RISC-V Bitmanip Extension

Document Version 0.37-draft

Editor: Clifford Wolf
Symbiotic GmbH
clifford@symbioticeda.com

May 26, 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.

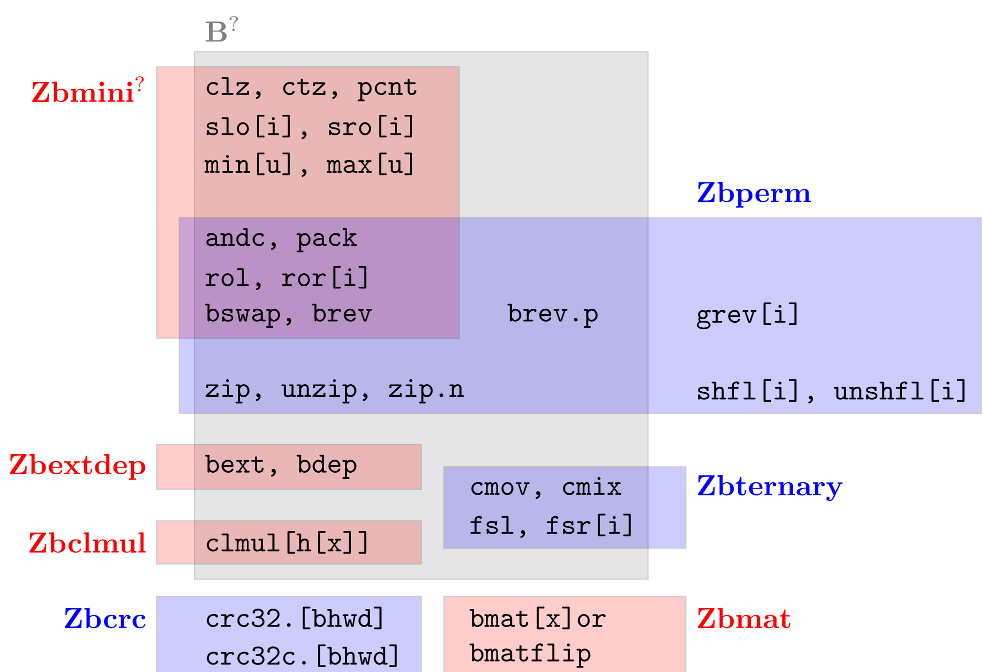
Chapter 2

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 3.1 for fast C code for use in emulators.

The section on opcode encodings are mere placeholders.

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 completely unclear as of yet what this will look like, but a guess could look 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 "Zbmini". 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 open questions are:

