r |= uint_xlen_t(1) << i;
            j++;
        }
    return r;
}
```

Implementations may choose to use smaller multi-cycle implementations of `bcompress` and `bdecompress`, or even emulate the instructions in software.

Even though multi-cycle `bcompress` and `bdecompress` often are not fast enough to outperform algorithms that use sequences of shifts and bit masks, dedicated instructions for those operations can still be of great advantage in cases where the mask argument is not constant.

For example, the following code efficiently calculates the index of the tenth set bit in `a0` using `bdecompress`:

```
li a1, 0x00000200
bdecompress a0, a1, a0
ctz a0, a0
```

For cases with a constant mask an optimizing compiler would decide when to use `bcompress` or `bdecompress` based on the optimization profile for the concrete processor it is optimizing for. This is similar to the decision whether to use MUL or DIV with a constant, or to perform the same operation using a longer sequence of much simpler operations.

The `bcompress` and `bdecompress` instructions are equivalent to the x86 BMI2 instructions PEXT and PDEP. But there is much older prior art. For example, the soviet BESM-6 mainframe computer, designed and built in the 1960s, had APX/AUX instructions with almost the same semantics. [1] (The BESM-6 APX/AUX instructions packed/unpacked at the MSB end instead of the LSB end. Otherwise it is the same instruction.)

Efficient hardware implementations of `bcompress` and `bdecompress` are described in [14] and demonstrated in [27].

2.6 Carry-Less Multiply (`clmul`, `clmulh`, `clmulr`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  clmul  rd, rs1, rs2
  clmulh rd, rs1, rs2
  clmulr rd, rs1, rs2
```

Calculate the carry-less product [22] of the two arguments. `clmul` produces the lower half of the carry-less product and `clmulh` produces the upper half of the 2·XLEN carry-less product.

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

`clmulr` produces bits 2·XLEN−2:XLEN-1 of the 2·XLEN carry-less product. That means `clmulh` is equivalent to `clmulr` followed by a 1-bit right shift. (The MSB of a `clmulh` result is always zero.) Another equivalent definition of `clmulr` is `clmulr(a,b) := rev(clmul(rev(a), rev(b)))`. (The "r" in `clmulr` means reversed.)

Unlike `mulh[[s]u]`, we add a *W variant of `clmulh`. This is because we expect some code to use 32-bit clmul intrisics, even on 64-bit architectures. For example in cases where data is processed in 32-bit chunks.

```
uint_xlen_t clmul(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t x = 0;
    for (int i = 0; i < XLEN; i++)
        if ((rs2 >> i) & 1)
            x ^= rs1 << i;
    return x;
}

uint_xlen_t clmulh(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t x = 0;
    for (int i = 1; i < XLEN; i++)
        if ((rs2 >> i) & 1)
            x ^= rs1 >> (XLEN-i);
    return x;
}
```

```
uint_xlen_t clmulr(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t x = 0;
    for (int i = 0; i < XLEN; i++)
        if ((rs2 >> i) & 1)
            x ^= rs1 >> (XLEN-i-1);
    return x;
}
```

The classic applications for `clmul` are Cyclic Redundancy Check (CRC) [12, 28] and Galois/Counter Mode (GCM), but more applications exist, including the following examples.

There are obvious applications in hashing and pseudo random number generations. For example, it has been reported that hashes based on carry-less multiplications can outperform Google's CityHash [18].

`clmul` of a number with itself inserts zeroes between each input bit. This can be useful for generating Morton code [24].

`clmul` of a number with -1 calculates the prefix XOR operation. This can be useful for decoding gray codes.

Another application of XOR prefix sums calculated with `clmul` is branchless tracking of quoted strings in high-performance parsers. [17]

Carry-less multiply can also be used to implement Erasure code efficiently. [15]

SPARC introduced similar instructions (XMULX, XMULXHI) in SPARC T3 in 2010. [7]

TI C6000 introduced a similar instruction (XORMPY) in C64x+. [8]

2.7 CRC Instructions (`crc32.[bhwd]`, `crc32c.[bhwd]`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  crc32.b rd, rs
  crc32.h rd, rs
  crc32.w rd, rs
  crc32c.b rd, rs
  crc32c.h rd, rs
  crc32c.w rd, rs

RV64 only:
  crc32.d rd, rs
  crc32c.d rd, rs
```

Unary Cyclic Redundancy Check (CRC) instructions that interpret the bits of rs1 as a CRC32/CRC32C state and perform a polynomial reduction of that state shifted left by 8, 16, 32, or 64 bits.

The instructions return the new CRC32/CRC32C state.

The `crc32.w`/`crc32c.w` instructions are equivalent to executing `crc32.h`/`crc32c.h` twice, and `crc32.h`/`crc32c.h` instructions are equivalent to executing `crc32.b`/`crc32c.b` twice.

All 8 CRC instructions operate on bit-reflected data.

```
uint_xlen_t crc32(uint_xlen_t x, int nbits)
{
    for (int i = 0; i < nbits; i++)
        x = (x >> 1) ^ (0xEDB88320 & ~((x&1)-1));
    return x;
}

uint_xlen_t crc32c(uint_xlen_t x, int nbits)
{
    for (int i = 0; i < nbits; i++)
        x = (x >> 1) ^ (0x82F63B78 & ~((x&1)-1));
    return x;
}

uint_xlen_t crc32_b(uint_xlen_t rs1) { return crc32(rs1, 8); }
uint_xlen_t crc32_h(uint_xlen_t rs1) { return crc32(rs1, 16); }
uint_xlen_t crc32_w(uint_xlen_t rs1) { return crc32(rs1, 32); }

uint_xlen_t crc32c_b(uint_xlen_t rs1) { return crc32c(rs1, 8); }
uint_xlen_t crc32c_h(uint_xlen_t rs1) { return crc32c(rs1, 16); }
uint_xlen_t crc32c_w(uint_xlen_t rs1) { return crc32c(rs1, 32); }

#if XLEN > 32
uint_xlen_t crc32_d (uint_xlen_t rs1) { return crc32 (rs1, 64); }
uint_xlen_t crc32c_d(uint_xlen_t rs1) { return crc32c(rs1, 64); }
#endif
```

Payload data must be XOR'ed into the LSB end of the state before executing the CRC instruction. The following code demonstrates the use of `crc32.b`:

```
uint32_t crc32_demo(const uint8_t *p, int len)
{
  uint32_t x = 0xffffffff;
  for (int i = 0; i < len; i++) {
    x = x ^ p[i];
    x = crc32_b(x);
  }
  return ~x;
}
```

In terms of binary polynomial arithmetic those instructions perform the operation

$$\texttt{rd}'(x) = (\texttt{rs1}'(x) \cdot x^N) \bmod \{1, P'\}(x),$$

with $N \in \{8, 16, 32, 64\}$, $P = \texttt{0xEDB8\_8320}$ for CRC32 and $P = \texttt{0x82F6\_3B78}$ for CRC32C, a'

denoting the XLEN bit reversal of a, and $\{a, b\}$ denoting bit concatenation. Note that for example for CRC32 $\{1, P'\}$ = `0x1_04C1_1DB7` on RV32 and $\{1, P'\}$ = `0x1_04C1_1DB7_0000_0000` on RV64.

These dedicated CRC instructions are meant for RISC-V implementations without fast multiplier and therefore without fast `clmul[h]`. For implementations with fast `clmul[h]` it is recommended to use the methods described in [12] and demonstrated in [28] that can process XLEN input bits using just one carry-less multiply for arbitrary CRC polynomials.

In applications where those methods are not applicable it is possible to emulate the dedicated CRC instructions using two carry-less multiplies that implement a Barrett reduction. The following example implements a replacement for `crc32.w` (RV32).

```
crc32_w:
  li t0, 0xF7011641
  li t1, 0xEDB88320
  clmul a0, a0, t0
  clmulr a0, a0, t1
  ret
```

2.8 Bit-Matrix Instructions (`bmatxor`, `bmator`, `bmatflip`, RV64 only)

```
RISC-V Bitmanip ISA
RV64 only:
  bmator rd, rs1, rs2
  bmatxor rd, rs1, rs2
  bmatflip rd, rs
```

These are 64-bit-only instruction that are not available on RV32. On RV128 they ignore the upper half of operands and sign extend the results.

This instructions interpret a 64-bit value as 8x8 binary matrix.

`bmatxor` performs a matrix-matrix multiply with boolean AND as multiply operator and boolean XOR as addition operator.

`bmator` performs a matrix-matrix multiply with boolean AND as multiply operator and boolean OR as addition operator.

`bmatflip` is a unary operator that transposes the source matrix. It is equivalent to `zip; zip; zip` on RV64.

```
uint64_t bmatflip(uint64_t rs1)
{
    uint64_t x = rs1;
    x = shfl64(x, 31);
    x = shfl64(x, 31);
    x = shfl64(x, 31);
    return x;
}

uint64_t bmatxor(uint64_t rs1, uint64_t rs2)
{
    // transpose of rs2
    uint64_t rs2t = bmatflip(rs2);

    uint8_t u[8]; // rows of rs1
    uint8_t v[8]; // cols of rs2

    for (int i = 0; i < 8; i++) {
        u[i] = rs1 >> (i*8);
        v[i] = rs2t >> (i*8);
    }

    uint64_t x = 0;
    for (int i = 0; i < 64; i++) {
        if (cpop(u[i / 8] & v[i % 8]) & 1)
            x |= 1LL << i;
    }

    return x;
}
```

```
uint64_t bmator(uint64_t rs1, uint64_t rs2)
{
    // transpose of rs2
    uint64_t rs2t = bmatflip(rs2);

    uint8_t u[8]; // rows of rs1
    uint8_t v[8]; // cols of rs2

    for (int i = 0; i < 8; i++) {
        u[i] = rs1 >> (i*8);
        v[i] = rs2t >> (i*8);
    }

    uint64_t x = 0;
    for (int i = 0; i < 64; i++) {
        if ((u[i / 8] & v[i % 8]) != 0)
            x |= 1LL << i;
    }

    return x;
}
```

Among other things, `bmatxor`/`bmator` can be used to perform arbitrary permutations of bits within each byte (permutation matrix as 2nd operand) or perform arbitrary permutations of bytes within a 64-bit word (permutation matrix as 1st operand).

There are similar instructions in Cray XMT [6]. The Cray X1 architecture even has a full 64x64 bit matrix multiply unit [5]. (See Section 4.6 for how to implement 64x64 bit matix operations with `bmat[x]or`.)

The MMIX architecture has MOR and MXOR instructions with the same semantic. [16, p. 182f]

The x86 EVEX/VEX/SSE instruction GF2P8AFFINEQB is equivalent to `bmatxor`.

The `bmm.8` instruction proposed in [13] is also equivalent to `bmatxor`.

2.9 Ternary Bit-Manipulation Instructions

2.9.1 Conditional Mix (cmix)

```
RISC-V Bitmanip ISA
RV32, RV64:
  cmix rd, rs2, rs1, rs3
```

(Note that the assembler syntax of `cmix` has the `rs2` argument first to make assembler code more readable. But the reference C code code below uses the “architecturally correct” argument order `rs1, rs2, rs3`.)

The `cmix rd, rs2, rs1, rs3` instruction selects bits from `rs1` and `rs3` based on the bits in the control word `rs2`.

```
uint_xlen_t cmix(uint_xlen_t rs1, uint_xlen_t rs2, uint_xlen_t rs3)
{
    return (rs1 & rs2) | (rs3 & ~rs2);
}
```

It replaces sequences like the following.

```
and rd, rs1, rs2
andn t0, rs3, rs2
or rd, rd, t0
```

Using `cmix` a single butterfly stage can be implemented in only two instructions. Thus, arbitrary bit-permutations can be implemented using only 18 instruction (32 bit) or 22 instructions (64 bits).

2.9.2 Conditional Move (`cmov`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  cmov rd, rs2, rs1, rs3
```

(Note that the assembler syntax of `cmov` has the `rs2` argument first to make assembler code more readable. But the reference C code code below uses the "architecturally correct" argument order `rs1, rs2, rs3`.)

The `cmov rd, rs2, rs1, rs3` instruction selects `rs1` if the control word `rs2` is non-zero, and `rs3` if the control word is zero.

```
uint_xlen_t cmov(uint_xlen_t rs1, uint_xlen_t rs2, uint_xlen_t rs3)
{
    return rs2 ? rs1 : rs3;
}
```

The `cmov` instruction helps avoiding branches, which can lead to better performance, and helps with constant-time code as used in some cryptography applications.

2.9.3 Funnel Shift (`fsl`, `fsr`, `fsri`)

```
RISC-V Bitmanip ISA
RV32, RV64:
  fsl  rd, rs1, rs3, rs2
  fsr  rd, rs1, rs3, rs2
  fsri rd, rs1, rs3, imm

RV64 only:
  fslw  rd, rs1, rs3, rs2
  fsrw  rd, rs1, rs3, rs2
  fsriw rd, rs1, rs3, imm
```

(Note that the assembler syntax for funnel shifts has the `rs2` argument last to make assembler code more readable. But the reference C code code below uses the “architecturally correct” argument order `rs1, rs2, rs3`.)

The `fsl rd, rs1, rs3, rs2` instruction creates a $2 \cdot \text{XLEN}$ word by concatenating rs1 and rs3 (with rs1 in the MSB half), rotate-left-shifts that word by the amount indicated in the $log_2(\text{XLEN})+1$ LSB bits in rs2, and then writes the MSB half of the result to rd.

The `fsr rd, rs1, rs3, rs2` instruction creates a $2 \cdot \text{XLEN}$ word by concatenating rs1 and rs3 (with rs1 in the LSB half), rotate-right-shifts that word by the amount indicated in the $log_2(\text{XLEN}) + 1$ LSB bits in rs2, and then writes the LSB half of the result to rd.

```
uint_xlen_t fsl(uint_xlen_t rs1, uint_xlen_t rs2, uint_xlen_t rs3)
{
    int shamt = rs2 & (2*XLEN - 1);
    uint_xlen_t A = rs1, B = rs3;
    if (shamt >= XLEN) {
        shamt -= XLEN;
        A = rs3;
        B = rs1;
    }
    return shamt ? (A << shamt) | (B >> (XLEN-shamt)) : A;
}

uint_xlen_t fsr(uint_xlen_t rs1, uint_xlen_t rs2, uint_xlen_t rs3)
{
    int shamt = rs2 & (2*XLEN - 1);
    uint_xlen_t A = rs1, B = rs3;
    if (shamt >= XLEN) {
        shamt -= XLEN;
        A = rs3;
        B = rs1;
    }
    return shamt ? (A >> shamt) | (B << (XLEN-shamt)) : A;
}
```

A shift unit capable of either `fsl` or `fsr` is capable of performing all the other shift functions, including the other funnel shift, with only minimal additional logic.

