RISC-V Bitmanip Extension

Document Version 0.37-draft

Editor: Clifford Wolf
Symbiotic GmbH
clifford@symbioticeda.com

May 29, 2019

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): Jacob Bachmeyer, Allen Baum, Steven Braeger, Rogier Brussee, Michael Clark, Ken Dockser, Dennis Ferguson, Fabian Giesen, Robert Henry, Bruce Hoult, Po-wei Huang, Rex McCrary, Jiří Moravec, Samuel Neves, Markus Oberhumer, Nils Pipenbrinck, Tommy Thorn, Andrew Waterman, and Clifford Wolf.

Contents

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

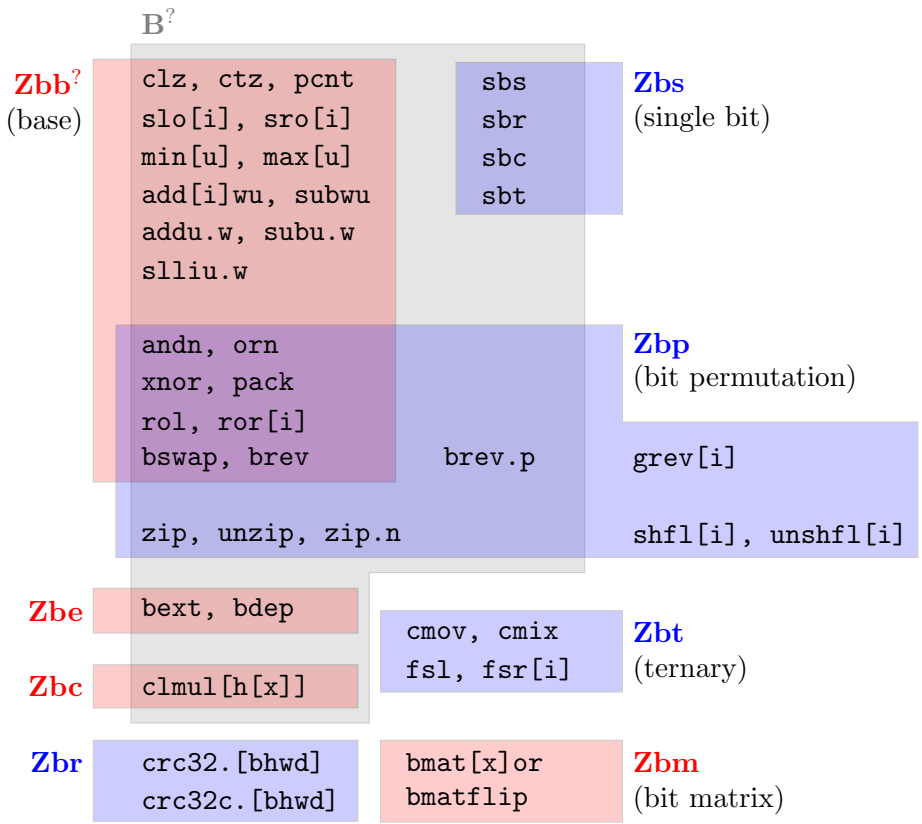
- Assign concrete instruction encodings so that we can start implementing the extension in processor cores and compilers.
- Add support for this extension to processor cores and compilers so we can run quantitative evaluations on the instructions.
- Create assembler snippets for common operations that 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. See Section 2.12 for fast C code for use in emulators.

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 this:



The main open questions of course relate to what should and shouldn't be included in "B", and what should or shouldn't be included in "Zbb". These decisions will be informed in big part by evaluations of the cost and added value for the individual instructions.

With regard to "B", the main open questions are:

- Should `clmul[h]` be included, or `crc32.[bhwd]/crc32c.[bhwd]`, or neither, or both?
- Should Zbt be included? It adds a large architectural cost by adding instructions with three source operands.
- Should Zbe be included? Should Zbm be included?
- Which Zbp pseudo-ops should be included in "B"?

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;
}
```

2.1.2 Count Bits Set (pcnt)

RISC-V Bitmanip ISA

```
RV32, RV64:
  pcnt rd, rs

RV64 only:
  pcntw 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. [18, 16]

```
uint_xlen_t pcnt(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 XLEN/2 words in one register (pack)

RISC-V Bitmanip ISA

```
RV32, RV64:
  pack rd, rs1, rs2

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

This 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 upper | lower;
}
```

Applications include XLEN/2-bit funnel shifts, zero-extend XLEN/2 bit values, duplicate the lower XLEN/2 bits (e.g. for mask creation), and loading unsigned 32 constants on RV64.

```
; 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
```

`pack` is most commonly used to zero-extend words <XLEN. For this purpose we define the following assembler pseudo-ops:

```
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   ->    pack  rd, rs, zero

RV128:.
  zext.b rd, rs   ->    andi  rd, rs, 255
  zext.h rd, rs   ->    packw rd, rs, zero
  zext.w rd, rs   ->    packd rd, rs, zero
  zext.d rd, rs   ->    pack  rd, rs, zero
```

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.

SAT solvers spend a lot of time calculating the absolute value of a signed integer due to the way CNF literals are commonly encoded [8]. With `max` (or `minu`) this is a two-instruction operation:

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

2.1.6 Single-bit instructions (`sbs, sbr, sbc, sbt`)

RISC-V Bitmanip ISA

```
RV32, RV64:
  sbs  rd, rs1, rs2
  sbr  rd, rs1, rs2
  sbc  rd, rs1, rs2
  sbt  rd, rs1, rs2
  sbsi rd, rs1, imm
  sbri rd, rs1, imm
  sbci rd, rs1, imm
  sbti rd, rs1, imm

RV64:
  sbsw  rd, rs1, rs2
  sbrw  rd, rs1, rs2
  sbcw  rd, rs1, rs2
  sbtw  rd, rs1, rs2
  sbsiw rd, rs1, imm
  sbriw rd, rs1, imm
  sbciw rd, rs1, imm
```

We define 4 single-bit instructions `sbs` (set), `sbr` (reset), `sbc` (complement), and `sbt` (test), and their immediate-variants, with the following semantics:

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

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

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

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

2.1.7 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

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'.