- Should `clmul[h]` be included, or `crc32.[bhwd]/crc32c.[bhwd]`, or neither, or both?
- Should Zbternary be included? It adds a large architectural cost by adding instructions with three source operands.
- Should Zbextdep be included? Should Zbmat be included?
- Which Zbperm 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. [20, 18]

```
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 And-with-complement (andc)

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

This instruction implements the and-with-complement operation.

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

Other with-complement operations (`orc, nand, nor`, etc) can be implemented by combining `not` with the base ALU operation. Only and-with-complement occurs frequently enough to warrant a dedicated instruction.

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 [9]. With `max` (or `minu`) this is a two-instruction operation:

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

2.1.6 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”.

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

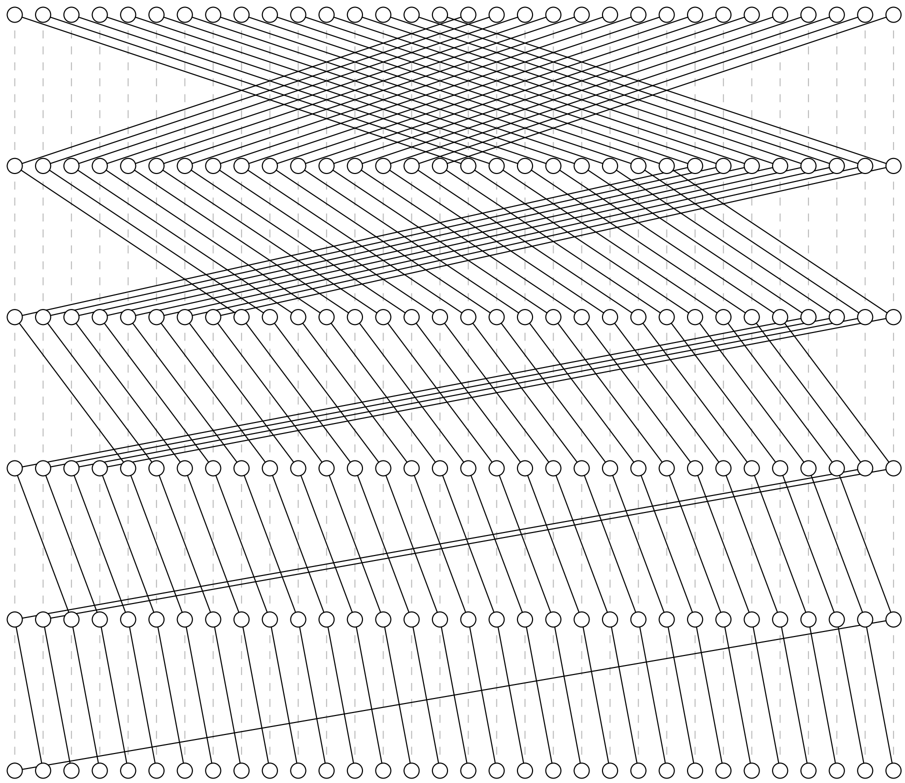


Figure 2.1: `ror` permutation network

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 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 $\mathrm{XLEN}-1$, and if the corresponding bit is set, swaps each adjacent pair of 2^i bits.

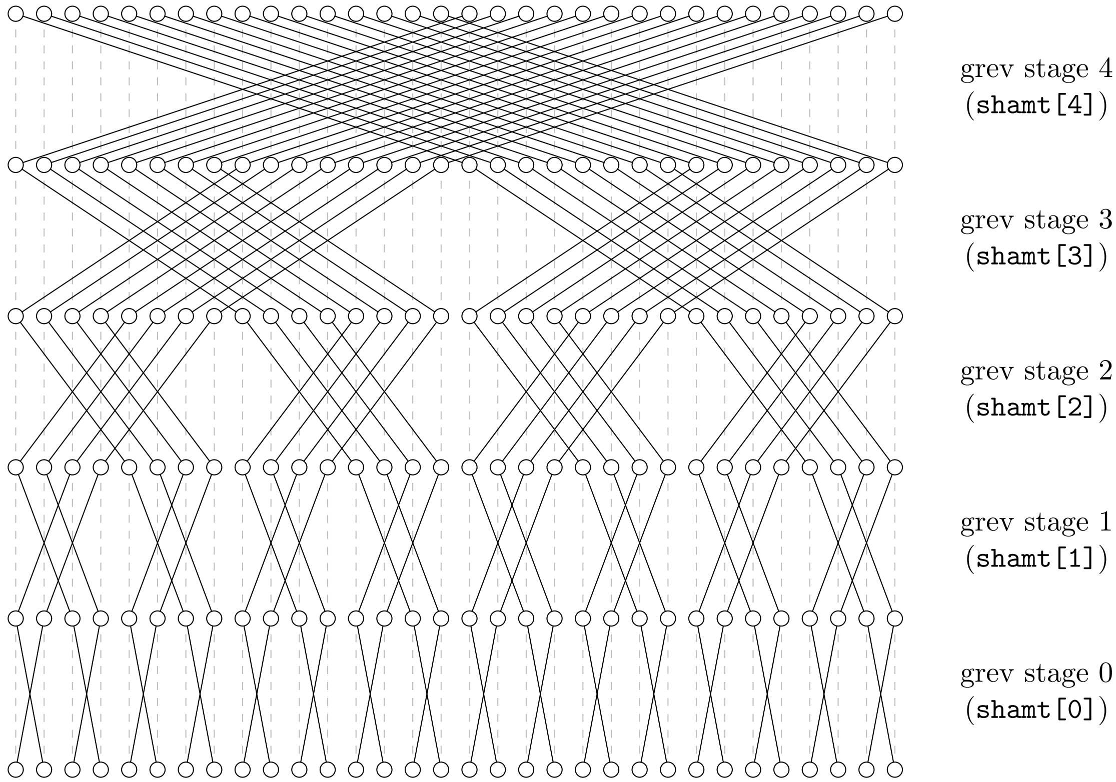


Figure 2.2: `grev` permutation network