For any values of `A`, `B`, and `C`:

```
fsl(A, B, C) = fsr(A, -B, C)
```

And for any values `x` and $0 \leq$ `shamt` $<$ `XLEN`:

```
sll(x, shamt) == fsl(x, shamt, 0)
srl(x, shamt) == fsr(x, shamt, 0)
sra(x, shamt) == fsr(x, shamt, sext_x)
slo(x, shamt) == fsl(x, shamt, ~0)
sro(x, shamt) == fsr(x, shamt, ~0)
ror(x, shamt) == fsr(x, shamt, x)
rol(x, shamt) == fsl(x, shamt, x)
```

Furthermore an RV64 implementation of either `fsl` or `fsr` is capable of performing the *W versions of all shift operations with only a few gates of additional control logic.

On RV128 there is no `fsri` instruction. But there is `fsriw` and `fsrid`.

2.10 Address calculation instructions

```
RISC-V Bitmanip ISA
RV32, RV64:
  sh1add rd, rs1, rs2
  sh2add rd, rs1, rs2
  sh3add rd, rs1, rs2

RV64 only:
  sh1add.uw rd, rs1, rs2
  sh2add.uw rd, rs1, rs2
  sh3add.uw rd, rs1, rs2
```

These instructions shift `rs1` left by 1, 2, or 3 bits, then add the result to `rs2`. The `sh?add.uw` instructions are identical to `sh?add`, except that bits XLEN-1:32 of the `rs1` argument are cleared before the shift.

```
uint_xlen_t sh1add(uint_xlen_t rs1, uint_xlen_t rs2)
{
    return (rs1 << 1) + rs2;
}

uint_xlen_t sh2add(uint_xlen_t rs1, uint_xlen_t rs2)
{
    return (rs1 << 2) + rs2;
}
```

```
uint_xlen_t sh3add(uint_xlen_t rs1, uint_xlen_t rs2)
{
    return (rs1 << 3) + rs2;
}

uint_xlen_t sh1adduw(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t rs1z = rs1 & 0xFFFFFFFF;
    return (rs1z << 1) + rs2;
}

uint_xlen_t sh2adduw(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t rs1z = rs1 & 0xFFFFFFFF;
    return (rs1z << 2) + rs2;
}

uint_xlen_t sh3adduw(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t rs1z = rs1 & 0xFFFFFFFF;
    return (rs1z << 3) + rs2;
}
```

An opcode for `sh4add`/`sh4add.uw` for RV128 and/or RVQ is reserved.

2.11 Add/shift with prefix zero-extend (`add.uw`, `slli.uw`)

```
RISC-V Bitmanip ISA
RV64:
  add.uw rd, rs1, rs2
  slli.uw rd, rs1, imm
```

`slli.uw` and `add.uw` are identical to `slli` and `add`, respectively, except that bits XLEN-1:32 of the `rs1` argument are cleared before the shift or add.

```
uint_xlen_t slliuw(uint_xlen_t rs1, int imm)
{
    uint_xlen_t rs1u = (uint32_t)rs1;
    int shamt = imm & (XLEN - 1);
    return rs1u << shamt;
}
uint_xlen_t adduw(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t rs1u = (uint32_t)rs1;
    return rs1u + rs2;
}
```

2.12 Opcode Encodings

This chapter contains proposed encodings for most of the instructions described in this document. **DO NOT IMPLEMENT THESE OPCODES YET.** We are trying to get official opcodes assigned and will update this chapter soon with the official opcodes.

The `andn`, `orn`, and `xnor` instruction are encoded the same way as `and`, `or`, and `xor`, but with `op[30]` set, mirroring the encoding scheme used for `add` and `sub`.

All shift instructions use `funct3=001` for left shifts and `funct3=101` for right shifts. Just like in the RISC-V integer base ISA, the shift-immediate instructions have a 5 bit immediate on RV32, and a 6 bit immediate on RV64, and we reserve encoding space for a 7 bit immediate for RV128. The same sizes apply to `bseti`, `bclri`, `binvi`, and `bexti`.

The immediate for `shfli`/`unshufli` is one bit smaller than the immediate for shift instructions, that is 4 bits on RV32, 5 bits on RV64, and we reserve 6 bits for RV128.

`op[26]=1` selects funnel shifts. For funnel shifts `op[30:29]` is part if the 3rd operand and therefore unused for encoding the operation. For all other shift operations `op[26]=0`.

`fsri` is also encoded with `op[26]=1`, leaving a 6 bit immediate. The 7th bit, that is necessary to perform a 128 bit funnel shift on RV64, can be emulated by swapping rs1 and rs3.

There is no `shfliw` instruction. The `slli.uw` instruction occupies the encoding slot that would be occupied by `shfliw`.

On RV128 `op[26]` contains the MSB of the immediate for the shift instructions. Therefore there is no FSRI instruction on RV128. (But there is FSRIW/FSRID.)

	SLL	SRL	SRA	SLO	SRO	ROL	ROR	FSL	FSR
op[30]	0	0	1	0	0	1	1	-	-
op[29]	0	0	0	1	1	1	1	-	-
op[26]	0	0	0	0	0	0	0	1	1
funct3	001	101	101	001	101	001	101	001	101

Only an encoding for RORI exists, as ROLI can be implemented with RORI by negating the immediate. Unary functions are encoded in the spot that would correspond to ROLI, with the function encoded in the 5 LSB bits of the immediate.

The CRC instructions are encoded as unary instructions with `op[24]` set. The polynomial is selected via `op[23]`, with `op[23]=0` for CRC32 and `op[23]=1` for CRC32C. The width is selected with `op[22:20]`, using the same encoding as is used in `funct3` for load/store operations.

`cmix` and `cmov` are encoded using the two remaining ternary operator encodings in `funct3=001` and `funct3=101`. (There are two ternary operator encodings per minor opcode using the `op[26]=1` scheme for marking ternary OPs.)

The single-bit instructions are also encoded within the shift opcodes, with `op[27]` set, and using `op[30]` and `op[29]` to select the operation:

	BCLR	BSET	BINV	BEXT	GORC	GREV
op[30]	1	0	1	1	0	1
op[29]	0	1	1	0	1	1
op[27]	1	1	1	1	1	1
funct3	001	001	001	101	101	101

GORC and GREV are encoded in the two remaining slots in the single-bit instruction encoding space.

The remaining instructions are encoded within `funct7=0000100` and `funct7=0000101`.

The `funct7=0000101` block contains `clmul[hr]`, `min[u]`, and `max[u]`.

The encoding of `clmul, clmulr, clmulh` is identical to the encoding of `mulh, mulhsu, mulhu`, except that `op[27]=1`.

The encoding of `min[u]/max[u]` uses `funct3=100..111`. The `funct3` encoding matches `op[31:29]` of the AMO min/max functions.

The remaining instructions are encoded within `funct7=0000100`. The shift-like `shfl/unshfl` instructions uses the same `funct3` values as the shift operations. `bdecompress` and `bcompress` are encoded in a way so that `funct3[2]` selects the "direction", similar to shift operations.

`bmat[x]or` use `funct3=011` and `funct3=111` in `funct7=0000100`.

`pack` occupies `funct3=100` in `funct7=0000100`.

`add.uw` is encoded like `addw`, except that `op[27]=1`.

Finally, RV64 has `*W` instructions for all bitmanip instructions, with the following exceptions:

`andn`, `cmix`, `cmov`, `min[u]`, `max[u]` have no `*W` variants because they already behave in the way a `*W` instruction would when presented with sign-exteded 32-bit arguments.

`bmatflip`, `bmatxor`, `bmator` have no `*W` variants because they are 64-bit only instructions.

`crc32.[bhwd]`, `crc32c.[bhwd]` have no `*W` variants because `crc32[c].w` is deemed sufficient.

There is no `[un]shfliw`, as a perfect outer shuffle always preserves the MSB bit, thus `[un]shfli` preserves proper sign extension when the upper bit in the control word is set. There's still `[un]shflw` that masks that upper control bit and sign-extends the output.