```
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)));
}
```

Rotate shift is implemented very similarly to the other shift instructions. One possible way to encode it is to re-use the way that bit 30 in the instruction encoding selects 'arithmetic shift' when bit 31 is zero (signalling a logical-zero shift). We can re-use this so that when bit 31 is set (signalling a logical-ones shift), if bit 30 is also set, then we are doing a rotate. The following table summarizes the behavior. The generalized reverse instructions can be encoded using the bit pattern that would otherwise encode an "Arithmetic Left Shift" (which is an operation that does not exist). Likewise, the generalized zip instruction can be encoded using the bit pattern that would otherwise encode an "Rotate left immediate".

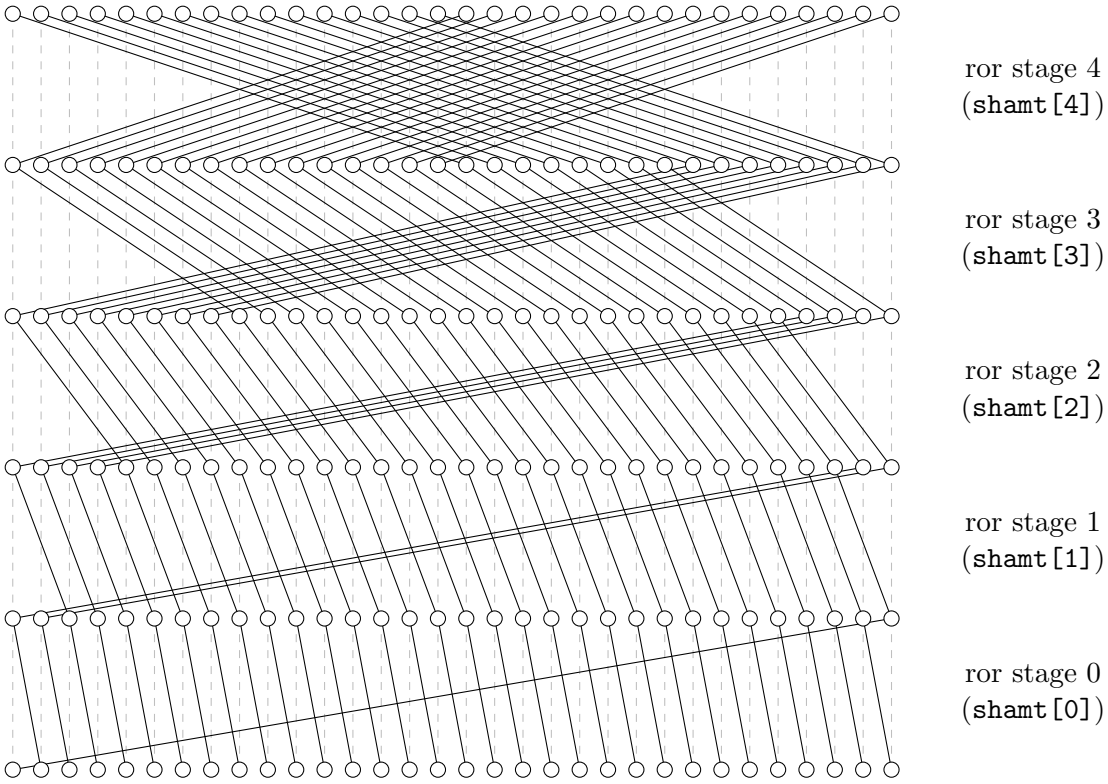


Figure 2.1: `ror` permutation network

Bit 31	Bit 30	Meaning
0	0	Logical Shift-Zeros
0	1	Arithmetic Shift
1	0	Logical Shift-Ones
1	1	Rotate

2.2.2 Generalized Reverse (`grev`, `grevi`)

RISC-V Bitmanip ISA

```
RV32, RV64:
  grev  rd, rs1, rs2
  grevi rd, rs1, imm

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

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. It takes in a single register value and an immediate