```
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
  shfliw   rd, rs1, imm
  unshfliw rd, rs1, imm
```

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.

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

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 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(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;
}
```

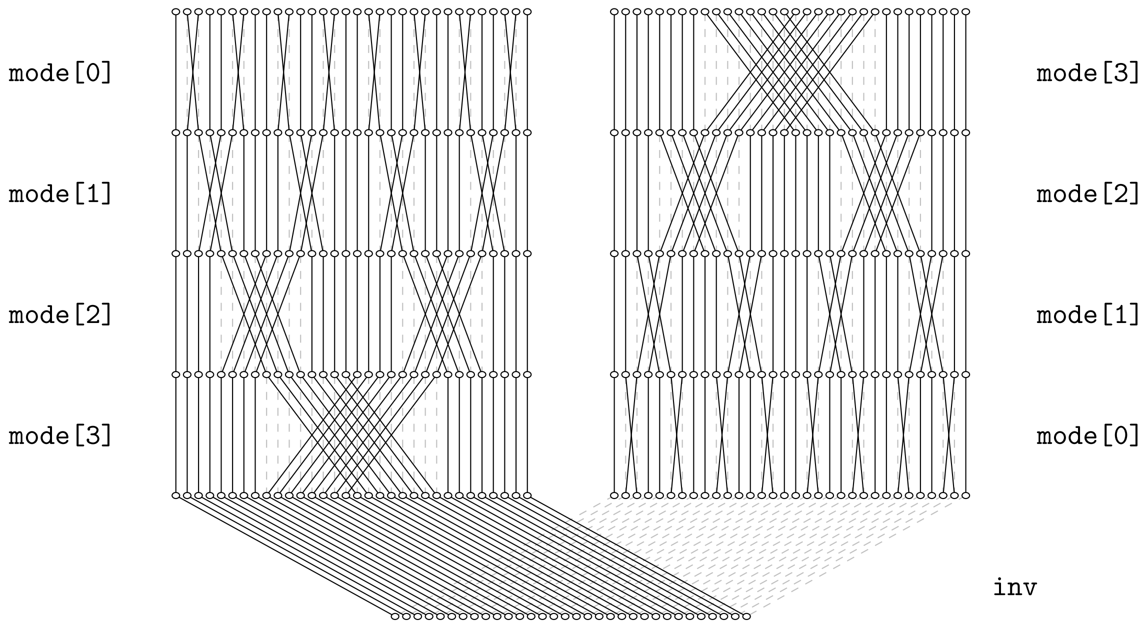


Figure 2.3: (un)shuffle permutation network without "flip" stages

```
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
```