Relevant instruction encodings from the base ISA are included in the table below and are marked with a `*`.

```
|  3                   2                   1                    |
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|
|---------------------------------------------------------------|
|    funct7    |    rs2   |    rs1   |  f3 |    rd    |   opcode    |  R-type
|   rs3   | f2|    rs2   |    rs1   |  f3 |    rd    |   opcode    |  R4-type
|         imm            |    rs1   |  f3 |    rd    |   opcode    |  I-type
|===============================================================|
|  0000000     |    rs2   |    rs1   | 111 |    rd    |  0110011    |  AND*
|  0000000     |    rs2   |    rs1   | 110 |    rd    |  0110011    |  OR*
|  0000000     |    rs2   |    rs1   | 100 |    rd    |  0110011    |  XOR*
|  0100000     |    rs2   |    rs1   | 111 |    rd    |  0110011    |  ANDN
|  0100000     |    rs2   |    rs1   | 110 |    rd    |  0110011    |  ORN
|  0100000     |    rs2   |    rs1   | 100 |    rd    |  0110011    |  XNOR
|---------------------------------------------------------------|
|  0000000     |    rs2   |    rs1   | 001 |    rd    |  0110011    |  SLL*
|  0000000     |    rs2   |    rs1   | 101 |    rd    |  0110011    |  SRL*
|  0100000     |    rs2   |    rs1   | 101 |    rd    |  0110011    |  SRA*
|  0010000     |    rs2   |    rs1   | 001 |    rd    |  0110011    |  SLO
|  0010000     |    rs2   |    rs1   | 101 |    rd    |  0110011    |  SRO
|  0110000     |    rs2   |    rs1   | 001 |    rd    |  0110011    |  ROL
|  0110000     |    rs2   |    rs1   | 101 |    rd    |  0110011    |  ROR
|---------------------------------------------------------------|
|  0010000     |    rs2   |    rs1   | 010 |    rd    |  0110011    |  SH1ADD
|  0010000     |    rs2   |    rs1   | 100 |    rd    |  0110011    |  SH2ADD
|  0010000     |    rs2   |    rs1   | 110 |    rd    |  0110011    |  SH3ADD
|---------------------------------------------------------------|
|  0100100     |    rs2   |    rs1   | 001 |    rd    |  0110011    |  BCLR
|  0010100     |    rs2   |    rs1   | 001 |    rd    |  0110011    |  BSET
|  0110100     |    rs2   |    rs1   | 001 |    rd    |  0110011    |  BINV
|  0100100     |    rs2   |    rs1   | 101 |    rd    |  0110011    |  BEXT
|  0010100     |    rs2   |    rs1   | 101 |    rd    |  0110011    |  GORC
|  0110100     |    rs2   |    rs1   | 101 |    rd    |  0110011    |  GREV
|---------------------------------------------------------------|
|  00000  |      imm      |    rs1   | 001 |    rd    |  0010011    |  SLLI*
|  00000  |      imm      |    rs1   | 101 |    rd    |  0010011    |  SRLI*
|  01000  |      imm      |    rs1   | 101 |    rd    |  0010011    |  SRAI*
|  00100  |      imm      |    rs1   | 001 |    rd    |  0010011    |  SLOI
|  00100  |      imm      |    rs1   | 101 |    rd    |  0010011    |  SROI
|  01100  |      imm      |    rs1   | 101 |    rd    |  0010011    |  RORI
|---------------------------------------------------------------|
|  01001  |      imm      |    rs1   | 001 |    rd    |  0010011    |  BCLRI
|  00101  |      imm      |    rs1   | 001 |    rd    |  0010011    |  BSETI
|  01101  |      imm      |    rs1   | 001 |    rd    |  0010011    |  BINVI
|  01001  |      imm      |    rs1   | 101 |    rd    |  0010011    |  BEXTI
|  00101  |      imm      |    rs1   | 101 |    rd    |  0010011    |  GORCI
|  01101  |      imm      |    rs1   | 101 |    rd    |  0010011    |  GREVI
|---------------------------------------------------------------|
|   rs3   | 11|    rs2   |    rs1   | 001 |    rd    |  0110011    |  CMIX
|   rs3   | 11|    rs2   |    rs1   | 101 |    rd    |  0110011    |  CMOV
|   rs3   | 10|    rs2   |    rs1   | 001 |    rd    |  0110011    |  FSL
|   rs3   | 10|    rs2   |    rs1   | 101 |    rd    |  0110011    |  FSR
|   rs3   |1|     imm     |    rs1   | 101 |    rd    |  0010011    |  FSRI
|---------------------------------------------------------------|
|  3                   2                   1                    |
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|
|===============================================================|
|  0110000    |  00000  |   rs1   | 001 |    rd   |   0010011   |  CLZ
|  0110000    |  00001  |   rs1   | 001 |    rd   |   0010011   |  CTZ
|  0110000    |  00010  |   rs1   | 001 |    rd   |   0010011   |  CPOP
|  0110000    |  00011  |   rs1   | 001 |    rd   |   0010011   |  BMATFLIP
|  0110000    |  00100  |   rs1   | 001 |    rd   |   0010011   |  SEXT.B
|  0110000    |  00101  |   rs1   | 001 |    rd   |   0010011   |  SEXT.H
|---------------------------------------------------------------|
|  0110000    |  10000  |   rs1   | 001 |    rd   |   0010011   |  CRC32.B
|  0110000    |  10001  |   rs1   | 001 |    rd   |   0010011   |  CRC32.H
|  0110000    |  10010  |   rs1   | 001 |    rd   |   0010011   |  CRC32.W
|  0110000    |  10011  |   rs1   | 001 |    rd   |   0010011   |  CRC32.D
|  0110000    |  11000  |   rs1   | 001 |    rd   |   0010011   |  CRC32C.B
|  0110000    |  11001  |   rs1   | 001 |    rd   |   0010011   |  CRC32C.H
|  0110000    |  11010  |   rs1   | 001 |    rd   |   0010011   |  CRC32C.W
|  0110000    |  11011  |   rs1   | 001 |    rd   |   0010011   |  CRC32C.D
|---------------------------------------------------------------|
|  0000101    |   rs2   |   rs1   | 001 |    rd   |   0110011   |  CLMUL
|  0000101    |   rs2   |   rs1   | 010 |    rd   |   0110011   |  CLMULR
|  0000101    |   rs2   |   rs1   | 011 |    rd   |   0110011   |  CLMULH
|  0000101    |   rs2   |   rs1   | 100 |    rd   |   0110011   |  MIN
|  0000101    |   rs2   |   rs1   | 101 |    rd   |   0110011   |  MINU
|  0000101    |   rs2   |   rs1   | 110 |    rd   |   0110011   |  MAX
|  0000101    |   rs2   |   rs1   | 111 |    rd   |   0110011   |  MAXU
|---------------------------------------------------------------|
|  0000100    |   rs2   |   rs1   | 001 |    rd   |   0110011   |  SHFL
|  0000100    |   rs2   |   rs1   | 101 |    rd   |   0110011   |  UNSHFL
|  0100100    |   rs2   |   rs1   | 110 |    rd   |   0110011   |  BDECOMPRESS
|  0000100    |   rs2   |   rs1   | 110 |    rd   |   0110011   |  BCOMPRESS
|  0000100    |   rs2   |   rs1   | 100 |    rd   |   0110011   |  PACK
|  0100100    |   rs2   |   rs1   | 100 |    rd   |   0110011   |  PACKU
|  0000100    |   rs2   |   rs1   | 011 |    rd   |   0110011   |  BMATOR
|  0100100    |   rs2   |   rs1   | 011 |    rd   |   0110011   |  BMATXOR
|  0000100    |   rs2   |   rs1   | 111 |    rd   |   0110011   |  PACKH
|  0100100    |   rs2   |   rs1   | 111 |    rd   |   0110011   |  BFP
|---------------------------------------------------------------|
|  000010   |    imm    |   rs1   | 001 |    rd   |   0010011   |  SHFLI
|  000010   |    imm    |   rs1   | 101 |    rd   |   0010011   |  UNSHFLI
|===============================================================|
|  00001  |     imm     |   rs1   | 001 |    rd   |   0011011   |  SLLI.UW
|  0000100    |   rs2   |   rs1   | 000 |    rd   |   0111011   |  ADD.UW
|---------------------------------------------------------------|
|  3                   2                   1                    |
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|
|===============================================================|
|  0010000    |   rs2   |   rs1   | 001 |    rd   |   0111011   |  SLOW
|  0010000    |   rs2   |   rs1   | 101 |    rd   |   0111011   |  SROW
|  0110000    |   rs2   |   rs1   | 001 |    rd   |   0111011   |  ROLW
|  0110000    |   rs2   |   rs1   | 101 |    rd   |   0111011   |  RORW
|---------------------------------------------------------------|
|  0010000    |   rs2   |   rs1   | 010 |    rd   |   0111011   |  SH1ADD.UW
|  0010000    |   rs2   |   rs1   | 100 |    rd   |   0111011   |  SH2ADD.UW
|  0010000    |   rs2   |   rs1   | 110 |    rd   |   0111011   |  SH3ADD.UW
|---------------------------------------------------------------|
|  0010100    |   rs2   |   rs1   | 101 |    rd   |   0111011   |  GORCW
|  0110100    |   rs2   |   rs1   | 101 |    rd   |   0111011   |  GREVW
|---------------------------------------------------------------|
|  0010000    |   imm   |   rs1   | 001 |    rd   |   0011011   |  SLOIW
|  0010000    |   imm   |   rs1   | 101 |    rd   |   0011011   |  SROIW
|  0110000    |   imm   |   rs1   | 101 |    rd   |   0011011   |  RORIW
|---------------------------------------------------------------|
|  0010100    |   imm   |   rs1   | 101 |    rd   |   0011011   |  GORCIW
|  0110100    |   imm   |   rs1   | 101 |    rd   |   0011011   |  GREVIW
|---------------------------------------------------------------|
|   rs3    | 10|   rs2   |   rs1   | 001 |    rd   |   0111011   |  FSLW
|   rs3    | 10|   rs2   |   rs1   | 101 |    rd   |   0111011   |  FSRW
|   rs3    | 10|   imm   |   rs1   | 101 |    rd   |   0011011   |  FSRIW
|---------------------------------------------------------------|
|  0110000    |  00000  |   rs1   | 001 |    rd   |   0011011   |  CLZW
|  0110000    |  00001  |   rs1   | 001 |    rd   |   0011011   |  CTZW
|  0110000    |  00010  |   rs1   | 001 |    rd   |   0011011   |  CPOPW
|---------------------------------------------------------------|
|  0000100    |   rs2   |   rs1   | 001 |    rd   |   0111011   |  SHFLW
|  0000100    |   rs2   |   rs1   | 101 |    rd   |   0111011   |  UNSHFLW
|  0100100    |   rs2   |   rs1   | 110 |    rd   |   0111011   |  BDECOMPRESSW
|  0000100    |   rs2   |   rs1   | 110 |    rd   |   0111011   |  BCOMPRESSW
|  0000100    |   rs2   |   rs1   | 100 |    rd   |   0111011   |  PACKW
|  0100100    |   rs2   |   rs1   | 100 |    rd   |   0111011   |  PACKUW
|  0100100    |   rs2   |   rs1   | 111 |    rd   |   0111011   |  BFPW
|---------------------------------------------------------------|

|  3                   2                   1                    |
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|
|===============================================================|
|  0010100    |   rs2   |   rs1   | 010 |    rd   |   0110011   |  XPERM.N
|  0010100    |   rs2   |   rs1   | 100 |    rd   |   0110011   |  XPERM.B
|  0010100    |   rs2   |   rs1   | 110 |    rd   |   0110011   |  XPERM.H
|  0010100    |   rs2   |   rs1   | 000 |    rd   |   0110011   |  XPERM.W
|---------------------------------------------------------------|
```

Encoding changes in v0.93 of the RISC-V Bitmanip Spec (+ for addition, - for removal):

```
|  3                   2                   1                 |
|1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|
|================================================================|
|  0010000     |   rs2   |   rs1   | 010 |    rd    |   0110011   |  + SH1ADD
|  0010000     |   rs2   |   rs1   | 100 |    rd    |   0110011   |  + SH2ADD
|  0010000     |   rs2   |   rs1   | 110 |    rd    |   0110011   |  + SH3ADD
|----------------------------------------------------------------|
|  0010000     |   rs2   |   rs1   | 010 |    rd    |   0111011   |  + SH1ADD.UW
|  0010000     |   rs2   |   rs1   | 100 |    rd    |   0111011   |  + SH2ADD.UW
|  0010000     |   rs2   |   rs1   | 110 |    rd    |   0111011   |  + SH3ADD.UW
|----------------------------------------------------------------|
|  0000101     |   rs2   |   rs1   | 101 |    rd    |   0110011   |  - MAX
|  0000101     |   rs2   |   rs1   | 110 |    rd    |   0110011   |  - MINU
|  0000101     |   rs2   |   rs1   | 110 |    rd    |   0110011   |  + MAX
|  0000101     |   rs2   |   rs1   | 101 |    rd    |   0110011   |  + MINU
|----------------------------------------------------------------|
|  0100100     |   rs2   |   rs1   | 000 |    rd    |   0111011   |  - SUBU.W
|----------------------------------------------------------------|
|  0010100     |   rs2   |   rs1   | 010 |    rd    |   0110011   |  + XPERM.N
|  0010100     |   rs2   |   rs1   | 100 |    rd    |   0110011   |  + XPERM.B
|  0010100     |   rs2   |   rs1   | 110 |    rd    |   0110011   |  + XPERM.H
|  0010100     |   rs2   |   rs1   | 000 |    rd    |   0110011   |  + XPERM.W
|----------------------------------------------------------------|
|       immediate         |   rs1   | 100 |    rd    |   0011011   |  - ADDIWU
|  0000101     |   rs2   |   rs1   | 000 |    rd    |   0111011   |  - ADDWU
|  0100101     |   rs2   |   rs1   | 000 |    rd    |   0111011   |  - SUBWU
|----------------------------------------------------------------|
|  0000101     |   rs2   |   rs1   | 001 |    rd    |   0111011   |  - CLMULW
|  0000101     |   rs2   |   rs1   | 010 |    rd    |   0111011   |  - CLMULRW
|  0000101     |   rs2   |   rs1   | 011 |    rd    |   0111011   |  - CLMULHW
|----------------------------------------------------------------|
```

2.13 Future compressed instructions

The RISC-V ISA has no dedicated instructions for bitwise inverse (`not`). Instead `not` is implemented as `xori rd, rs, -1` and `neg` is implemented as `sub rd, x0, rs`.

In bitmanipulation code `not` is a very common operation. But there is no compressed encoding for those operation because there is no `c.xori` instruction.

On RV64, `zext.w` (`add.uw`) is commonly used to zero-extend unsigned words. For RV128, `zext.d` is expected to be similarly common, and might map to a hypothetical `add.ud` instruction.

It presumably would make sense for a future revision of the "C" extension to include compressed opcodes for those instructions.

An encoding with the constraint `rd` = `rs` would fit nicely in the reserved space in `c.addi16sp/c.lui`.

funct7	funct3							
30|29|27|25	001	101	000	010	011	100	110	111
-00-0-0	SLL	SRL	ADD	SLT^	SLTU^	XOR^	OR^	AND^
-10-0-0		SRA	SUB			XNOR^	ORN^	ANDN^
-00-0-1	MULH^(2)	DIVU (2)	MUL	MULHSU^	MULHU^	DIV	REM	REMU
-10-0-1	(2)	(2)						
-00-1-0	SHFL (4)	UNSHFL	ADD.UW (1)		BMATOR^	PACK	BDECOMPRESS	PACKH^
-10-1-0	BCLR	BEXT			BMATXOR^	PACKU	BCOMPRESS	BFP
-00-1-1	CLMUL^(2)	MINU^(2)		CLMULR^	CLMULH^	MIN^	MAX^	MAXU^
-10-1-1	(2)	(2)						
-01-0-0	SLO	SRO	(SH4ADD)	SH1ADD		SH2ADD	SH3ADD	
-11-0-0	ROL (3)	ROR						
-01-0-1	(2)	(2)						
-11-0-1	(2)	(2)						
-01-1-0	BSET	GORC	XPERM.W	XPERM.N		XPERM.B	XPERM.H	
-11-1-0	BINV	GREV						
-01-1-1	(2)	(2)						
-11-1-1	(2)	(2)						

```
Notes:
- funct7 bits not shown: 31,28 unused (always zero), 26 used for ternary instructions
- columns reordered to show shift opcodes on the left
- rows reordered in groups with and without bit 30 set
(1) These instructions only exist in OP-32.
(2) No "shift-immediate" encoding for opcodes with bit 25 set.
(3) All unary instructions use the code space for the non-existing ROLI instruction.
(4) SLLI.UW is encoded in the code space for the non-existing SHFLIW instruction.
^ Instructions marked with ^ have no *W equivalent in OP-32
```

Figure 2.5: OP/OP-32 Binary Instruction Map

funct7	funct3							
RS3|26|25	001	101	000	010	011	100	110	111
-----10	FSL (2)	FSR (1)						
-----11	CMIX	CMOV						

```
Notes:
- funct7 bits: bits 31:27 RS3, bit 26 always set for ternary instructions
- columns reordered to show shift opcodes on the left
(1) There is also an FSRI instruction in OP-IMM/OP-IMM-32 (except OP-IMM on RV128)
(2) The encoding for the non-existing FSLI instruction is reserved
(RV128 FSRI could be implemented using the reserved FSLI opcodes in OP-IMM-32/OP-IMM-64)
```

Figure 2.6: OP/OP-32 Ternary Instruction Map

The entire RVC encoding space is 15.585 bits wide, the remaining reserved encoding space in RVC is 11.155 bits wide, not including space that is only reserved on RV32/RV64. This means that above encoding would use 0.0065% of the RVC encoding space, or 1.4% of the remaining reserved RVC encoding space. Preliminary experiments have shown that NOT instructions alone make up approximately 1% of bitmanipulation code size. [29]

2.14 Macro-op fusion patterns

Some bitmanip operations have been left out of this spec because of lack of a (sensible) way to encode them in the 32-bit RISC-V encoding space. Instead we present fuse-able sequences of up to three instructions for those operations, so that high-end processors can implement them in a single fused macro-op, should they decide to do so.

For this document we only consider sequences fuse-able if they read at most two registers, only

15 14 13	12	11 10	9 8 7	6 5 4 3 2	1 0	
011	nzimm[9]	2		nzimm[4|6|8:7|5]	01	C.ADDI16SP *(~~RES, nzimm=0~~)*
011	nzimm[17]	rd≠{0, 2}		nzimm[16:12]	01	C.LUI *(~~RES, nzimm=0~~; HINT, rd=0)*
011	0	00	rs1′/rd′	0	01	C.NOT
011	0	01	rs1′/rd′	0	01	C.NEG
011	0	10	rs1′/rd′	0	01	C.ZEXT.W *(RV64/128; RV32 RES)*
011	0	11	rs1′/rd′	0	01	C.ZEXT.D *(RV128; RV32/64 RES)*

write one register, and contain no branch instructions.

2.14.1 Fused `*-bfp` sequences

The `bfp` instruction is most commonly used in sequences of one the the following forms.

For 32-bit (RV32 or *W instructions on RV64):

```
addi rd, zero, ...
pack[w] rd, rs2, rd
bfp[w] rd, rs1, rd
```

```
lui rd, ...
addi rd, rd, ...
bfp[w] rd, rs1, rd
```

And for 64-bit:

```
lui rd, ...
pack rd, rs2, rd
bfp rd, rs1, rd
```

2.14.2 Load-immediate

For 32-bit code (RV32 or *W instructions on RV64) we recommend to fuse the `lui+addi` pattern for loading a 32-bit constants:

```
lui rd, imm
addi[w] rd, rd, imm
```

Further, for loading 64-bit constants in two macro-ops:

```
lui rd, imm
addiw rd, rd, imm
pack rd, rd, rs
```

2.14.3 Fused shift sequences

Pairs of left and right shifts are common operations for extracting a bit field.