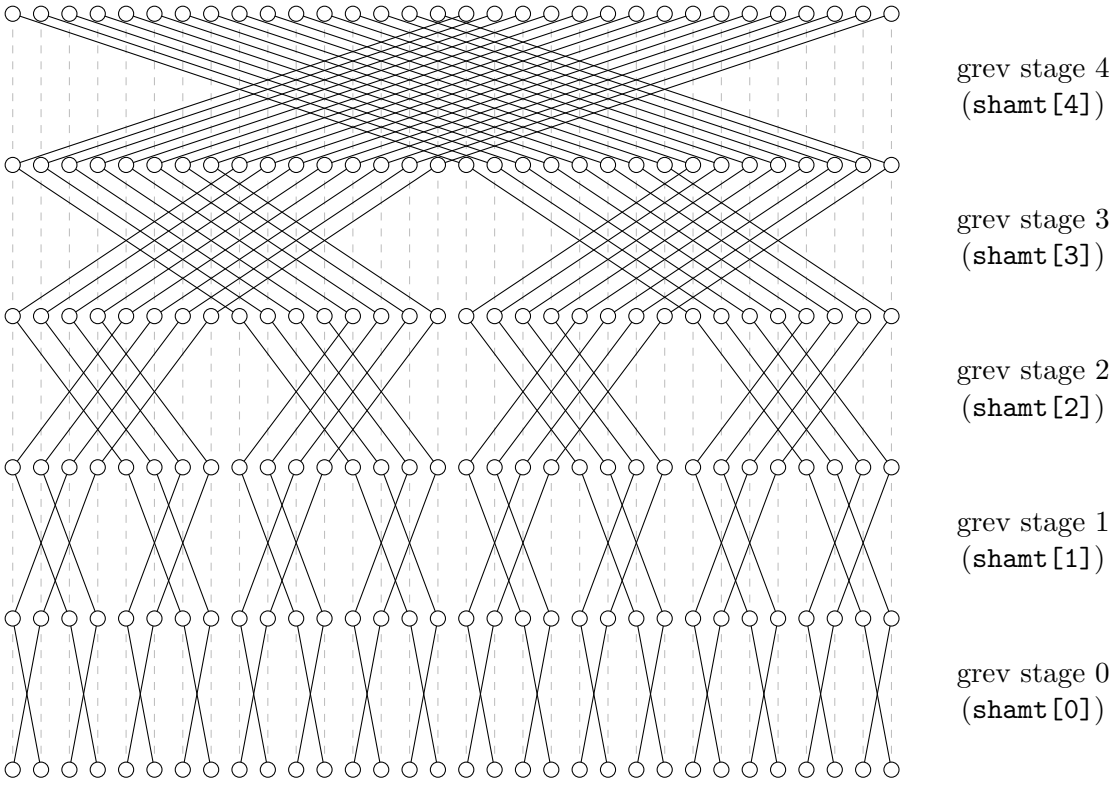


Figure 2.2: `grev` permutation network

that controls which function occurs, through controlling the levels in the recursive tree at which reversals occur.

This operation iteratively checks each bit i in rs2 from $i = 0$ to 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`.

`grev` is encoded as standard R-type opcode and `grevi` is encoded as standard I-type opcode. `grev` and `grevi` can use the instruction encoding for "arithmetic shift left".

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

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		RV64			
shamt	Instruction	shamt	Instruction	shamt	Instruction
0: 00000	—	0: 000000	—	32: 100000	`wswap`
1: 00001	`brev.p`	1: 000001	`brev.p`	33: 100001	—
2: 00010	`pswap.n`	2: 000010	`pswap.n`	34: 100010	—
3: 00011	`brev.n`	3: 000011	`brev.n`	35: 100011	—
4: 00100	`nswap.b`	4: 000100	`nswap.b`	36: 100100	—
5: 00101	—	5: 000101	—	37: 100101	—
6: 00110	`pswap.b`	6: 000110	`pswap.b`	38: 100110	—
7: 00111	`brev.b`	7: 000111	`brev.b`	39: 100111	—
8: 01000	`bswap.h`	8: 001000	`bswap.h`	40: 101000	—
9: 01001	—	9: 001001	—	41: 101001	—
10: 01010	—	10: 001010	—	42: 101010	—
11: 01011	—	11: 001011	—	43: 101011	—
12: 01100	`nswap.h`	12: 001100	`nswap.h`	44: 101100	—
13: 01101	—	13: 001101	—	45: 101101	—
14: 01110	`pswap.h`	14: 001110	`pswap.h`	46: 101110	—
15: 01111	`brev.h`	15: 001111	`brev.h`	47: 101111	—
16: 10000	`hswap`	16: 010000	`hswap.w`	48: 110000	`hswap`
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	`bswap`	24: 011000	`bswap.w`	56: 111000	`bswap`
25: 11001	—	25: 011001	—	57: 111001	—
26: 11010	—	26: 011010	—	58: 111010	—
27: 11011	—	27: 011011	—	59: 111011	—
28: 11100	`nswap`	28: 011100	`nswap.w`	60: 111100	`nswap`
29: 11101	—	29: 011101	—	61: 111101	—
30: 11110	`pswap`	30: 011110	`pswap.w`	62: 111110	`pswap`
31: 11111	`brev`	31: 011111	`brev.w`	63: 111111	`brev`

Table 2.1: Pseudo-instructions for `grevi` instruction

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 continuous regions of set control bits), and the `unshfl` operation performs the swaps in LSB-to-MSB order (performing a rotate right shift on continuous 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 continuous region of set bits. We call those operations zip/unzip and provide pseudo-instructions for them.

Shuffle/unshuffle operations that only have individual bits set (not a continuous 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 2.2: 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.