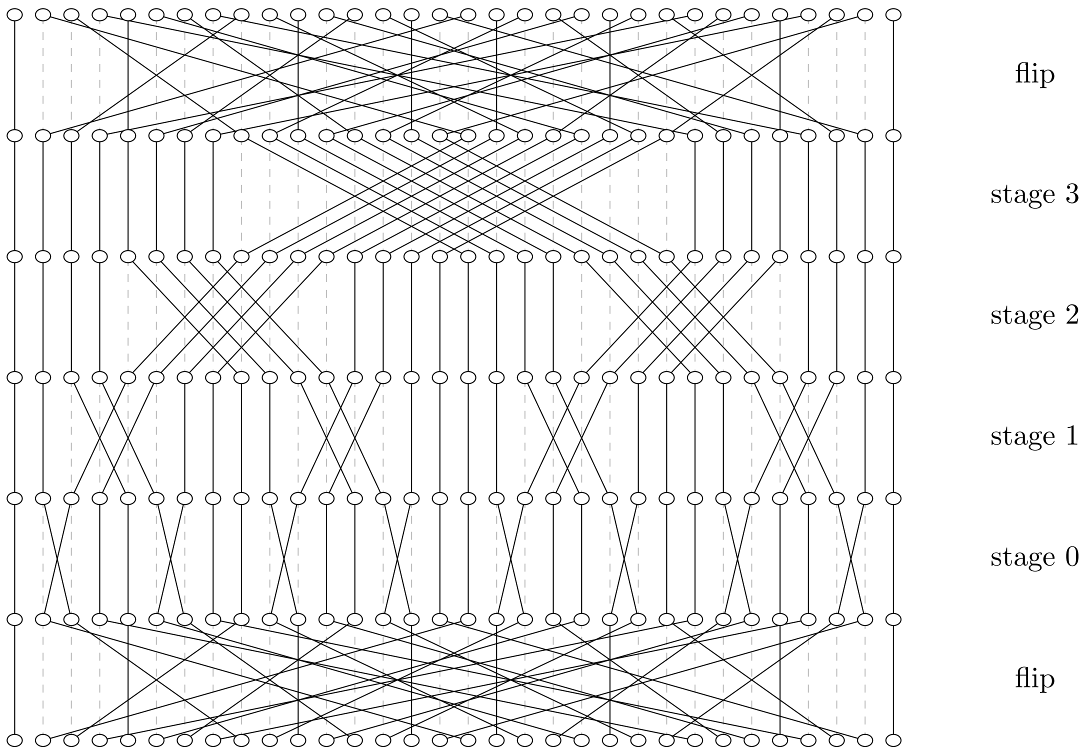


Figure 2.4: (un)shuffle permutation network with “flip” stages

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
```

Calculate the carry-less product [19] 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.

```
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 [10, 23] 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 [15].

`clmul` of a number with itself inserts zeroes between each input bit. This can be useful for generating Morton code [21].

`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. [14]

Carry-less multiply can also be used to implement Erasure code efficiently. [12]

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

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

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 $\{1, P'\} =$ `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 [10] and demonstrated in [23] 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
  clmulh a1, a0, t1
  xor a1, a1, a0
  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. [13, 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
andc 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 `andc` instruction is encoded the same way as `and`, 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 `fsli` instruction, as it can be easily emulated with `fsri` by taking the two's complement of the shift amount. The `sllu.w` instruction occupies the encoding slot that would be occupied by `fsli`.

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

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

The encoding of `clmul[h[x]]` is identical to the encoding of `mul[h]`, except that `op[27]=1` and `funct3[0]` is inverted. `op[30]` is used to turn `clmulh` into `clmulhx`.

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

bmat[x]or uses funct3=000 in funct7=0000100 with op[30] selecting the XOR variant of the instruction.

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.

pack occupies funct3=001 in funct7=0000100.

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

andc, 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.

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.

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*
|  0100000      |    rs2    |    rs1    | 111 |    rd    |   0110011    |  ANDC
|---------------------------------------------------------------|
|  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
|   rs3    | 10|    rs2    |    rs1    | 001 |    rd    |   0110011    |  FSL
|   rs3    | 10|    rs2    |    rs1    | 101 |    rd    |   0110011    |  FSR
|---------------------------------------------------------------|
|  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
|   rs3    |1|     imm      |    rs1    | 101 |    rd    |   0010011    |  FSRI
|---------------------------------------------------------------|
|  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
|---------------------------------------------------------------|
|   rs3    | 11|    rs2    |    rs1    | 001 |    rd    |   0110011    |  CMIX
|   rs3    | 11|    rs2    |    rs1    | 101 |    rd    |   0110011    |  CMOV
|---------------------------------------------------------------|
```

```
|  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
|===============================================================|
|  0000101     |    rs2    |    rs1    | 000 |     rd    |    0110011    |  CLMULH
|  0100101     |    rs2    |    rs1    | 000 |     rd    |    0110011    |  CLMULHX
|  0000101     |    rs2    |    rs1    | 001 |     rd    |    0110011    |  CLMUL
|  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    | 000 |     rd    |    0110011    |  BMATOR
|  0100100     |    rs2    |    rs1    | 000 |     rd    |    0110011    |  BMATXOR
|  0000100     |    rs2    |    rs1    | 001 |     rd    |    0110011    |  PACK
|  0000100     |    rs2    |    rs1    | 010 |     rd    |    0110011    |  BDEP
|  0000100     |    rs2    |    rs1    | 110 |     rd    |    0110011    |  BEXT
|  0000100     |    rs2    |    rs1    | 001 |     rd    |    0110011    |  SHFL
|  0000100     |    rs2    |    rs1    | 101 |     rd    |    0110011    |  UNSHFL
|---------------------------------------------------------------|
|  000010    |     imm     |    rs1    | 001 |     rd    |    0010011    |  SHFLI
|  000010    |     imm     |    rs1    | 101 |     rd    |    0010011    |  UNSHFLI
|---------------------------------------------------------------|
```