To extract the contiguous bit field starting at `pos` with length `len` from `rs` (with $\texttt{pos} > 0$, $\texttt{len} > 0$, and $\texttt{pos} + \texttt{len} \leq \text{XLEN}$):

```
  slli rd, rs, (XLEN-len-pos)
  srli rd, rd, (XLEN-len)
```

Using `srai` instead of `srli` will sign-extend the extracted bit-field.

Similarly, placing a bit field with length `len` at the position `pos`:

```
  slli rd, rs, (XLEN-len)
  srli rd, rd, (XLEN-len-pos)
```

Note that the postfix right shift instruction can use a compressed encoding, yielding a 48-bit fused instruction if the left shift is a 32-bit instruction.

More generally, a processor might fuse all destructive shift operations following any other ALU operation.

We define the following assembler pseudo-ops for `sr[la]i` postfix fusion:

```
  bfext  rd, rs, len, pos    ->   slli rd, rs, (XLEN-len-pos); srai rd, rd, (XLEN-len)
  bfextu rd, rs, len, pos    ->   slli rd, rs, (XLEN-len-pos); srli rd, rd, (XLEN-len)
  bfmak  rd, len, pos        ->   sroi rd, zero, len; srli rd, rd, (XLEN-len-pos)
```

(The names `bfext`, `bfextu`, and `bfmak` are borrowed from m88k, that had dedicated instructions of those names (without `bf`-prefix) with equivalent semantics. [4, p. 3-28])

2.15 Other micro-architectural considerations

In addition to macro-op fusion, we issue the following recommendations for cores that aim at better performance for bitmanipulation tasks.

2.15.1 Unaligned memory access

The base ISA spec requires load/store operations that are not naturally aligned to succeed in U-mode, but explicitly states that execution may be slow.

For many bitmanipulation tasks it can be of great advantage to be able to perform load and store operations with arbitrary alignments quickly. Thus we recommend that cores optimized for bitmanipulation tasks provide fast hardware support for load/store with arbitrary alignment.

There should be a `getauxval()`-based mechanism as part of the RISC-V Linux ABI that can be used to query information on support for unaligned memory access. [3]

2.15.2 Fast multiply

A lot of bitmanipulation tricks rely on multiplication with "magic numbers" and similar tricks involving multiply/divide instructions. Thus, cores optimized for bitmanipulation tasks should provide reasonably fast implementations of the "M"-extension multiply/divide instructions.

2.16 C intrinsics via `<rvintrin.h>`

A C header file `<rvintrin.h>` is provided that contains assembler templates for directly creating assembler instructions from C code.

The header defines `_rv_*(...)` functions that operate on the `long` data type, `_rv32_*(...)` functions that operate on the `int32_t` data type, and `_rv64_*(...)` functions that operate on the `int64_t` data type. The `_rv64_*(...)` functions are only available on RV64. See table 2.10 for a complete list of intrinsics defined in `<rvintrin.h>`.

Usage example:

```
#include <rvintrin.h>

int find_nth_set_bit(unsigned int value, int cnt) {
  return _rv32_ctz(_rv32_bdep(1 << cnt, value));
}
```

Defining `RVINTRIN_EMULATE` before including `<rvintrin.h>` will define plain C functions that emulate the behavior of the RISC-V instructions. This is useful for testing software on non-RISC-V platforms.

	RV32		RV64		
Instruction	`_rv_*`	`_rv32_*`	`_rv_*`	`_rv32_*`	`_rv64_*`
`clz`	✓	✓	✓	✓	✓
`ctz`	✓	✓	✓	✓	✓
`cpop`	✓	✓	✓	✓	✓
`pack`	✓	✓	✓	✓	✓
`min`	✓	✓	✓	✓	✓
`minu`	✓	✓	✓	✓	✓
`max`	✓	✓	✓	✓	✓
`maxu`	✓	✓	✓	✓	✓
`bset`	✓	✓	✓	✓	✓
`bclr`	✓	✓	✓	✓	✓
`binv`	✓	✓	✓	✓	✓
`bext`	✓	✓	✓	✓	✓
`sll`	✓	✓	✓	✓	✓
`srl`	✓	✓	✓	✓	✓
`sra`	✓	✓	✓	✓	✓
`slo`	✓	✓	✓	✓	✓
`sro`	✓	✓	✓	✓	✓
`rol`	✓	✓	✓	✓	✓
`ror`	✓	✓	✓	✓	✓
`grev`	✓	✓	✓	✓	✓
`gorc`	✓	✓	✓	✓	✓
`shfl`	✓	✓	✓	✓	✓
`unshfl`	✓	✓	✓	✓	✓
`bfp`	✓	✓	✓	✓	✓
`bcompress`	✓	✓	✓	✓	✓
`bdecompress`	✓	✓	✓	✓	✓
`clmul`	✓	✓	✓	✓	✓
`clmulh`	✓	✓	✓	✓	✓
`clmulr`	✓	✓	✓	✓	✓
`bmatflip`			✓		✓
`bmator`			✓		✓
`bmatxor`			✓		✓
`fsl`	✓	✓	✓	✓	✓
`fsr`	✓	✓	✓	✓	✓
`cmix`	✓		✓		
`cmov`	✓		✓		
`crc32_b`	✓		✓		
`crc32_h`	✓		✓		
`crc32_w`	✓		✓		
`crc32_d`			✓		
`crc32c_b`	✓		✓		
`crc32c_h`	✓		✓		
`crc32c_w`	✓		✓		
`crc32c_d`			✓		

Table 2.10: C intrinsics defined in `<rvintrin.h>`

Chapter 3

Reference Implementations

3.1 Verilog reference implementations

We have implemented Verilog cores for all instructions proposed in this specification. These cores are permissively licensed under the ISC license and can be obtained from `https://github.com/riscv/riscv-bitmanip/tree/master/verilog`.

For evaluation purposes we synthesized these cores for RV32 and RV64 to the following mockup ASIC cell library:

Cell	Gate Count
NOT	0.5
NAND	1
NOR	1
XOR	3
XNOR	3
DFF	4

Cell	Gate Count
AOI3	1.5
OAI3	1.5
AOI4	2
OAI4	2
NMUX	2.5
MUX	3

For comparison we also synthesized the rocket-chip MulDiv cores obtained using the following rocket-chip configurations:

```
class MulDivConfig64 extends Config(
    new WithFastMulDiv ++
    new DefaultConfig
)

class MulDivConfig32 extends Config(
    new WithRV32 ++
    new WithFastMulDiv ++
    new DefaultConfig
)
```

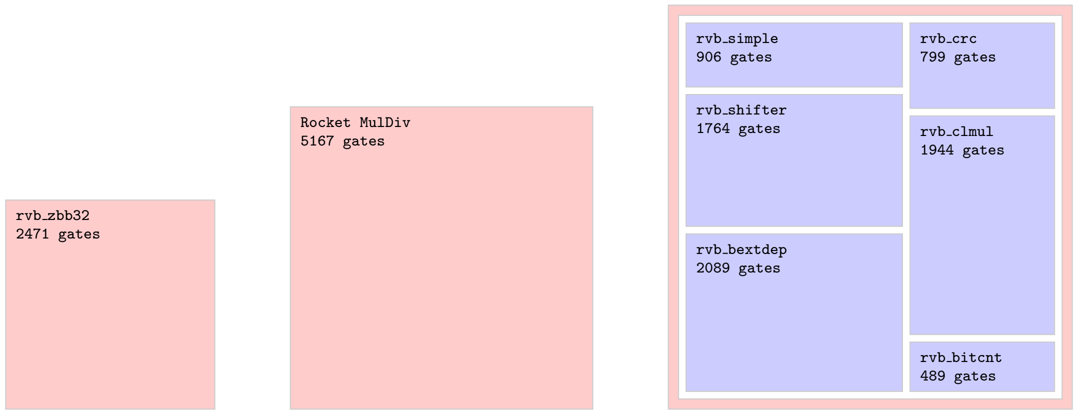


Figure 3.1: Area of 32-bit Rocket MulDiv core (center) compared to a complete implementation of all 32-bit instructions proposed in this specification (right), and the 32-bit “Zbb” extension (left).

The following table lists the verilog reference cores and the instructions they implement:

Module	Instructions
`rvb_bextdep`	bext bdep grev gorc shfl unshfl
`rvb_clmul`	clmul clmulr clmulh
`rvb_shifter`	sll srl sra slo sro rol ror fsl fsr slli.uw bset bclr binv bext bfp
`rvb_bmatxor`	bmatxor bmator
`rvb_simple`	min max minu maxu andn orn xnor pack cmix cmov addiwu addwu subwu adduw subuw
`rvb_bitcnt`	clz ctz cpop bmatflip
`rvb_crc`	crc32.[bhwd] crc32c.[bhwd]
`rvb_full`	All of the above

On RV64 these cores also implement all `*W` instruction variants of the above instructions.

Note that `rvb_shifter` also implements the base ISA `sll`, `srl`, and `sra` instructions. Thus it can replace an existing implementation of the base ISA shift instructions.

Fig. 3.1 shows the area comparison for RV32 and fig. 3.2 shows the comparison for RV64. The area of the red frame surrounding the blue `rvb_*` modules accurately represents the added area by the `rvb_full` wrapper module.

Regarding timing we evaluate the longest paths for `rvb_full` and rocket-chip `MulDiv`, measured in gate delays:

	RV32	RV64
`rvb_full`	30	57
`MulDiv`	43	68

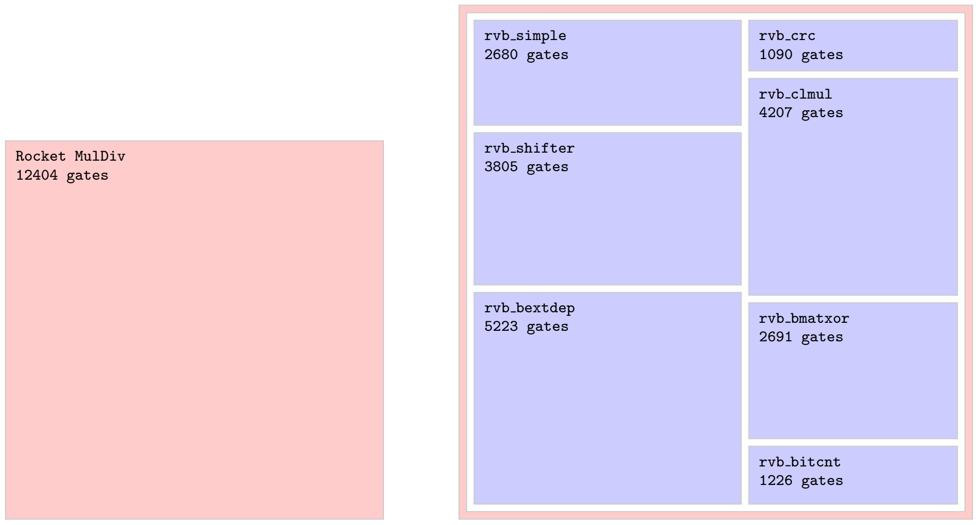


Figure 3.2: Area of 64-bit Rocket MulDiv core (left) compared to a complete implementation of all 64-bit instructions proposed in this specification (right).

All `rvb_*` reference cores provide single-cycle implementations of their functions, with the exception of `rvb_clmul` which requires 4 cycles for a 32-bit carry-less multiply and 8 cycles for a 64-bit carry-less multiply, and `rvb_crc` which requires 1 cycle for each payload byte.

3.2 Fast C reference implementations

GCC has intrinsics for the bit counting instructions `clz`, `ctz`, and `cpop`. So a performance-sensitive application (such as an emulator) should probably just use those:

```
uint32_t fast_clz32(uint32_t rs1)
{
    if (rs1 == 0)
        return 32;
    assert(sizeof(int) == 4);
    return __builtin_clz(rs1);
}

uint64_t fast_clz64(uint64_t rs1)
{
    if (rs1 == 0)
        return 64;
    assert(sizeof(long long) == 8);
    return __builtin_clzll(rs1);
}
```

```
uint32_t fast_ctz32(uint32_t rs1)
{
    if (rs1 == 0)
        return 32;
    assert(sizeof(int) == 4);
    return __builtin_ctz(rs1);
}

uint64_t fast_ctz64(uint64_t rs1)
{
    if (rs1 == 0)
        return 64;
    assert(sizeof(long long) == 8);
    return __builtin_ctzll(rs1);
}

uint32_t fast_pcnt32(uint32_t rs1)
{
    assert(sizeof(int) == 4);
    return __builtin_popcount(rs1);
}

uint64_t fast_pcnt64(uint64_t rs1)
{
    assert(sizeof(long long) == 8);
    return __builtin_popcountll(rs1);
}
```

For processors with BMI2 support GCC has intrinsics for bit extract and bit deposit instructions (compile with `-mbmi2` and include `<x86intrin.h>`):

```
uint32_t fast_bext32(uint32_t rs1, uint32_t rs2)
{
    return _pext_u32(rs1, rs2);
}

uint64_t fast_bext64(uint64_t rs1, uint64_t rs2)
{
    return _pext_u64(rs1, rs2);
}

uint32_t fast_bdep32(uint32_t rs1, uint32_t rs2)
{
    return _pdep_u32(rs1, rs2);
}

uint64_t fast_bdep64(uint64_t rs1, uint64_t rs2)
{
    return _pdep_u64(rs1, rs2);
}
```

For other processors we need to provide our own implementations. The following implementation is a good compromise between code complexity and runtime:

```
uint_xlen_t fast_bext(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t c = 0, i = 0, mask = rs2;
    while (mask) {
        uint_xlen_t b = mask & ~((mask | (mask-1)) + 1);
        c |= (rs1 & b) >> (fast_ctz(b) - i);
        i += fast_pcnt(b);
        mask -= b;
    }
    return c;
}

uint_xlen_t fast_bdep(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t c = 0, i = 0, mask = rs2;
    while (mask) {
        uint_xlen_t b = mask & ~((mask | (mask-1)) + 1);
        c |= (rs1 << (fast_ctz(b) - i)) & b;
        i += fast_pcnt(b);
        mask -= b;
    }
    return c;
}
```

For the other Bitmanip instructions the C reference functions given in Chapter 2 are already reasonably efficient.