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

```
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;
}
```

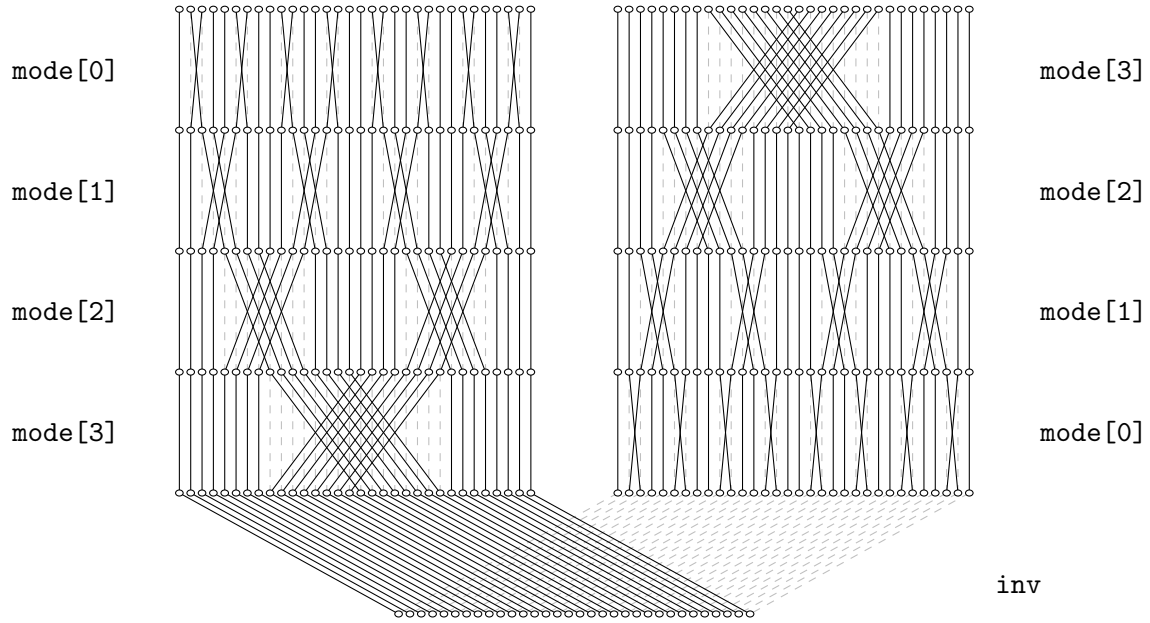


Figure 2.3: (un)shuffle permutation network without "flip" stages

```
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;
}

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 `bdep` 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
```

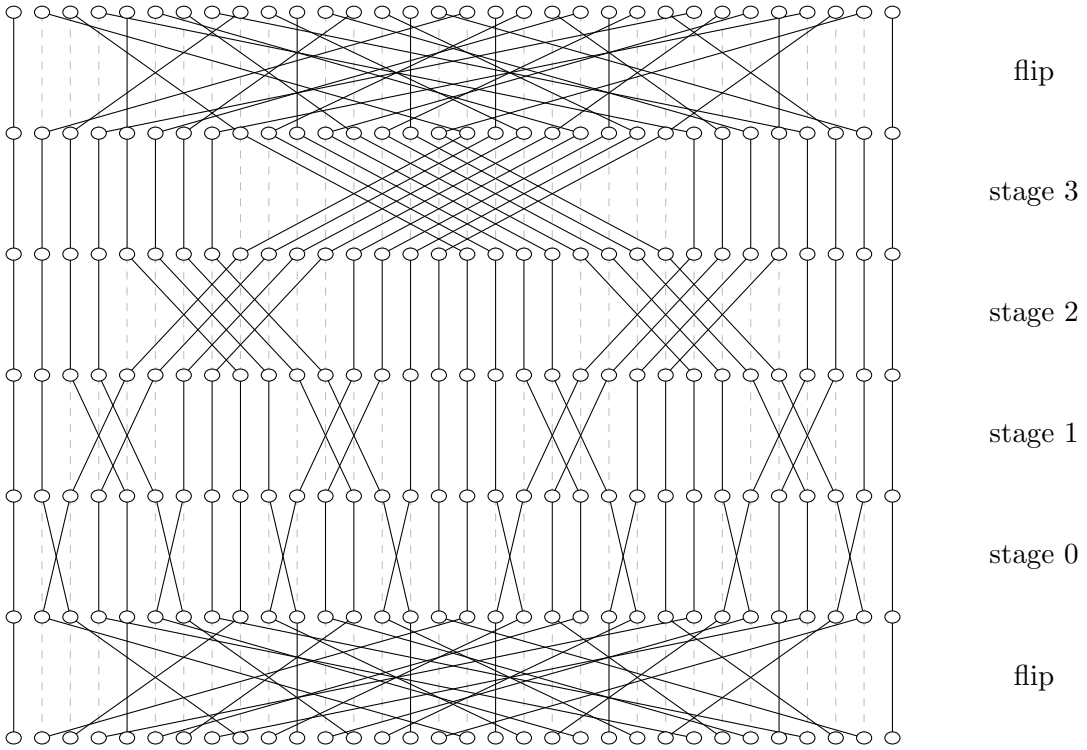


Figure 2.4: (un)shuffle permutation network with "flip" stages

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:

```
andi a0, a0, 0xff
zip a0, a0
zip a0, a0
slli a1, a0, 1
or a0, a0, a1
slli a1, a0, 2
or a0, a0, a1
li a1, 0x87654321
and a1, a0, a1
bext 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
nswap.b t1, t0 ; <- 0x1020304050607080
zip8 t2, a0    ; <- 0x0012003400560078
bswap.h t3, t2 ; <- 0x1200340056007800
zip16 t4, a0   ; <- 0x0000123400005678
hswap.w t5, t4 ; <- 0x1234000056780000
```

Another application for the zip instruction is generating Morton code [**?**, MortonCode]

2.3 Bit Extract/Deposit (`bext`, `bdep`)

RISC-V Bitmanip ISA

```
RV32, RV64:
  bext rd, rs1, rs2
  bdep rd, rs1, rs2

RV64 only:
  bextw rd, rs1, rs2
  bdepw 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.

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

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

```
uint_xlen_t bext(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 bdep(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 `bext` and `bdep`, or even emulate the instructions in software.