```
|  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
|   rs3    | 10|    rs2    |    rs1    | 001 |     rd    |    0111011    |  FSLW
|   rs3    | 10|    rs2    |    rs1    | 101 |     rd    |    0111011    |  FSRW
|---------------------------------------------------------------|
|  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
|---------------------------------------------------------------|
|       immediate          |    rs1    | 000 |     rd    |    0011011    |  ADDIW*
|       immediate          |    rs1    | 100 |     rd    |    0011011    |  ADDIWU
|---------------------------------------------------------------|
|  0000101      |    rs2    |    rs1    | 001 |     rd    |    0111011    |  CLMULW
|  0000100      |    rs2    |    rs1    | 001 |     rd    |    0111011    |  PACKW
|  0000100      |    rs2    |    rs1    | 010 |     rd    |    0111011    |  BDEPW
|  0000100      |    rs2    |    rs1    | 110 |     rd    |    0111011    |  BEXTW
|  0000100      |    rs2    |    rs1    | 001 |     rd    |    0111011    |  SHFLW
|  0000100      |    rs2    |    rs1    | 101 |     rd    |    0111011    |  UNSHFLW
|---------------------------------------------------------------|
|  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
|  0110000      |   00000   |    rs1    | 001 |     rd    |    0011011    |  CLZW
|  0110000      |   00001   |    rs1    | 001 |     rd    |    0011011    |  CTZW
|  0110000      |   00010   |    rs1    | 001 |     rd    |    0011011    |  PCNTW
|   rs3    |1|0|    imm    |    rs1    | 101 |     rd    |    0011011    |  FSRIW
|  00000   |1|0|    imm    |    rs1    | 001 |     rd    |    0011011    |  SLLIU.W
|---------------------------------------------------------------|
|  000010000      |   imm   |    rs1    | 001 |     rd    |    0011011    |  SHFLIW
|  000010000      |   imm   |    rs1    | 101 |     rd    |    0011011    |  UNSHFLIW
|---------------------------------------------------------------|
```

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$\neq\{0,2\}$		nzimm[16:12]	01	C.LUI *(~~RES, nzimm=0~~; HINT, rd=0)*
011	0	rs1/rd		0	01	C.NOT *(RV32)*
011	0	00	rs1′/rd′	0	01	C.NOT *(RV64/128)*
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. [24]

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
```

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. [24]

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 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/XOR/ADD/SUB and ANDI/ORI/XORI/ADDI/SUBI.

2.11.6 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]

Chapter 3

Reference Implementations

3.1 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 4

Discussion

IMPORTANT NOTE: Some of the discussions below refer to an older draft of the RISC-V Bitmanip extension and are now out-of-date.

4.1 Frequently Asked Questions

Which instructions were considered but not included in the RISC-V Bitmanip extension?

- A bit-field extract instruction. Unfortunately the encoding space required by such an instruction would be enormous (for forward-looking support up to RV128 this would require a $2 = 14$ bits immediate. This was deemed too expensive considering that the same functionality can already be implemented using only two shift instructions. However, see Chapter **??** for an alternative. (The same encoding space argument applies to instructions for inserting bits in a bit field, or setting or clearing ranges of bits.)

- A "butterfly" instruction with an XLEN/2 immediate that would run one butterfly stage (see Section 5.6.1). The `smartbextdep` implementation of `bext/bdep` (see Section 4.3) already includes the hardware for that, all that would be required is exposing a few internal control signals. However, the encoding cost of such an instruction would be enormous, the instruction itself would only be useful in niche applications, and it would require implementers to implement `bext/bdep` in a certain way. So it's is not worth the effort considering that a butterfly stage can also be implemented in four instructions using `grevi` and the MIX pattern (see Section 5.6.1).