3.3 Bit permutation instructions as bit-index operations

For programs that synthesize sequences of bit permutation instructions it can be useful to describe bit permutation instructions in terms of bit-index operations.

Expressed as bit-index operation, `ror` is just subtraction and `grev` is just XOR:

```
uint_xlen_t ror(uint_xlen_t a, uint_xlen_t b)
{
  uint_xlen_t ret = 0;
  for (int i = 0; i < XLEN; i++) {
    int j = (i - b) & (XLEN-1);
    ret |= ((a >> i) & 1) << j;
  }
  return ret;
}
```

```
uint_xlen_t grev(uint_xlen_t a, uint_xlen_t b)
{
  uint_xlen_t ret = 0;
  for (int i = 0; i < XLEN; i++) {
    int j = (i ^ b) & (XLEN-1);
    ret |= ((a >> i) & 1) << j;
  }
  return ret;
}
```

The following `unperm()` function calculates the new position of bit `i` after `unshfl` by `k`.

```
int unperm(int k, int i)
{
  return ((k + (i & k & ~(k<<1))) & ~k) |
         (i & ~(k | (k<<1))) | ((i>>1) & k);
}
```

This allows us to write `shfl` and `unshfl` in the following way.

```
uint_xlen_t shfl(uint_xlen_t a, uint_xlen_t b)
{
  uint_xlen_t ret = 0;
  for (int i = 0; i < XLEN; i++) {
    int j = unperm(b & (XLEN/2-1), i);
    ret |= ((a >> j) & 1) << i;
  }
  return ret;
}

uint_xlen_t unshfl(uint_xlen_t a, uint_xlen_t b)
{
  uint_xlen_t ret = 0;
  for (int i = 0; i < XLEN; i++) {
    int j = unperm(b & (XLEN/2-1), i);
    ret |= ((a >> i) & 1) << j;
  }
  return ret;
}
```

Chapter 4

Example Applications

This chapter contains a collection of short code snippets and algorithms using the Bitmanip extension. It also contains some examples of bit manipulation code that doesn't require any extension beyond the base ISA.

4.1 Basic Bitmanipulation

4.1.1 Loading constants

On RV32 any arbitrary constant can be loaded using LUI+ADDI.

```
  lui a0, ((v - (v << 20 >> 20)) >> 12) & 0xfffff
  addi a0, a0, (v << 20 >> 20)
```

(Assuming signed 32-bit arithmetic for the expression `(v << 20 >> 20)`.)

Using the following sequence on RV64 will yield the 32-bit constant in sign-extended form.

```
  lui a0, ((v - (v << 52 >> 52)) >> 12) & 0xfffff
  addiw a0, a0, (v << 52 >> 52)
```

(`addiw` is needed instead of `addi` to handle the cases correctly that have bits 11-30 of the constant set to one.)

Using `addiwu` instead of `addiw` produces a zero-extended version of the same constant, iff any of the bits 11-31 of the constant is zero.

```
  lui a0, ((v - (v << 52 >> 52)) >> 12) & 0xfffff
  addiwu a0, a0, (v << 52 >> 52)
```

In the remaining cases with bits 11-31 all set, `addi`+`pack` can be used to produce the constant:

```
addi a0, zero, v
pack a0, a0, zero
```

64-bit constants of the form $R \times 1$ $S \times 0$ $T \times 1$ (with $R + S + T = 64$) can be constructed using `sloi` and `rori`:

```
sloi a0, zero, R+T
rori a0, a0, R
```

Likewise, constructing $R \times 0$ $S \times 1$ $T \times 0$:

```
sloi a0, zero, S
slli a0, a0, T
```

Any constant that is just a repeating 16-bit pattern can be constructed in two instructions with `lui` and `orc16`. For example, constructing `0xABCD_ABCD_ABCD_ABCD`:

```
lui a0, 0x0BCDA
orc16 a0, a0
```

Finally, any arbitrary 64-bit constant can be created using the following 5-instruction pattern and one spill register:

```
lui a0, ((v - (v << 52 >> 52)) >> 12) & 0xfffff
addiw a0, a0, (v << 52 >> 52)
lui a1, ((v - (v << 20 >> 20)) >> 44) & 0xfffff
addiw a1, a1, (v << 20 >> 52)
pack a0, a0, a1
```

4.1.2 Bitfield extract

Extracting a bit field of length `len` at position `pos` can be done using two shift operations.

```
slli a0, a0, (XLEN-len-pos)
srli a0, a0, (XLEN-len)
```

Or using `srai` for a signed bit-field.

```
slli a0, a0, (XLEN-len-pos)
srai a0, a0, (XLEN-len)
```

4.1.3 Packing bit-fields

There are different ways of packing bit-fields with the help of RISC-V BitManip instructions.

For example, packing a 16-bit RGB value in 5:6:5 format, from the bytes in a0, a1, and a2, using `pack[h]` and `bext`:

```
li t0, 0x00f8fcf8

packh a0, a0, a1
pack  a0, a0, a2
bext  a0, a0, t0
```

Or using funnel shifts (assuming `a2` is already in zero-extended form and/or the upper bits of the return value do not matter):

```
srli a2, a2, 3
slli a1, a1, XLEN-8
fsli a1, a2, a1, 6
slli a0, a0, XLEN-8
fsli a0, a1, a0, 5
```

Using only base-ISA instructions, at least 7 instructions are needed to pack a 5:6:5 RGB value (assuming `a0` is alredy in zero-extended form):

```
andi a2, a2, 0xf8
slli a2, a2, 9
andi a1, a1, 0xfc
slli a1, a1, 3
srli a0, a0, 3
or a1, a1, a2
or a0, a0, a1
```

Another example for packing bit fields is generating IEEE floats using only integer instructions (aka “soft float”). For example, generating a 32-bit float in the range $[-1.0 \cdots +1.0)$ from a signed 16-bit integer:

```
short2float:
  neg a1, a0
  max a1, a1, a0
  clz a2, a1
  srli a0, a0, 31
  sll a3, a1, a2
  srli a3, a3, 15
  neg a2, a2
  addi a2, a2, 143
  packh a0, a2, a0
  pack a0, a3, a0
  slli a4, a4, 7
  orc a1, a1
  and a0, a0, a1
  ret
```

Or using funnel shifts:

```
short2float:
  neg a1, a0
  max a1, a1, a0
  clz a2, a1
  sll a3, a1, a2
  slli a3, a3, 1
  neg a2, a2
  addi a2, a2, 143
  fsri a3, a3, a2, 8
  srli a0, a0, 31
  fsri a0, a3, a0, 1
  cmov a0, a1, a0, zero
  ret
```

4.1.4 Parity check

The parity of a word (xor of all bits) is the LSB of the population count.

```
cpop a0, a0
andi a0, a0, 1
```

4.1.5 Average of two integers

The following four instructions calculate the average of the unsigned integers in `a0` and `a1`, with compensation for overflow:

```
and  a2, a0, a1
xor  a0, a0, a1
srli a0, a0, 1
add  a0, a0, a2
```

And likewise the average of two signed integers:

```
and  a2, a0, a1
xor  a0, a0, a1
srai a0, a0, 1
add  a0, a0, a2
```

With `fsri` the unsigned case can be accomplished in just three instructions:

```
add  a0, a0, a1
sltu a1, a0, a1
fsri a0, a1, 1
```

4.1.6 Detecting integer overflow

Overflow in unsigned addition can be detected in two instructions:

```
add  a0, a1, a2
bltu a0, a1, overflow
```

For signed addition, if the sign of one operand is known, for example because it is constant:

```
addi a0, a1, +imm
blt  a0, a1, overflow
```

```
addi a0, a1, -imm
bgt  a0, a1, overflow
```

And for signed addition in the general case:

```
add  a0, a1, a2
slti a3, a1, 0
slt  a4, a0, a2
bne  a3, a4, overflow
```

And finally, generating the carry flag for an addition:

```
add  a0, a1, a2
sltu a3, a0, a1
```

Thus, adding `a0`, `a1`, and `a2` with results in `a0` and carry-out in `a1`:

```
add  a0, a0, a1
sltu a1, a0, a1
add  a0, a0, a2
sltu a2, a0, a2
add  a1, a1, a2
```

4.1.7 Fuse-able sequences for logic operations

RISC-V has dedicated instructions for branching on equal/not-equal. But C code such as the following would require set-equal and set-not-equal instructions, similar to `slt`.

```
int is_equal = (a == b);
int is_noteq = (c != d);
```

Those can be implemented using the following fuse-able sequences:

```
sub rd, rs1, rs2
sltui rd, rd, 1

sub rd, rs1, rs2
sltu rd, zero, rd
```

Likewise for logic OR:

```
int logic_or  = (c || d);

or rd, rs1, rs2
sltu rd, zero, rd
```

And for logic AND, if `rd != rs2`:

```
int logic_and  = (c && d);

orc rd, rs1
and rd, rd, rs2
sltu rd, zero, rd
```

4.1.8 Rotate shift of bytes and half-words

Rotate right shift of the byte in `a0` by the shift amount in `a1`, assuming `a0` is stored in zero-extended form:

```
orc8 a0, a0
ror a0, a1
andi a0, a0, 255
```

And rotate right shift of the 16-bit half-word in `a0` by the shift amount in `a1`, assuming `a0` is stored in zero-extended form:

```
orc16 a0, a0
ror a0, a1
pack[w] a0, a0, zero
```

4.1.9 Rank and select

Rank and select are fundamental operations in succinct data structures [20].

`select(a0, a1)` returns the position of the `a1`th set bit in `a0`. It can be implemented efficiently using `bdep` and `ctz`:

```
select:
  bset a1, zero, a1
  bdep a0, a1, a0
  ctz a0, a0
  ret
```

`rank(a0, a1)` returns the number of set bits in `a0` up to and including position `a1`.

```
rank:
  not a1, a1
  sll a0, a1
  cpop a0, a0
  ret
```

4.1.10 OR/AND/XOR-reduce in byte vectors

OR-ing the bytes in a register and returning the resulting byte is easy with GORC:

```
gorci a0, a0, -8
andi a0, 255
```

AND-ing can accomplished by applying De Morgan's laws:

```
not a0, a0
gorci a0, a0, -8
not a0, a0
andi a0, 255
```

XOR-ing can be accomplished with CLMUL (see also section 4.7).

```
andi a1, zero, 0x80
gorci a1, a1, -8
clmulr a0, a0, a1
andi a0, 255
```

Where the first two instructions (andi+gorci) just create the constant 0x8080..8080.

Finally, on RV64, XOR-ing the bytes in a register can also be accomplished with BMATXOR:

```
andi a1, zero, 0xff
bmatxor a0, a1, a0
```

4.1.11 Counting trailing non-zero bytes

Counting the trailing (LSB-end) non-zero bytes in a word is a helpful operation in optimized implementations of `strlen()` and `strcpy()`:

```
int count_trailing_nonzero_bytes(long x)
{
  return _rv_ctz(~_rv_orc_b(x)) >> 3;
}
```

4.1.12 Finding bytes of certain values

Finding zero bytes is a useful operations for `strchr()` and `memchr()`:

```
bool check_zero_bytes(long x)
{
  return ~_rv_orc_b(x) != 0;
}
```

To find other bytes we simply XOR the value with a mask of the byte value we are looking for:

```
bool check_byte(long x, unsigned char c)
{
  return ~_rv_orc_b(x ^ _rv_orc8(c)) != 0;
}
```

These schemes can easily be extended with `ctz` and `cpop` to perform operations such as counting the number of bytes of a certain value within a word, or finding the position of the first such byte.

4.1.13 Fill right of most significant set bit

The "fill right" or "fold right" operation is a pattern commonly used in bit manipulation code. [9]

The straight-forward RV64 implementation requires 12 instructions:

```
uint64_t rfill(uint64_t x)
{
  x |= x >> 1;   // SRLI, OR
  x |= x >> 2;   // SRLI, OR
  x |= x >> 4;   // SRLI, OR
  x |= x >> 8;   // SRLI, OR
  x |= x >> 16;  // SRLI, OR
  x |= x >> 32;  // SRLI, OR
  return x;
}
```

With `clz` it can be implemented in only 4 instructions. Notice the handling of the case where `x=0` using `sltiu+addi`.

```
uint64_t rfill_clz(uint64_t x)
{
  uint64_t t;
  t = clz(x);         // CLZ
  x = (!x)-1;         // SLTIU, ADDI
  x = x >> (t & 63);  // SRL
  return x;
}
```

Alternatively, a Trailing Bit Manipulation (TBM) code pattern can be used together with `rev` to implement this function in 4 instructions:

```
uint64_t rfill_rev(uint64_t x)
{
  x = rev(x);         // GREVI
  x = x | ~(x - 1);   // ADDI, ORN
  x = rev(x);         // GREVI
  return x;
}
```

Finally, there is another implementation in 4 instructions using BMATOR, if we do not count the extra instructions for loading utility matrices.

```
uint64_t rfill_bmat(uint64_t x)
{
  uint64_t m0, m1, m2, t;

  m0 = 0xFF7F3F1F0F070301LL;  // LD
  m1 = bmatflip(m0 << 8);     // SLLI, BMATFLIP
  m2 = -1LL;                  // ADDI

  t = bmator(x, m0);          // BMATOR
  x = bmator(x, m2);          // BMATOR
  x = bmator(m1, x);          // BMATOR
  x |= t;                     // OR

  return x;
}
```

4.1.14 Round to next power of two

One common application of `rfill()` is rounding up to the next power of two:

```
uint64_t round_pow2(uint64_t x)
{
  return rfill(x-1)+1;
}
```

This can also be implemented in just 4 instructions, if we don't care about the case where the above code overflows because `x` is already larger than the largest power-of-two representable in an `uint64_t`.

```
uint64_t round_pow2(uint64_t x)
{
  uint64_t t;
  t = clz(x-1);     // ADDI, CLZ
  x = ror(!!x, t);  // SLTU, ROR
  return x;
}
```

Note that this code handles $0 \rightarrow 0$ and $1 \rightarrow 1$ correctly, i.e. equivialent to `rfill(x-1)+1`.