Even though multi-cycle `bext` and `bdep` 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 `bdep`:

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

For cases with a constant mask an optimizing compiler would decide when to use `bext` or `bdep` 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 `bext` and `bdep` 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.)

2.4 Carry-less multiply (`clmul`, `clmulh`, `clmulhx`)

RISC-V Bitmanip ISA

```
RV32, RV64:
  clmul   rd, rs1, rs2
  clmulh  rd, rs1, rs2
  clmulhx rd, rs1, rs2

RV64 only:
  clmulw   rd, rs1, rs2
  clmulhw  rd, rs1, rs2
  clmulhxw rd, rs1, rs2
```

Calculate the carry-less product [17] 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.

`clmulhx` produces the upper half of the 2·XLEN carry-less product after appending a 1-bit to `rs2`, which is equivalent to XOR'ing `rs1` with the `clmulh` result. In CRC calculations `clmulhx` eliminates those additional XORs.

Unlike `mulh[[s]u]`, we add *W variants of `clmulh[x]`. 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 clmulhx(uint_xlen_t rs1, uint_xlen_t rs2)
{
    return clmulh(rs1, rs2) ^ rs1;
}
```

The classic applications for `clmul` are CRC [9, 20] and 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 [13].

`clmul` of a number with itself inserts zeroes between each input bit. This can be useful for generating Morton code [19].

`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. [12]

Carry-less multiply can also be used to implement Erasure code efficiently. [10]

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

2.5 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 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 `crc32w`/`crc32cw` instructions are equivalent to executing `crc32h`/`crc32ch` twice, and `crc32h`/`crc32ch` instructions are equivalent to executing `crc32b`/`crc32cb` 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

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

with $N \in \{8, 16, 32, 64\}$, $P =$ `0xEDB8_8320` for CRC32 and $P =$ `0x82F6_3B78` for CRC32C, a' denoting the XLEN bit reversal of a, and $\{a, b\}$ denoting bit concatenation. Note that for example for CRC32 $\{\texttt{1}, P'\} =$ `0x1_04C1_1DB7` on RV32 and $\{\texttt{1}, P'\} =$ `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 [9] and demonstrated in [20] 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, 0x04C11DB7
  li t1, 0x04D101DF
  brev a0, a0
  clmulhx a1, a0, t1
  clmul a0, a1, t0
  brev a0, a0
  ret
```

2.6 Bit-matrix operations (`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.

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 [5]. The Cray X1 architecture even has a full 64x64 bit matrix multiply unit [4].

The MMIX architecture has MOR and MXOR instructions with the same semantic. [11, p. 182f]

```
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 (pcnt(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;
}
```

2.7 Ternary bit-manipulation instructions

2.7.1 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 is equivalent to the following sequence.

```
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).

2.7.2 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, rs1, rs2, 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;
}
```

2.7.3 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`, `C`, 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`.

2.8 Unsigned address calculation instructions

Consider C code that's using unsigned 32-bit ints as array indices. For example:

```
char addiwu_demo(char *p, unsigned int i) {
  return p[i-1];
}

int slliuw_demo(int *p, unsigned int i, unsigned int j) {
  return p[i^j];
}
```

In both cases the expression within `p[...]` must overflow according to 32-bit arithmetic, then be zero-extended, and then this zero-extended result must be used in the address calculation.

The instructions below make sure that no explicit `zext.w` instruction is needed in those cases, to make sure there is no systematic performance penalty for code like shown above on RV64 compared to RV32.

2.8.1 Add/sub with postfix zero-extend (`addwu`, `subwu`, `addiwu`)

RISC-V Bitmanip ISA

```
RV64:
  addwu rd, rs1, rs2
  subwu rd, rs1, rs2
  addiwu rd, rs1, imm
```

These instructions are identlical to `addw`, `subw`, `addiw`, except that bits XLEN-1:32 of the result are cleared after the addition. I.e. these instructions zero-extends instead of sign-extends the 32-bit result.

```
uint_xlen_t addwu(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t result = rs1 + rs2;
    return (uint32_t)result;
}

uint_xlen_t subwu(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t result = rs1 - rs2;
    return (uint32_t)result;
}
```

2.8.2 Add/sub/shift with prefix zero-extend (`addu.w`, `subu.w`, `slliu.w`)

RISC-V Bitmanip ISA

```
RV64:
  addu.w rd, rs1, rs2
  subu.w rd, rs1, rs2
  slliu.w rd, rs1, imm
```

`slliu.w` is identlical to `slli`, except that bits XLEN-1:32 of the `rs1` argument are cleared before the shift.

`addu.w` and `subu.w` are identlical to `add` and `sub`, except that bits XLEN-1:32 of the `rs2` argument are cleared before the shift.

```
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 rs2u = (uint32_t)rs2;
    return rs1 + rs2u;
}

uint_xlen_t subuw(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t rs2u = (uint32_t)rs2;
    return rs1 - rs2u;
}
```

2.9 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. GREV occupies the spot that would decode as SLA (arithmetic left shift).