- A reverse-subtract-immediate operation that performs the calculation $\texttt{imm} - \texttt{rs}$. This comes up often in calculating bit indices (for example $\texttt{XLEN} - i$ is a very common operation, and compressed instructions are only of some help here because compressed immediates are in the range $-32 \ldots 31$). However, a reverse-subtract-immediate operation is not very bitmanipulation specific, would require a much larger encoding space than the rest of the RISC-V Bitmanip extension (it would have a 12 bit immediate), and does not add any functionality that can not be emulated easily with `neg` and `addi`: $\texttt{rsbi}(x, k) = \texttt{neg}(\texttt{addi}(x, -k)) = \texttt{addi}(\texttt{neg}(x), k)$

- Conditional move and mix instructions. Those would be very helpful! However, they would require three source operands, and we did not want to add a register file with three read ports as requirement for the RISC-V Bitmanip extension. See Section 5.2 for short sequences implementing conditional move (aka mux) and mix operations.

- Funnel shift instructions. Those are also super helpful! For example when processing input that is a bit-packed stream of words of arbitrary (variable) bit width. However, a funnel shift instruction would also require three source operands, therefore it was not considered for inclusion in the RISC-V Bitmanip extension.

- Instructions for testing, setting and clearing bits. This operations can be performed in few easily macro-op fusible instructions. So implementations might provide hardware support for them, but we do not reserve dedicated opcodes for those operations.

- Instructions for the usual patterns of ALU operations to fiddle with trailing bits, such as `x & -x` (extract lowest set bit), or `x ^ (x - 1)` (mask up to and including lowest set bit). But those operations all fit into short easily macro-op fusible instruction sequences (see Section 6.1).

- Instructions for carry-less multiplication (similar to the x86 CLMUL instructions). They are useful for cryptographic applications, CRC calculations, and compression. But for cryptographic applications they need to be implemented in a way that executes them in constant time to avoid leaking secret information via a timing side-channel. Because of those considerations, and the area requirements for a core implementing the operations, I think it is better to leave these instructions for a RISC-V ISA extension for security instructions.

`grev` seems to be overly complicated? Do we really need it?

The `grev` instruction can be used to build a wide range of common bit permutation instructions, such as endianness conversion or bit reversal.

If `grev` were removed from this spec we would need to add a few new instructions in its place for those operations.

Do we really need all the `*W` opcodes for 32 bit ops on RV64?

I don't know. I think nobody does know at the moment. But they add very little complexity to the core. So the only question is if it is worth the encoding space. We need to run proper experiments with compilers that support those instructions. So they are in for now and if future evaluations show that they are not worth the encoding space then we can still throw them out.

Why only `andc` and not any other complement operators?

Early versions of this spec also included other `*c` operators. But experiments have shown that `andc` is much more common in bit manipulation code than any other operators. [24] Especially because it is commonly used in `mix` and `mux` operations.

Why `andc`? It can easily be emulated using `and` and `not`.

Yes, and we did not include any other ALU+complement operators. But `andc` is so common (mostly because of the `mix` and `mux` patterns), and its implementation is so cheap, that we decided to dedicate an R-type instruction to the operation.

The shift-ones instructions can be emulated using `not` and logical shift? Do we really need it?

Yes, a shift-ones instruction can easily be implemented using the logical shift instructions, with a bitwise invert before and after it. (This is literally the code we are using in the reference C implementation of shift-ones.)

We have decided to include it for now so that we can collect benchmark data before making a final decision on the inclusion or exclusion of those instructions. The main objection here is instruction encoding space. The hardware overhead of adding this functionality to a shifter is relatively low.

BEXT/BDEP look like really expensive operations. Do we really need them?

Yes, they are expensive, but not as expensive as one might expect. A single-cycle 32 bit BEXT+BDEP+GREV core can be implemented in less space than a single-cycle 16x16 bit multiplier with 32 bit output. [22]

It is also important to keep in mind that implementing those operations in software is very expensive. Hacker's Delight contains a highly optimized software implementation of 32-bit BEXT that requires > 120 instructions. Their BDEP software implementation requires > 160 instructions. (Please disregard the "hardware-oriented algorithm" described in Hacker's Delight. It is extremely expensive compared to other implementations. [22])

But do we really need 64-bit BEXT/BDEP?