4.2 Packed vectors

4.2.1 Packing bytes

The following RV32 code packs the lower 8 bits from a0, a1, a2, a3 into a 32-bit word returned in a0, ignoring other bits in the input values.

```
packh a0, a0, a1
packh a1, a2, a3
pack  a0, a0, a1
```

And the following RV64 code packs 8 bytes into a register.

```
packh a0, a0, a1
packh a1, a2, a3
packh a2, a4, a5
packh a3, a6, a7
packw a0, a0, a1
packw a1, a2, a3
pack  a0, a0, a1
```

4.2.2 Permuting bytes

There are 24 ways of arranging the four bytes in a 32-bit word. `ror`, `grev`, and `[un]shfl` can perform any of those permutations in at most 3 instructions. Table 4.1 lists those sequences. [26]

4.2.3 Widening and narrowing

The `[un]zip` instructions can help with widening and narrowing packed vectors. For example, narrowing the bytes in two words into a single word with the values in nibbles with values from `a0` in LSB half and values from `a1` in MSB half:

```
unzip4 a0, a0
unzip4 a1, a1
pack a0, a0, a1
```

And widening the nibbles from `a0` into bytes in `a1` (MSB half) and `a0` (LSB half), with zero extension:

```
srli a1, a0, XLEN/2
pack a0, a0, zero
zip4 a1, a1
zip4 a0, a0
```

And finally the same widening operation with sign extension:

Bytes	Instructions
A B C D	*initial byte order*
A B D C	`ROR(24),SHFL(8),ROR(8)`
A C B D	`SHFL(8)`
A C D B	`ROR(8),GREV(8),SHFL(8)`
A D B C	`ROR(16),SHFL(8),ROR(24)`
A D C B	`ROR(8),GREV(8)`
B A C D	`ROR(8),SHFL(8),ROR(24)`
B A D C	`GREV(8)`
B C A D	`ROR(16),SHFL(8),ROR(8)`
B C D A	`ROR(24)`
B D A C	`GREV(8),SHFL(8)`
B D C A	`ROR(24),SHFL(8)`
C A B D	`ROR(8),GREV(24),SHFL(8)`
C A D B	`ROR(16),SHFL(8)`
C B A D	`ROR(8),GREV(24)`
C B D A	`SHFL(8),ROR(24)`
C D A B	`ROR(16)`
C D B A	`ROR(8),SHFL(8),ROR(8)`
D A B C	`ROR(8)`
D A C B	`SHFL(8),ROR(8)`
D B A C	`ROR(8),SHFL(8)`
D B C A	`GREV(24),SHFL(8)`
D C A B	`ROR(24),SHFL(8),ROR(24)`
D C B A	`GREV(24)`

Table 4.1: Instruction sequences for arbitrary permutations of bytes in a 32-bit word.

```
addi t0, zero, 8
orc4 t0, t0
and t0, t0, a0
orc.n t0, t0
srli t1, t0, XLEN/2
srli a1, a0, XLEN/2
pack a1, a1, t1
pack a0, a0, t0
zip4 a1, a1
zip4 a0, a0
```

4.2.4 Shifting packed vector elements

Using `zip` we can re-arrange the bits in a packed vector of N elements so that a shift by k of each byte becomes a shift of Nk of the entire new vector. So we zip, shift, and then `unzip` to shuffle everything back. The number of `zip` and `unzip` is $\log2(N)$. This works for all kinds of shift operations. For example, rotating a vector of bytes on RV32 in 6 instructions:

```
zip a0, a0
zip a0, a0
slli a1, a1, 2
ror a0, a0, a1
unzip a0, a0
unzip a0, a0
```

Because `zip; zip; zip` is equal to `unzip; unzip` on RV32, and equal to `unzip; unzip; unzip` on RV64, we need never more than 2 `[un]zip` on RV32, or 3 `[un]zip` on RV64.

4.2.5 Adding packed vectors

The following six instructions will add the elements of the two vectors passed in `a0` and `a1`, and return the vector of sums in `a0`.

This expects a mask in `a2` that marks the MSB bit of each vector element. For a vector of bytes this mask would be `0x8080...80` (which can be obtained in two instructions via `orc8(0x80)`).

```
xor  a3, a0, a1
and  a3, a3, a2
andn a0, a0, a2
andn a1, a1, a2
add  a0, a0, a1
xor  a0, a0, a3
```

4.3 Funnel shifts

A funnel shift takes two XLEN registers, concatenates them to a $2 \times$ XLEN word, shifts that by a certain amount, then returns the lower half of the result for a right shift and the upper half of the result for a left shift.

The `fsl`, `fsr`, and `fsri` instructions perform funnel shifts.

4.3.1 Bigint shift

A common application for funnel shifts is shift operations in bigint libraries.

For example, the following functions implement rotate-shift operations for bigints made from `n` XLEN words.

```
void bigint_rol(uint_xlen_t data[], int n, int shamt)
{
  if (n <= 0)
    return;

  uint_xlen_t buffer = data[n-1];
  for (int i = n-1; i > 0; i--)
    data[i] = fsl(data[i], shamt, data[i-1]);
  data[0] = fsl(data[0], shamt, buffer);
}

void bigint_ror(uint_xlen_t data[], int n, int shamt)
{
  if (n <= 0)
    return;

  uint_xlen_t buffer = data[0];
  for (int i = 0; i < n-1; i++)
    data[i] = fsr(data[i], shamt, data[i+1]);
  data[n-1] = fsr(data[n-1], shamt, buffer);
}
```

These version only works for shift-amounts <XLEN. But functions supporting other kinds of shift operations, or shifts ≥XLEN can easily be built with `fsl` and `fsr`.

4.3.2 Parsing bit-streams of 27-bit words

The following function parses `n` 27-bit words from a packed array of XLEN words:

```
void parse_27bit(uint_xlen_t *idata, uint_xlen_t *odata, int n)
{
  uint_xlen_t lower = 0, upper = 0;
  int reserve = 0;

  while (n--) {
    if (reserve < 27) {
      uint_xlen_t buf = *(idata++);
      lower |= sll(buf, reserve);
      upper = reserve ? srl(buf, -reserve) : 0;
      reserve += XLEN;
    }
    *(odata++) = lower & ((1 << 27)-1);
    lower = fsr(lower, 27, upper);
    upper = srl(upper, 27);
    reserve -= 27;
  }
}
```

And here the same thing in RISC-V assembler:

```
parse_27bit:
  li t1, 0              ; lower
  li t2, 0              ; upper
  li t3, 0              ; reserve
  li t4, 27             ; shamt
  slo t5, zero, t4      ; mask
  beqz a2, endloop      ; while (n--)
loop:
  addi a2, a2, -1
  bge t3, t4, output       ; if (reserve < 27)
  lw t6, 0(a0)                 ; buf = *(idata++)
  addi a0, a0, 4
  sll t7, t6, t3               ; lower |= sll(buf, reserve)
  or t1, t1, t7
  sub t7, zero, t3             ; upper = reserve ? srl(buf, -reserve) : 0
  srl t7, t6, t7
  cmov t2, t3, t7, zero
  addi t3, t3, 32              ; reserve += XLEN;
output:
  and t6, t1, t5           ; *(odata++) = lower & ((1 << 27)-1)
  sw t6, 0(a1)
  addi a1, a1, 4
  fsr t1, t1, t2, t4       ; lower = fsr(lower, 27, upper)
  srl t2, t2, t4           ; upper = srl(upper, 27)
  sub t3, t3, t4           ; reserve -= 27
  bnez a2, loop         ; while (n--)
endloop:
  ret
```

A loop iteration without fetch is 9 instructions long, and a loop iteration with fetch is 17 instructions long.

Without ternary operators that would be 13 instructions and 22 instructions, i.e. assuming one cycle per instruction, that function would be about 30% slower without ternary instructions.

4.3.3 Parsing bit-streams of 6-bit words

The following code accepts three 64-bit words in `t0`, `t1`, and `t2` containing a bit-stream of 32 6-bit words, and outputs these 32 values in the bytes of `t0`, `t1`, `t2`, and `t3`.

```
addi a0, zer0, 0x3f
orc8 a0, a0          // a0 = 0x3f3f3f3f3f3f3f3f

srli t3, t2, 16
bdep t3, t3, a0
fsli t2, t2, t1, 32
bdep t2, t2, a0
fsli t1, t1, t0, 16
bdep t1, t1, a0
bdep t0, t0, a0
```

That's 7 instructions without the two instructions for constructing the mask in `a0`.

Without funnel shift this operation requires 11 instructions:

```
addi a0, zer0, 0x3f
orc8 a0, a0          // a0 = 0x3f3f3f3f3f3f3f3f

srli t3, t2, 16
bdep t3, t3, a0
slli t2, t2, 32
srli a1, t1, 32
or t2, t2, a1
bdep t2, t2, a0
slli t1, t1, 16
srli a1, t0, 48
or t1, t1, a1
bdep t1, t1, a0
bdep t0, t0, a0
```

Sign-extending the 6-bit values in the bytes of `t0`, `t1`, `t2`, and `t3` with bit-matrix multiply:

```
li a0, 0x80c0e01008040201

bmator t0, t0, a0
bmator t1, t1, a0
bmator t2, t2, a0
bmator t3, t3, a0
```

Or without bit-matrix multiply:

```
    addi a0, zer0, 0x60
    orc8 a0, a0          // a0 = 0x6060606060606060

    add t0, t0, a0
    add t1, t1, a0
    add t2, t2, a0
    add t3, t3, a0
    xor t0, t0, a0
    xor t1, t1, a0
    xor t2, t2, a0
    xor t3, t3, a0
```

4.3.4 Fixed-point multiply

A fixed-point multiply is simply an integer multiply, followed by a right shift. If the entire dynamic range of XLEN bits should be useable for the factors, then the product before shift must be 2*XLEN wide. Therefore `mul`+`mulh` is needed for the multiplication, and funnel shift instructions can help with the final right shift. For fixed-point numbers with N fraction bits:

```
  mul_fracN:
    mulh a2, a0, a1
    mul a0, a0, a1
    fsri a0, a0, a2, N
    ret
```

4.4 Arbitrary bit permutations

This section lists code snippets for computing arbitrary bit permutations that are defined by data (as opposed to bit permutations that are known at compile time and can likely be compiled into shift-and-mask operations and/or a few instances of bext/bdep).

4.4.1 Using butterfly operations

The following macro performs a stage-`N` butterfly operation on the word in `a0` using the mask in `a1`.

```
  grevi a2, a0, (1 << N)
  cmix a0, a1, a2, a0
```

The bitmask in `a1` must be preformatted correctly for the selected butterfly stage. A butterfly operation only has a XLEN/2 wide control word. The following macros format the mask assuming those XLEN/2 bits in the lower half of `a1` on entry:

```
bfly_msk_0:
  pack a1, a1, a1
  zip a1, a1

bfly_msk_1:
  pack a1, a1, a1
  zip2 a1, a1

bfly_msk_2:
  pack a1, a1, a1
  zip4 a1, a1

...
```

A sequence of $2 \cdot log_2(\text{XLEN}) - 1$ butterfly operations can perform any arbitrary bit permutation (Beneš network):

```
  butterfly(LOG2_XLEN-1)
  butterfly(LOG2_XLEN-2)
  ...
  butterfly(0)
  ...
  butterfly(LOG2_XLEN-2)
  butterfly(LOG2_XLEN-1)
```

Many permutations arising from real-world applications can be implemented using shorter sequences. For example, any sheep-and-goats operation (SAG, see section 4.4.4) with either the sheep or the goats bit reversed can be implemented in $log_2(\text{XLEN})$ butterfly operations.

Reversing a permutation implemented using butterfly operations is as simple as reversing the order of butterfly operations.

4.4.2 Using omega-flip networks

The omega operation is a stage-0 butterfly preceded by a zip operation:

```
  zip a0, a0
  grevi a2, a0, 1
  cmix a0, a1, a2, a0
```

The flip operation is a stage-0 butterfly followed by an unzip operation:

```
  grevi a2, a0, 1
  cmix a0, a1, a2, a0
  unzip a0, a0
```

A sequence of log_2(XLEN) omega operations followed by log_2(XLEN) flip operations can implement any arbitrary 32 bit permutation.

As for butterfly networks, permutations arising from real-world applications can often be implemented using a shorter sequence.

4.4.3 Using baseline networks

Another way of implementing arbitrary 32 bit permutations is using a baseline network followed by an inverse baseline network.

A baseline network is a sequence of log_2(XLEN) butterfly(0) operations interleaved with unzip operations. For example, a 32-bit baseline network:

```
butterfly(0)
unzip
butterfly(0)
unzip.h
butterfly(0)
unzip.b
butterfly(0)
unzip.n
butterfly(0)
```

An inverse baseline network is a sequence of log_2(XLEN) butterfly(0) operations interleaved with zip operations. The order is opposite to the order in a baseline network. For example, a 32-bit inverse baseline network:

```
butterfly(0)
zip.n
butterfly(0)
zip.b
butterfly(0)
zip.h
butterfly(0)
zip
butterfly(0)
```

A baseline network followed by an inverse baseline network can implement any arbitrary bit permutation.

4.4.4 Using sheep-and-goats

The Sheep-and-goats (SAG) operation is a common operation for bit permutations. It moves all the bits selected by a mask (goats) to the LSB end of the word and all the remaining bits (sheep) to the MSB end of the word, without changing the order of sheep or goats.

The SAG operation can easily be performed using `bext` (data in `a0` and mask in `a1`):

```
bext a2, a0, a1
not a1, a1
bext a0, a0, a1
cpop a1, a1
ror a0, a0, a1
or a0, a0, a2
```

Any arbitrary bit permutation can be implemented in log_2(XLEN) SAG operations.

The Hacker's Delight describes an optimized standard C implementation of the SAG operation. Their algorithm takes 254 instructions (for 32 bit) or 340 instructions (for 64 bit) on their reference RISC instruction set. [10, p. 152f, 162f]

4.4.5 Using bit-matrix multiply

`bat[x]or` performs a permutation of bits within each byte when used with a permutation matrix in `rs2`, and performs a permutation of bytes when used with a permutation matrix in `rs1`.