`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 `slliu.w` 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	GREV	SLO	SRO	ROL	ROR	FSL	FSR
op[30]	0	0	1	1	0	0	1	1	-	-
op[29]	0	0	0	0	1	1	1	1	-	-
op[26]	0	0	0	0	0	0	0	0	1	1
funct3	001	101	101	001	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:

	SBS	SBR	SBC	SBT
op[30]	0	1	1	1
op[29]	1	0	1	0
op[27]	1	1	1	1
funct3	001	001	001	101

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

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

The encoding of `clmul, clmulhx, 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. `bdep` and `bext` 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`.

`addwu` and `subwu` are encoded like `addw` and `subw`, except that `op[25]=1` and `op[27]=1`.

`addu.w` and `subu.w` are encoded like `addw` and `subw`, except that `op[27]=1`.

`addiwu` is encoded using `funct3=100` (XOR) instead of `funct3=000` in OP-32.

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.

`clmulh[x]` has no `*W` variants because, similar to `mulh`, one doesn't need a 32-bit mulh operator if a 64-bit mul operator is available.

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 / 001 / rd / 0110011	GREV
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
0010100	rs2	rs1 / 001 / rd / 0110011	SBS
0100100	rs2	rs1 / 001 / rd / 0110011	SBR
0110100	rs2	rs1 / 001 / rd / 0110011	SBC
0100100	rs2	rs1 / 101 / rd / 0110011	SBT
00000	imm	rs1 / 001 / rd / 0010011	SLLI*
00000	imm	rs1 / 101 / rd / 0010011	SRLI*
01000	imm	rs1 / 001 / rd / 0010011	GREVI
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
00101	imm	rs1 / 001 / rd / 0110011	SBSI
01001	imm	rs1 / 001 / rd / 0110011	SBRI
01101	imm	rs1 / 001 / rd / 0110011	SBCI
01001	imm	rs1 / 101 / rd / 0110011	SBTI
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			
funct7	rs2	rs1 f3 rd opcode	R-type
rs3 f2	rs2	rs1 f3 rd opcode	R4-type
imm		rs1 f3 rd opcode	I-type
0110000	00000	rs1 001 rd 0010011	CLZ
0110000	00001	rs1 001 rd 0010011	CTZ
0110000	00010	rs1 001 rd 0010011	PCNT
0110000	00011	rs1 001 rd 0010011	BMATFLIP
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	CLMULHX
0000101	rs2	rs1 011 rd 0110011	CLMULH
0000101	rs2	rs1 100 rd 0110011	MIN
0000101	rs2	rs1 101 rd 0110011	MAX
0000101	rs2	rs1 110 rd 0110011	MINU
0000101	rs2	rs1 111 rd 0110011	MAXU
0000100	rs2	rs1 001 rd 0110011	SHFL
0000100	rs2	rs1 101 rd 0110011	UNSHFL
0000100	rs2	rs1 010 rd 0110011	BDEP
0000100	rs2	rs1 110 rd 0110011	BEXT
0000100	rs2	rs1 100 rd 0110011	PACK
0000100	rs2	rs1 011 rd 0110011	BMATOR
0000100	rs2	rs1 111 rd 0110011	BMATXOR
000010	imm	rs1 001 rd 0010011	SHFLI
000010	imm	rs1 101 rd 0010011	UNSHFLI
immediate		rs1 000 rd 0011011	ADDIW*
immediate		rs1 100 rd 0011011	ADDIWU
00001	imm	rs1 001 rd 0011011	SLLIU.W
0000000	rs2	rs1 000 rd 0111011	ADDW*
0100000	rs2	rs1 000 rd 0111011	SUBW*
0000101	rs2	rs1 000 rd 0111011	ADDWU
0100101	rs2	rs1 000 rd 0111011	SUBWU
0000100	rs2	rs1 000 rd 0111011	ADDU.W
0100100	rs2	rs1 000 rd 0111011	SUBU.W

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	001	rd	0111011	SLLW*
0000000	rs2	rs1	101	rd	0111011	SRLW*
0100000	rs2	rs1	001	rd	0111011	GREVW
0100000	rs2	rs1	101	rd	0111011	SRAW*
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
0010100	rs2	rs1	001	rd	0111011	SBSW
0100100	rs2	rs1	001	rd	0111011	SBRW
0110100	rs2	rs1	001	rd	0111011	SBCW
0100100	rs2	rs1	101	rd	0111011	SBTW
0000000	imm	rs1	001	rd	0011011	SLLIW*
0000000	imm	rs1	001	rd	0011011	SRLIW*
0100000	imm	rs1	001	rd	0011011	GREVIW
0100000	imm	rs1	001	rd	0011011	SRAIW*
0010000	imm	rs1	001	rd	0011011	SLOIW
0010000	imm	rs1	101	rd	0011011	SROIW
0110000	imm	rs1	101	rd	0011011	RORIW
0010100	imm	rs1	001	rd	0110011	SBSIW
0100100	imm	rs1	001	rd	0110011	SBRIW
0110100	imm	rs1	001	rd	0110011	SBCIW
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	PCNTW
0000101	rs2	rs1	001	rd	0111011	CLMULW
0000101	rs2	rs1	010	rd	0111011	CLMULHXW
0000101	rs2	rs1	011	rd	0111011	CLMULHW
0000100	rs2	rs1	001	rd	0111011	SHFLW
0000100	rs2	rs1	101	rd	0111011	UNSHFLW
0000100	rs2	rs1	010	rd	0111011	BDEPW
0000100	rs2	rs1	110	rd	0111011	BEXTW
0000100	rs2	rs1	100	rd	0111011	PACKW

2.10 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 (and RV128) `zext.w` and `zext.d` (`pack` and `packw`) are commonly used to zero-extend unsigned values <XLEN.