Good question. A 64-bit BEXT/BDEP unit certainly is more than 2x the size of a 32-bit unit and in most cases 32-bit would be sufficient. It is also not very difficult to emulate 64-bit BEXT/BDEP using 32-bit BEXT/BDEP. On RV64 (with data in `a0` and mask in `a1`):

```
bext64:
  pcntw a2, a1
  bextw a3, a0, a1
  c.srli a0, 32
  c.srli a1, 32
  bextw a0, a0, a1
  sloi a1, zero, 32
  c.and a3, a1
  c.and a0, a1
```

```
bdep64:
  pcntw a2, a1
  bdepw a3, a0, a1
  srl a0, a0, a2
  c.srli a1, 32
  bdepw a0, a0, a1
  c.slli a0, 32
  c.slli a3, 32
  c.srli a3, 32
  c.or a0, a3
  ret
```

However, one solution here would be to still reserve the opcode for 64-bit BEXT/BDEP and leave it to the implementation to decide whether to implement the function in hardware or emulating it using a software trap.

Compressed encoding space is expensive. Do we need `c.neg`, `c.not`, `c.brev`?

Because they are unary operations and encoded with $\texttt{rd}' = \texttt{rs}'$, they only take up $\approx 0.05\%$ of the total "C" encoding space, and only $\approx 1.1\%$ of the remaining reserved "C" encoding space.

They are common building blocks of sequences that are candidates for macro-op fusion, and keeping these sequences short can help with decoding them efficiently, especially `c.not`.

`c.neg` can be emulated using `c.not` if the next (or previous) operation is an `addi`. But commonly `c.neg` is simply used to turn the value 1 into all-bits-set and leave the value 0 at 0. Or it is used stand-alone to effectively perform the operation $\text{XLEN} - i$ to calculate the shift amount for the 2nd shift in a funnel shift (see Section 5.5, XLEN does not actually need to be added because the shift operations only uses the lower $log_2(\text{XLEN})$ bits of their 2nd argument anyways). In those cases there is no addition before or after the `c.neg` instruction.

Further evaluation will show which of those instructions are worth their cost in compressed encoding space and decoder logic.

The compressed instructions use weird encodings. Is this really necessary?

This criticism is justified.

1. `c.neg` contains $\texttt{rd}'/\texttt{rs2}'$ instead of $\texttt{rd}'/\texttt{rs1}'$.
2. All three instructions use an encoding that can not be determined by looking only at the bits 15:13, and 1:0. This complicates decoder logic and decoder logic depth.

One possible solution would be to have only one compressed instruction (`c.not` is the obvious candidate). This instruction would occupy the entire reserved encoding space in `c.lui/c.addi16sp`

RV32		RV64		Instruction
3x	0	6x	0	`clz, clzw, ctz, ctzw, pcnt, pcntw`
3x	15	3x	15	`grev, shfl, unshfl`
2x	15	2x	16	`grevi, shfli/unshfli`
2x	15	6x	15	`slo, sro, slow, srow, sloiw, sroiw`
2x	15	2x	16	`sloi, sroi`
2x	15	5x	15	`ror, rol, rorw, rolw, roriw`
1x	15	1x	16	`rori`
3x	15	3x	15	`andc, bext, bdep`
		2x	15	`bextw, bdepw`
3x	4	3x	4	`c.neg, c.not, c.brev`

Table 4.1: Bitmanip encoding space ($log2$, i.e. in equivalent number of bits)

and would operate on `rd/rs` instead of `rd′/rs′`. This way the instruction format of `c.not` would match that of `c.addi16sp` and `c.lui`.

Another solution would be to use the reserved `c.addi4spn` $nzuimm = 0$ encoding with rd′=0 still being an illegal instruction.

4.2 Analysis of used encoding space

Table 4.1 lists all Bitmanip instructions and the encoding space needed to implement them. We do not count any encoding space for the unary instructions `clz`, `clzw`, `ctz`, `ctzw`, `pcnt`, and `pcntw` because they can be implemented in the reserved modes in `gzip`.