4.5 Mirroring and rotating bitboards

Bitboards are 64-bit bitmasks that are used to represent part of the game state in chess engines (and other board game AIs). The bits in the bitmask correspond to squares on a 8×8 chess board:

```
56 57 58 59 60 61 62 63
48 49 50 51 52 53 54 55
40 41 42 43 44 45 46 47
32 33 34 35 36 37 38 39
24 25 26 27 28 29 30 31
16 17 18 19 20 21 22 23
 8  9 10 11 12 13 14 15
 0  1  2  3  4  5  6  7
```

Many bitboard operations are simple straight-forward operations such as bitwise-AND, but mirroring and rotating bitboards can take up to 20 instructions on x86.

4.5.1 Mirroring bitboards

Flipping horizontally or vertically can easily done with `grevi`:

```
Flip horizontal:
 63 62 61 60 59 58 57 56    RISC-V Bitmanip:
 55 54 53 52 51 50 49 48       rev.b
 47 46 45 44 43 42 41 40
 39 38 37 36 35 34 33 32
 31 30 29 28 27 26 25 24    x86:
 23 22 21 20 19 18 17 16       13 operations
 15 14 13 12 11 10  9  8
  7  6  5  4  3  2  1  0

Flip vertical:
  0  1  2  3  4  5  6  7    RISC-V Bitmanip:
  8  9 10 11 12 13 14 15       rev8
 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31
 32 33 34 35 36 37 38 39    x86:
 40 41 42 43 44 45 46 47       bswap
 48 49 50 51 52 53 54 55
 56 57 58 59 60 61 62 63
```

Rotating by 180 (flip horizontal and vertical):

```
Rotate 180:
  7  6  5  4  3  2  1  0    RISC-V Bitmanip:
 15 14 13 12 11 10  9  8       rev
 23 22 21 20 19 18 17 16
 31 30 29 28 27 26 25 24
 39 38 37 36 35 34 33 32    x86:
 47 46 45 44 43 42 41 40       14 operations
 55 54 53 52 51 50 49 48
 63 62 61 60 59 58 57 56
```

4.5.2 Rotating bitboards

Using `zip` a bitboard can be transposed easily:

```
Transpose:
  7 15 23 31 39 47 55 63    RISC-V Bitmanip:
  6 14 22 30 38 46 54 62       zip, zip, zip
  5 13 21 29 37 45 53 61
  4 12 20 28 36 44 52 60
  3 11 19 27 35 43 51 59    x86:
  2 10 18 26 34 42 50 58       18 operations
  1  9 17 25 33 41 49 57
  0  8 16 24 32 40 48 56
```

A rotation is simply the composition of a flip operation and a transpose operation. This takes 19 operations on x86 [2]. With Bitmanip the rotate operation only takes 4 operations:

```
rotate_bitboard:
  rev8 a0, a0
  zip a0, a0
  zip a0, a0
  zip a0, a0
```

4.5.3 Explanation

The bit indices for a 64-bit word are 6 bits wide. Let `i[5:0]` be the index of a bit in the input, and let `i′[5:0]` be the index of the same bit after the permutation.

As an example, a rotate left shift by N can be expressed using this notation as `i′[5:0]` $=$ `i[5:0]` $+ N \pmod{64}$.

The GREV operation with shamt N is `i′[5:0]` $=$ `i[5:0]` XOR N.

And a SHFL operation corresponds to a rotate left shift by one position of any contiguous region of `i[5:0]`. For example, `zip` is a left rotate shift of the entire bit index:

$$\texttt{i}'\texttt{[5:0]} = \{\texttt{i[4:0]}, \texttt{i[5]}\}$$

And `zip4` performs a left rotate shift on bits `5:2`:

$$\texttt{i}'\texttt{[5:0]} = \{\texttt{i[4:2]}, \texttt{i[5]}, \texttt{i[1:0]}\}$$

In a bitboard, `i[2:0]` corresponds to the X coordinate of a board position, and `i[5:3]` corresponds to the Y coordinate.

Therefore flipping the board horizontally is the same as negating bits `i[2:0]`, which is the operation performed by `grevi rd, rs, 7` (`rev.b`).

Likewise flipping the board vertically is done by `grevi rd, rs, 56` (`rev8`).

Finally, transposing corresponds by swapping the lower and upper half of `i[5:0]`, or rotate shifting `i[5:0]` by 3 positions. This can easily done by rotate shifting the entire `i[5:0]` by one bit position (`zip`) three times.

4.5.4 Rotating Bitcubes

Let's define a bitcube as a $4 \times 4 \times 4$ cube with $x =$ `i[1:0]`, $y =$ `i[3:2]`, and $z =$ `i[5:4]`. Using the same methods as described above we can easily rotate a bitcube by 90° around the X-, Y-, and Z-axis:

```
rotate_x:                   rotate_y:                   rotate_z:
  rev16 a0, a0                rev.n a0, a0                rev4.h
  zip4 a0, a0                 zip a0, a0                  zip.h a0, a0
  zip4 a0, a0                 zip a0, a0                  zip.h a0, a0
                              zip4 a0, a0
                              zip4 a0, a0
```

4.6 Manipulating 64x64 Bit Matrices

The `bmat[x]or` and `bmatflip` instructions operate on 8x8 bit matrices stored in single 64-bit registers, where each byte of such a 64-bit value represents one row (column) of a 8x8 bit matrix.

Let's assume we have a 64x64 bit matrix in memory, stored as one row (column) per 64-bit value. In order to use `bmat[x]or` and `bmatflip` on such a matrix, we must first convert it into a 8x8 block matrix of 64 individual 8x8 matrices, each stored in a 64-bit value. The following function performs this transformation for a single row (column) of the block matrix in 40 instructions.

```
void conv8x8(const uint64_t x[8], uint64_t y[8])
{
  uint64_t x0_x1_31_00 = _rv64_pack (x[0], x[1]);
  uint64_t x2_x3_31_00 = _rv64_pack (x[2], x[3]);
  uint64_t x4_x5_31_00 = _rv64_pack (x[4], x[5]);
  uint64_t x6_x7_31_00 = _rv64_pack (x[6], x[7]);
  uint64_t x0_x1_63_32 = _rv64_packu(x[0], x[1]);
  uint64_t x2_x3_63_32 = _rv64_packu(x[2], x[3]);
  uint64_t x4_x5_63_32 = _rv64_packu(x[4], x[5]);
  uint64_t x6_x7_63_32 = _rv64_packu(x[6], x[7]);

  uint64_t x0_x1_31_00_z = _rv64_unzip16(x0_x1_31_00);
  uint64_t x2_x3_31_00_z = _rv64_unzip16(x2_x3_31_00);
  uint64_t x4_x5_31_00_z = _rv64_unzip16(x4_x5_31_00);
  uint64_t x6_x7_31_00_z = _rv64_unzip16(x6_x7_31_00);
  uint64_t x0_x1_63_32_z = _rv64_unzip16(x0_x1_63_32);
  uint64_t x2_x3_63_32_z = _rv64_unzip16(x2_x3_63_32);
  uint64_t x4_x5_63_32_z = _rv64_unzip16(x4_x5_63_32);
  uint64_t x6_x7_63_32_z = _rv64_unzip16(x6_x7_63_32);

  uint64_t x0_x1_x2_x3_15_00 = _rv64_pack (x0_x1_31_00_z, x2_x3_31_00_z);
  uint64_t x4_x5_x6_x7_15_00 = _rv64_pack (x4_x5_31_00_z, x6_x7_31_00_z);
  uint64_t x0_x1_x2_x3_31_16 = _rv64_packu(x0_x1_31_00_z, x2_x3_31_00_z);
  uint64_t x4_x5_x6_x7_31_16 = _rv64_packu(x4_x5_31_00_z, x6_x7_31_00_z);
  uint64_t x0_x1_x2_x3_47_32 = _rv64_pack (x0_x1_63_32_z, x2_x3_63_32_z);
  uint64_t x4_x5_x6_x7_47_32 = _rv64_pack (x4_x5_63_32_z, x6_x7_63_32_z);
  uint64_t x0_x1_x2_x3_63_48 = _rv64_packu(x0_x1_63_32_z, x2_x3_63_32_z);
  uint64_t x4_x5_x6_x7_63_48 = _rv64_packu(x4_x5_63_32_z, x6_x7_63_32_z);
  uint64_t x0_x1_x2_x3_15_00_z = _rv64_unzip8(x0_x1_x2_x3_15_00);
  uint64_t x4_x5_x6_x7_15_00_z = _rv64_unzip8(x4_x5_x6_x7_15_00);
  uint64_t x0_x1_x2_x3_31_16_z = _rv64_unzip8(x0_x1_x2_x3_31_16);
  uint64_t x4_x5_x6_x7_31_16_z = _rv64_unzip8(x4_x5_x6_x7_31_16);
  uint64_t x0_x1_x2_x3_47_32_z = _rv64_unzip8(x0_x1_x2_x3_47_32);
  uint64_t x4_x5_x6_x7_47_32_z = _rv64_unzip8(x4_x5_x6_x7_47_32);
  uint64_t x0_x1_x2_x3_63_48_z = _rv64_unzip8(x0_x1_x2_x3_63_48);
  uint64_t x4_x5_x6_x7_63_48_z = _rv64_unzip8(x4_x5_x6_x7_63_48);

  y[0] = _rv64_pack (x0_x1_x2_x3_15_00_z, x4_x5_x6_x7_15_00_z);
  y[1] = _rv64_packu(x0_x1_x2_x3_15_00_z, x4_x5_x6_x7_15_00_z);
  y[2] = _rv64_pack (x0_x1_x2_x3_31_16_z, x4_x5_x6_x7_31_16_z);
  y[3] = _rv64_packu(x0_x1_x2_x3_31_16_z, x4_x5_x6_x7_31_16_z);
  y[4] = _rv64_pack (x0_x1_x2_x3_47_32_z, x4_x5_x6_x7_47_32_z);
  y[5] = _rv64_packu(x0_x1_x2_x3_47_32_z, x4_x5_x6_x7_47_32_z);
  y[6] = _rv64_pack (x0_x1_x2_x3_63_48_z, x4_x5_x6_x7_63_48_z);
  y[7] = _rv64_packu(x0_x1_x2_x3_63_48_z, x4_x5_x6_x7_63_48_z);
}
```

Each of the 5 blocks in this function only consumes the eight outputs of the previous block. Therefore 16 registers are sufficient to run this function in registers only without the need to spill any data on the stack.

Note that this function is its own inverse. Therefore the same function can be used for the convertion from block matrix form back to row (column) major form.

A bit 64x64 bit matrix in block matrix form can easily be transposed by running `bmatflip` (or `zip; zip; zip`) on the blocks of the matrix and then renaming the individual 64-bit variables.

To multiply 64x64 bit matrices in block matrix form, the matrix-matrix-product is decomposed in the obvious way in $8 \times 8 \times 8 = 512$ `bmat[x]or` instructions and $7 \times 8 \times 8 = 448$ `[x]or` instructions.

4.7 Inverting Xorshift RNGs

Xorshift RNGs are a class of fast RNGs for different bit widths. There are 648 Xorshift RNGs for 32 bits, but this is the one that the author of the original Xorshift RNG paper recommends. [19, p. 4]

```
  uint32_t xorshift32(uint32_t x)
  {
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    return x;
  }
```

This function of course has been designed and selected so it's efficient, even without special bit-manipulation instructions. So let's look at the inverse instead. First, the naïve form of inverting this function:

```
uint32_t xorshift32_inv(uint32_t x)
{
  uint32_t t;
  t = x ^ (x << 5);
  t = x ^ (t << 5);
  t = x ^ (t << 5);
  t = x ^ (t << 5);
  t = x ^ (t << 5);
  x = x ^ (t << 5);
  x = x ^ (x >> 17);
  t = x ^ (x << 13);
  x = x ^ (t << 13);
  return x;
}
```

This translates to 18 RISC-V instructions, not including the function call overhead.

Obviously the C expression `x ^ (x >> 17)` is already its own inverse (because $17 \geq XLEN/2$) and therefore already has an effecient inverse. But the two other blocks can easily be implemented using a single `clmul` instruction each:

```
uint32_t xorshift32_inv(uint32_t x)
{
  x = clmul(x, 0x42108421);
  x = x ^ (x >> 17);
  x = clmul(x, 0x04002001);
  return x;
}
```

This are 8 RISC-V instructions, including 4 instructions for loading the constants, but not including the function call overhead.

An optimizing compiler could easily generate the clmul instructions and the magic constants from the C code for the naïve implementation. (`0x04002001 = (1 << 2*13) | (1 << 13) | 1` and `0x42108421 = (1 << 6*5) | (1 << 5*5) | ...| (1 << 5) | 1`)

The obvious remaining question is "if `clmul(x, 0x42108421)` is the inverse of `x ^ (x << 5)`, what's the inverse of `x ^ (x >> 5)`?" It's `clmulr(x, 0x84210842)`, where `0x84210842` is the bit-reversal of `0x42108421`.

A special case of xorshift is `x ^ (x >> 1)`, which is a gray encoder. The corresponding gray decoder is `clmulr(x, 0xffffffff)`.

4.8 Cyclic redundency checks (CRC)

There are special instructions for performing CRCs using the two most widespread 32-bit CRC polynomials, CRC-32 and CRC-32C.

CRCs with other polynomials can be computed efficiently using CLMUL. The following examples are using CRC32Q.