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`.

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.ZEXT.W *(RV64/128)*
011	0	11	rs1′/rd′	0	01	C.ZEXT.D *(RV128)*

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. [21]

2.11 Micro architectural considerations and macro-op fusion for bit-manipulation

2.11.1 Fast `MUL`, `MULH`, `MULHSU`, `MULHU`

A lot of bit manipulation code depends on "multiply with magic number"-tricks. Often those tricks need the upper half of the $2 \cdot \text{XLEN}$ product. Therefore decent performance for the `MUL` and especially `MULH[[S]U]` instructions is important for fast bit manipulation code.

2.11.2 Fused load-immediate sequences

Bit manipulation code, even more than other code, requires a lot of "magic numbers", bitmasks, and other (usually large) constants. On some microarchitectures those can easily be loaded from a

nearby data section using load instructions. On other microarchitectures however this comes at a high cost, and it is more efficient to load immediates using a sequence of instructions.

Loading a 32-bit constant:

```
lui rd, imm
addi rd, rd, imm
```

On RV64 a 64 bit constant can be loaded by loading two 32-bit constants anmd combining them with a `PACK` instruction:

```
lui tmp, imm
addi tmp, tmp, imm
lui rd, imm
addi rd, rd, imm
pack rd, rd, tmp
```

(Without the temporary register and without the `PACK` instruction more complex/diverse sequences are used to load 64-bit immediates. But the `PACK` instruction streamlines the pattern and thus simplifies macro-op fusion.)

A 32-bit core should be capable of fusing the `lui`+`addi` pattern.

In addition to that, a 64 bit core may consider fusing the following sequences as well:

```
lui rd, imm
addi rd, rd, imm
pack rd, rd, rs2

lui rd, imm
pack rd, rd, rs2

addi rd, zero, imm
pack rd, rd, rs2
```

Furthermore, a core may consider fusing 32-bit immediate loads with any ALU instruction, not just `pack`:

```
lui rd, imm
addi rd, rd, imm
alu_op rd, rd, rs2

lui rd, imm
alu_op rd, rd, rs2

addi rd, zero, imm
alu_op rd, rd, rs2
```

And finally, a 64-bit core should fuse sequences with `addiwu` as well as `addi`, for loading unsigned 32-bit numbers that have their MSB set. This is often the case with masks in bit manipulation code.

2.11.3 Fused `*-not` sequences

Preliminary experiments have shown that NOT instructions make up approximately 1% of bitmanipulation code size, more when looking at dynamic instruction count. [21]

Therefore it makes sense to fuse NOT instructions with other ALU instructions, if possible.

The most important form of NOT fusion is postfix fusion:

```
alu_op rd, rs1, rs2
not rd, rd
```

A future compressed NOT instruction would help keeping those fused sequences short.

2.11.4 Fused `*-srli` and `*-srai` sequences

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

To extract the continuous bit field starting at `pos` with length `len` from `rs` (with `pos` > 0, `len` > 0, and `pos` + `len` $\leq$ 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-pos)
srli rd, rd, (XLEN-len)
```

If possible, an implementation should fuse the following macro ops:

```
alu_op rd, rs1, rs2
srli rd, rd, imm

alu_op rd, rs1, rs2
srai rd, rd, imm
```

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

For generating masks, i.e. constants with one continous run of 1 bits, a sequence like the following can be used that would utilize postfix fusion of right shifts:

```
sroi rd, zero, len
c.srli rd, (XLEN-len-pos)
```

This can be a useful sequence on RV64, where loading an arbitrary 64-bit constant would usually require at least 96 bits (using `c.ld`).

2.11.5 Fused 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 == rs1`:

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

  beq rd, zero, skip_sltu
  sltu rd, zero, rs2
skip_sltu:
```

Note that the first instruction can be compressed in all four cases if `rd == rs1`.

2.11.6 Fused ternary ALU sequences

Architectures with support for ternary operations may want to support fusing two ALU operations.

```
  alu_op rd, ...
  alu_op rd, rd, ...
```

This would be a postfix-fusion pattern, extending the postfix shift-right fusion described in the previous section.

Candidates for this kind of postfix fusion would be simple ALU operations, specifically AND/OR/X-OR/ADD/SUB and ANDI/ORI/XORI/ADDI/SUBI.

2.11.7 Pseudo-ops for fused sequences

Assembler pseudo-ops for `not` postfix fusion:

```
nand rd, rs1, rs2        ->   and rd, rs1, rs2; not rd, rd
nor  rd, rs1, rs2        ->   or  rd, rs1, rs2; not rd, rd
xnor rd, rs1, rs2        ->   xor rd, rs1, rs2; not rd, rd
```

Assembler bitfield pseudo-ops for `c.sr[la]i` postfix fusion:

```
bfext  rd, rs, len, pos    ->   slli rd, rs, (XLEN-len-pos); c.srai rd, (XLEN-len)
bfextu rd, rs, len, pos    ->   slli rd, rs, (XLEN-len-pos); c.srli rd, (XLEN-len)
bfmak  rd, len, pos        ->   sroi rd, zero, len; c.srli 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. [3, p. 3-28]

2.12 Fast C reference implementations

GCC has intrinsics for the bit counting instructions `clz`, `ctz`, and `pcnt`. 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 XLEN;
    assert(sizeof(int) == 4);
    return __builtin_clz(rs1);
}

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

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

uint64_t fast_ctz64(uint64_t rs1)
{
    if (rs1 == 0)
        return XLEN;
    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.

Chapter 3

Evaluation

This chapter contains a collection of short code snippets and algorithms using the Bitmanip extension for evaluation purposes. For the sake of simplicity we assume RV32 for most examples in this chapter.

3.1 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)
```

3.2 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.

3.2.1 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`.

3.2.2 Parsing bit-streams

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.

3.2.3 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
```

3.3 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).

3.3.1 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 (preformatted mask in `a1` on exit):

```
bfly_msk_0:
  zip a1, a1
  slli a2, a1, 1
  or a1, a1, a2

bfly_msk_1:
  zip2 a1, a1
  slli a2, a1, 2
  or a1, a1, a2

bfly_msk_2:
  zip4 a1, a1
  slli a2, a1, 4
  or a1, a1, a2

...
```

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 with either the sheep or the goats bit reversed can be implemented in log_2(XLEN) butterfly operations.

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

3.3.2 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.

3.3.3 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.

3.3.4 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
pcnt 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. [7, p. 152f, 162f]

3.3.5 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`.

3.4 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.

3.4.1 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       brev.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       bswap
 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       brev
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
```

3.4.2 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:
  bswap a0, a0
  zip a0, a0
  zip a0, a0
  zip a0, a0
```

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

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

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

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

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` (`brev.b`).

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

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.

3.4.4 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:
  hswap a0, a0
  zip4 a0, a0
  zip4 a0, a0
```

```
rotate_y:
  brev.n a0, a0
  zip a0, a0
  zip a0, a0
  zip4 a0, a0
  zip4 a0, a0
```

```
rotate_z:
  nswap.h
  zip.h a0, a0
  zip.h a0, a0
```

3.5 Rank and select

Rank and select are fundamental operations in succinct data structures [15].

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

```
select:
  li a2, 1
  sll a1, a2, 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
  pcnt a0, a0
  ret
```

3.6 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. [14, 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 `clmulhx(x, 0x08421084)`, where `0x08421084 == 0x42108421 << (clz(0x42108421)+1)` (mod 2^{32}).

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

3.7 Fill right of most significant set bit

The "fill right" or "fold right" operation is a pattern commonly used in bit manipulation code. [6]

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 `brev` to implement this function in 4 instructions:

```
uint64_t rfill_brev(uint64_t x)
{
  x = brev(x);        // GREVI
  x = x | ~(x - 1);   // ADDI, ORN
  x = brev(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;
}
```

3.8 Cyclic redundancy 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 crc = 0;

  for (int i = 0; i < length; i++) {
    crc ^= bswap32(data[i]);
    crc = clmulhx32(crc, mu);
    crc = clmul32(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;   // rest of x^128 divided by CRC polynomial
  uint64_t k2 =  0x9BE9878FLL;   // rest of x^96 divided by CRC polynomial
  uint64_t k3 =  0xB1EFC5F6LL;   // rest 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 = bswap(p[0]);
  a1 = bswap(p[1]);

  // Main loop: Reduce to 2x 64 bits

  for (const uint64_t *t0 = p+2; t0 != p+len; t0++)
  {
    a2 = bswap(*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.

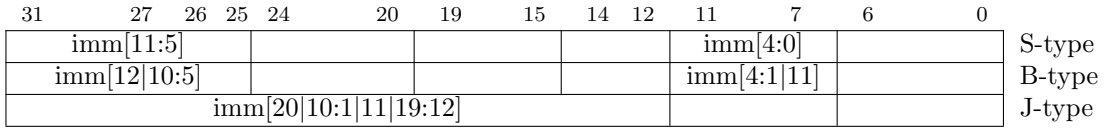
The following python code can be used to calculate the "magic constants" `k1`, `k2`, and `k3`:

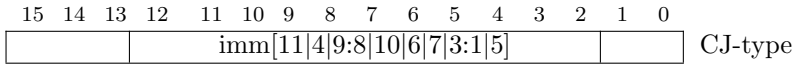
```
def polymod(dividend, divisor):
    quotient = 0
    while dividend.bit_length() >= divisor.bit_length():
        i = dividend.bit_length() - divisor.bit_length()
        dividend = dividend ^ (divisor << i)
        quotient |= 1 << i
    return dividend

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 [20]. A more detailed descriptions of the algorithms employed can be found in [9].

3.9 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
```

Change History

Date	Rev	Changes
2017-07-17	0.10	Initial Draft
2017-11-02	0.11	Removed roli, assembler can convert it to use a rori
		Removed bitwise subset and replaced with `andc`
		Doc source text same base for study and spec.
		Fixed typos
2017-11-30	0.32	Jump rev number to be on par with associated Study
		Moved 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
		Fixed typos and fixed 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
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`

Date	Rev	Changes
????-??-??	0.37	Add dedicated CRC instructions
		Add proposed opcode encodings
		Renamed from XBitmanip to RISC-V Bitmanip
		Removed chapter on `bfxp[c]` instruction
		Refactored proposal into one big chapter
		Removed `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 `sbs[i]`, `sbr[i]`, `sbc[i]`, `sbt[i]`

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

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

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

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

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

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

[9] 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.

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

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

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

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

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

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

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

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

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

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

[20] Clifford Wolf. Reference implementations of various crcs using carry-less multiply. `http://svn.clifford.at/handicraft/2018/clmulcrc/`. Accessed: 2018-11-06.

[21] Clifford Wolf. A simple synthetic compiler benchmark for bit manipulation operations. `http://svn.clifford.at/handicraft/2017/bitcode/`. Accessed: 2017-04-30.