The easiest way of implementing CRC32Q with clmul is using a Barrett reduction. On RV32:

```
uint32_t crc32q_simple(const uint32_t *data, int length)
{
  uint32_t P  = 0x814141AB;  // CRC polynomial (implicit x^32)
  uint32_t mu = 0xFEFF7F62;  // x^64 divided by CRC polynomial
  uint32_t mu1 = 0xFF7FBFB1; // "mu" with leading 1, shifted right by 1 bit
  uint32_t crc = 0;

  for (int i = 0; i < length; i++) {
    crc ^= rev8(data[i]);
    crc = clmulr(crc, mu1);
    crc = clmul(crc, P);
  }

  return crc;
}
```

The following python code calculates the value of `mu` for a given CRC polynomial:

```
def polydiv(dividend, divisor):
    quotient = 0
    while dividend.bit_length() >= divisor.bit_length():
        i = dividend.bit_length() - divisor.bit_length()
        dividend = dividend ^ (divisor << i)
        quotient |= 1 << i
    return quotient

P = 0x1814141AB
print("0x%X" % (polydiv(1<<64, P)))   # prints 0x1FEFF7F62
```

A more efficient method would be the following, which processes 64-bit at a time (RV64):

```
uint32_t crc32q_fast(const uint64_t *p, int len)
{
  uint64_t P  = 0x1814141ABLL;   // CRC polynomial
  uint64_t k1 =  0xA1FA6BECLL;   // remainder of x^128 divided by CRC polynomial
  uint64_t k2 =  0x9BE9878FLL;   // remainder of x^96 divided by CRC polynomial
  uint64_t k3 =  0xB1EFC5F6LL;   // remainder of x^64 divided by CRC polynomial
  uint64_t mu = 0x1FEFF7F62LL;   // x^64 divided by CRC polynomial

  uint64_t a0, a1, a2, t1, t2;

  assert(len >= 2);
  a0 = rev8(p[0]);
  a1 = rev8(p[1]);
```

```
    // Main loop: Reduce to 2x 64 bits

    for (const uint64_t *t0 = p+2; t0 != p+len; t0++)
    {
      a2 = rev8(*t0);
      t1 = clmulh(a0, k1);
      t2 = clmul(a0, k1);
      a0 = a1 ^ t1;
      a1 = a2 ^ t2;
    }

    // Reduce to 64 bit, add 32 bit zero padding

    t1 = clmulh(a0, k2);
    t2 = clmul(a0, k2);

    a0 = (a1 >> 32) ^ t1;
    a1 = (a1 << 32) ^ t2;

    t2 = clmul(a0, k3);
    a1 = a1 ^ t2;

    // Barrett Reduction

    t1 = clmul(a1 >> 32, mu);
    t2 = clmul(t1 >> 32, P);
    a0 = a1 ^ t2;

    return a0;
  }
```

The main idea is to transform an array of arbitrary length to an array with the same CRC that's only two 64-bit elements long. (That's the "Main loop" portion of above code.)

Then we further reduce it to just 64-bit. And then we use a Barrett reduction to get the final 32-bit result.

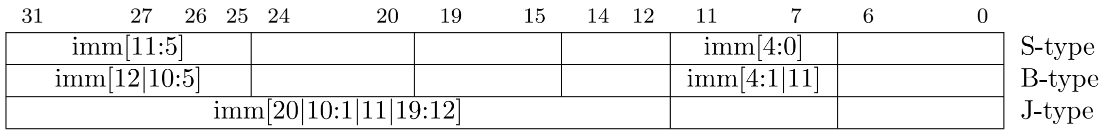
The following python code can be used to calculate the "magic constants" `k1`, `k2`, and `k3`:

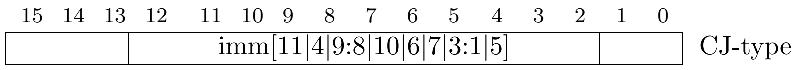
```
def polymod(dividend, divisor):
    quotient = 0
    while dividend.bit_length() >= divisor.bit_length():
        i = dividend.bit_length() - divisor.bit_length()
        dividend = dividend ^ (divisor << i)
        quotient |= 1 << i
    return dividend

print("0x%X" % (polymod(1<<128, P)))   # prints 0xA1FA6BEC
print("0x%X" % (polymod(1<< 96, P)))   # prints 0x9BE9878F
print("0x%X" % (polymod(1<< 64, P)))   # prints 0xB1EFC5F6
```

The above example code is taken from [28]. A more detailed description of the algorithms employed can be found in [12].

4.9 Decoding RISC-V Immediates

The following code snippets decode and sign-extend the immediate from RISC-V S-type, B-type, J-type, and CJ-type instructions. They are nice “nothing up my sleeve”-examples for real-world bit permutations.





```
decode_s:
  li t0, 0xfe000f80
  bext a0, a0, t0
  c.slli a0, 20
  c.srai a0, 20
  ret

decode_b:
  li t0, 0xeaa800aa
  rori a0, a0, 8
  grevi a0, a0, 8
  shfli a0, a0, 7
  bext a0, a0, t0
  c.slli a0, 20
  c.srai a0, 19
  ret

decode_j:
  li t0, 0x800003ff
  li t1, 0x800ff000
  bext a1, a0, t1
  c.slli a1, 23
  rori a0, a0, 21
  bext a0, a0, t0
  c.slli a0, 12
  c.or a0, a1
  c.srai a0, 11
  ret
```

```
// variant 1 (with RISC-V Bitmanip)
decode_cj:
  li t0, 0x28800001
  li t1, 0x000016b8
  li t2, 0xb4e00000
  li t3, 0x4b000000
  bext a1, a0, t1
  bdep a1, a1, t2
  rori a0, a0, 11
  bext a0, a0, t0
  bdep a0, a0, t3
  c.or a0, a1
  c.srai a0, 20
  ret
```

```
// variant 2 (without RISC-V Bitmanip)
decode_cj:
  srli a5, a0, 2
  srli a4, a0, 7
  c.andi a4, 16
  slli a3, a0, 3
  c.andi a5, 14
  c.add a5, a4
  andi a3, a3, 32
  srli a4, a0, 1
  c.add a5, a3
  andi a4, a4, 64
  slli a2, a0, 1
  c.add a5, a4
  andi a2, a2, 128
  srli a3, a0, 1
  slli a4, a0, 19
  c.add a5, a2
  andi a3, a3, 768
  c.slli a0, 2
  c.add a5, a3
  andi a0, a0, 1024
  c.srai a4, 31
  c.add a5, a0
  slli a0, a4, 11
  c.add a0, a5
  ret
```

```
// variant 3 (with RISC-V Zbp only)
decode_cj:
  shfli   a0, a0, 15
  rori    a0, a0, 28
  shfli   a0, a0, 2
  shfli   a0, a0, 14
  rori    a0, a0, 26
  shfli   a0, a0, 8
  rori    a0, a0, 10
  unshfli a0, a0, 12
  rori    a0, a0, 18
  unshfli a0, a0, 14
  rori    a0, a0, 28
  shfli   a0, a0, 6
  rori    a0, a0, 28
  unshfli a0, a0, 15
  slli    a0, a0, 21
  srai    a0, a0, 20
  ret
```

Change History

Date	Rev	Changes
2017-07-17	0.10	Initial Draft
2017-11-02	0.11	Remove roli, assembler can convert it to use a rori
		Remove bitwise subset and replace with `andc`
		Doc source text same base for study and spec.
		Fix typos
2017-11-30	0.32	Jump rev number to be on par with associated Study
		Move pdep/pext into spec draft and called it scatter-gather
2018-04-07	0.33	Move to github, throw out study, convert from .md to .tex
		Fix typos and fix some reference C implementations
		Rename bgat/bsca to bext/bdep
		Remove post-add immediate from clz
		Clean up encoding tables and code sections
2018-04-20	0.34	Add GREV, CTZ, and compressed instructions
		Restructure document: Move discussions to extra sections
		Add FAQ, add analysis of used encoding space
		Add Pseudo-Ops, Macros, Algorithms
		Add Generalized Bit Permutations (shuffle)
2018-05-12	0.35	Replace `shuffle` with generalized zip (`gzip`)
		Add additional XBitfield ISA Extension
		Add figures and tables, Clean up document
		Extend discussion and evaluation chapters
		Add Verilog reference implementations
		Add fast C reference implementations

Date	Rev	Changes
2018-10-05	0.36	XBitfield is now a proper extension proposal
		Add `bswaps.[hwd]` instructions
		Add `cmix`, `cmov`, `fsl`, `fsr`
		Rename `gzip` to `shfl`/`unshfl`
		Add `min`, `max`, `minu`, `maxu`
		Add `clri`, `maki`, `join`
		Add `cseln`, `cselz`, `mvnez`, `mveqz`
		Add `clmul`, `clmulh`, `bmatxor`, `bmator`, `bmatflip`
		Remove `bswaps.[hwd]`, `clri`, `maki`, `join`
		Remove `cseln`, `cselz`, `mvnez`, `mveqz`
2019-06-10	0.90	Add dedicated CRC instructions
		Add proposed opcode encodings
		Rename from XBitmanip to RISC-V Bitmanip
		Remove chapter on `bfxp[c]` instruction
		Refactor proposal into one big chapter
		Remove `c.brev` and `c.neg` instructions
		Add `fsri`, `pack`, `addiwu`, `slliu.w`
		Add `addwu`, `subwu`, `addu.w`, `subu.w`
		Rename `andc` to `andn`, Add `orn` and `xnor`
		Add `sbset[i]`, `sbclr[i]`, `sbinv[i]`, `sbext[i]`
		New naming scheme for `grevi` pseudo-ops
		Add `clmulr` instruction (reversed clmul)
		Jump to Rev 0.90 to indicate spec matureness
2019-08-29	0.91	Change encodings of `bmatxor` and `grev[i][w]`
		Add `gorc[i][w]` and `bfp[w]` instructions
2019-11-08	0.92	Add `packh` and `packu[w]` instructions
		Add `sext.b` and `sext.h` instructions
		Change encoding and behavior of `bfp[w]`
		Change encoding of `bdep[w]`
????-??-??	0.93	Add `sh[123]add` and `sh[123]add.uw`
		Move `slo[i]` and `sro[i]` to "Zbp"
		Add `xperm.[nbhw]`
		Rename *`u.w` instructions to *`.uw`
		Rename `sb`* instructions to `b`*
		Rename `pcnt`* instructions to `cpop`*
????-??-??	0.94	Remove `bset[i]w`, `bclr[i]w`, `binv[i]w`, `bextw`
		Rename `bext`/`bdep` to `bcompress`/`bdecompress`

Bibliography

[1] Apx/aux (pack/unpack) instructions on besm-6 mainframe computers. `http://www.mailcom.com/besm6/instset.shtml#pack`. Accessed: 2019-05-06.

[2] Chess programming wiki, flipping mirroring and rotating. `https://chessprogramming.wikispaces.com/Flipping%20Mirroring%20and%20Rotating`. Accessed: 2017-05-05.

[3] Risc-v sw dev mailing list discussion on fast misaligned load/store. `https://groups.google.com/a/groups.riscv.org/d/topic/sw-dev/DtxGqTBletI/discussion`.

[4] *MC88110 Second Generation RISC Microprocessor User's Manual.* Motorola Inc., 1991.

[5] *Cray Assembly Language (CAL) for Cray X1 Systems Reference Manual.* Cray Inc., 2003. Version 1.1, S-2314-50.

[6] *Cray XMT Principles of Operation.* Cray Inc., 2009. Version 1.3, S-2473-13.

[7] *SPARC T3 Supplement to the UltraSPARC Architecture 2007 Specification.* Oracle, 2010.

[8] *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide (Rev. J).* Texas Instruments, 2010.

[9] The Aggregate. The aggregate magic algorithms. `http://aggregate.org/MAGIC/`. Accessed: 2019-05-26.

[10] Sean Eron Anderson. Bit twiddling hacks. `http://graphics.stanford.edu/~seander/bithacks.html`. Accessed: 2017-04-24.

[11] Armin Biere. private communication, October 2018.

[12] Vinodh Gopal, Erdinc Ozturk, Jim Guilford, Gil Wolrich, Wajdi Feghali, Martin Dixon, and Deniz Karakoyunlu. Fast crc computation for generic polynomials using pclmulqdq instruction. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/fast-crc-computation-generic-polynomials-pclmulqdq-paper.pdf`, 2009. Intel White Paper, Accessed: 2018-10-23.

[13] Y. Hilewitz, C. Lauradoux, and R. B. Lee. Bit matrix multiplication in commodity processors. In *2008 International Conference on Application-Specific Systems, Architectures and Processors*, pages 7–12, July 2008.

[14] Yedidya Hilewitz and Ruby B. Lee. Fast bit compression and expansion with parallel extract and parallel deposit instructions. In *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors*, ASAP '06, pages 65–72, Washington, DC, USA, 2006. IEEE Computer Society.

[15] James Hughes. Using carry-less multiplication (clmul) to implement erasure code. Patent US13866453, 2013.

[16] Donald E. Knuth. *The Art of Computer Programming, Volume 4A*. Addison-Wesley, 2011.

[17] Geoff Langdale and Daniel Lemire. Parsing gigabytes of JSON per second. *CoRR*, abs/1902.08318, 2019.

[18] Daniel Lemire and Owen Kaser. Faster 64-bit universal hashing using carry-less multiplications. *CoRR*, abs/1503.03465, 2015.

[19] George Marsaglia. Xorshift rngs. *Journal of Statistical Software, Articles*, 8(14):1–6, 2003.

[20] Prashant Pandey, Michael A. Bender, and Rob Johnson. A fast x86 implementation of select. *CoRR*, abs/1706.00990, 2017.

[21] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

[22] Wikipedia. Carry-less product. `https://en.wikipedia.org/wiki/Carry-less_product`. Accessed: 2018-10-05.

[23] Wikipedia. Hamming weight. `https://en.wikipedia.org/wiki/Hamming_weight`. Accessed: 2017-04-24.

[24] Wikipedia. Morton code (z-order curve, lebesgue curve). `https://en.wikipedia.org/wiki/Z-order_curve`. Accessed: 2018-10-12.

[25] Claire Wolf. Application examples for xperm instructions. `http://svn.clairexen.net/handicraft/2020/lut4perm`. Accessed: 2019-09-07.

[26] Claire Wolf. Brute-force enumeration of bit permutations reachable with rot/grev/[un]shift. `http://svn.clairexen.net/handicraft/2019/permexplore`. Accessed: 2019-09-07.

[27] Claire Wolf. Reference hardware implementations of bit extract/deposit instructions. `https://github.com/cliffordwolf/bextdep`. Accessed: 2017-04-30.

[28] Claire Wolf. Reference implementations of various crcs using carry-less multiply. `http://svn.clairexen.net/handicraft/2018/clmulcrc/`. Accessed: 2018-11-06.

[29] Claire Wolf. A simple synthetic compiler benchmark for bit manipulation operations. `http://svn.clairexen.net/handicraft/2017/bitcode/`. Accessed: 2017-04-30.