The compressed encoding space is ≈ 15.6 bits wide.

$$log_2(3 \cdot 2^{14}) \approx 15.585$$

The compressed Bitmanip instructions need the equivalent of a 4.6 bit encoding space, or ≈ 0.05% of the total ≈ 15.6 bits available.

$$log_2(3 \cdot 2^{3}) \approx 4.585$$

$$100/(2^{15.585-4.585}) \approx 0.049$$

The reserved "C" encoding space as of Version 2.2 of the RISC-V User-Level ISA is summarized in Table 4.2. (This is not including RV32-only or RV64-only reserved encoding space.) According to this information the reserved space is ≈ 11.2 bits wide. Therefore the compressed Bitmanip instructions would use ≈ 1.1% of the remaining reserved "C" encoding space.

$$log_2(2^3 - 1 + 2^{11} + 2^5 + 2^7 + 2^6 + 1) \approx 11.155$$

$$100/(2^{11.155-4.585}) \approx 1.053$$

15 14 13	12	11 10	9 8 7	6	5	4 3 2	1 0	
000	0					nz	00	2^3-1
100	—						00	2^{11}
011	0	—		0			01	2^5
100	111		—	1	—		01	2^7
010	—	0		—			10	2^6
100	0	0		0			10	1

Table 4.2: Reserved "C" encoding space as of Version 2.2 of the RISC-V User-Level ISA

On RV32, Bitmanip requires the equivalent of a ≈ 18.9 bit encoding space in the uncompressed encoding space. For comparison: A single standard I-type instruction (such as `ADDI` or `SLTIU`) requires a 22 bit encoding space. I.e. the entire RV32 Bitmanip extension needs less than one-eighth of the encoding space of the `SLTIU` instruction.

$$log_2(15 \cdot 2^{15}) \approx 18.907$$

On RV64, Bitmanip requires the equivalent of a ≈ 19.9 bit encoding space in the uncompressed encoding space. I.e. the entire RV64 Bitmanip extension needs about one-quarter of the encoding space of the `SLTIU` instruction.

$$log_2(19 \cdot 2^{15} + 5 \cdot 2^{16}) \approx 19.858$$

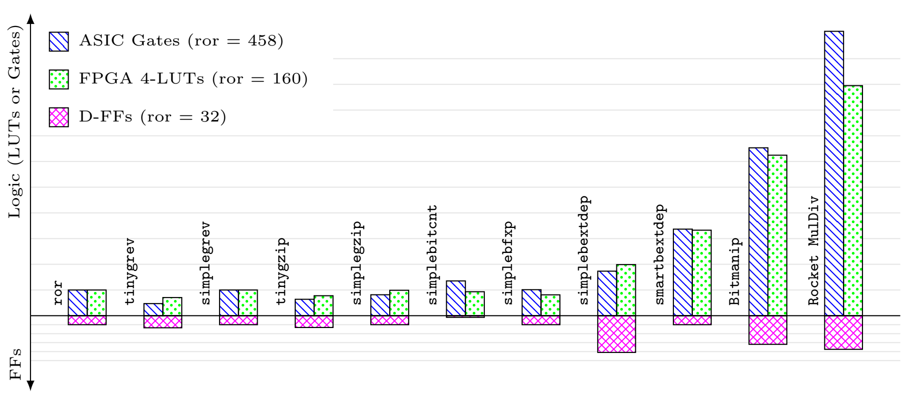


Figure 4.1: Relative area of Bitmanip reference cores compared to simple 32-bit rotate shift and Rocket MulDiv. The height of 1 LUT $\hat{=}$ 2.86 Gates, 1 FF $\hat{=}$ 5.00 Gates.

4.3 Area usage of reference implementations

We created RV32 implementations for the different compute cores necessary to implement Bitmanip (and XBitfield): `https://github.com/riscv/riscv-bitmanip/tree/master/verilog`

We are comparing the area of those cores with the following two references:

1. A very basic right-rotate shift core (`ror`):

```
module reference_ror (
    input clock,
    input [31:0] din,
    input [4:0] shamt,
    output reg [31:0] dout
);
    always @(posedge clock)
        dout <= {din, din} >> shamt;
endmodule
```

2. A version of the Rocket RV32 `MulDiv` core that performs multiplication in 5 cycles and division/modulo in 32+ cycles. This is the `MulDiv` default configuration used in Rocket for small cores.

The implementations of cores performing Bitmanip (and XBitfield) operations are:

- `tinygrev` — A small multi-cycle implementation of `grev`/`grevi`. It takes 6 cycles for one operation.
- `tinygzip` — A small multi-cycle implementation of `shfl/unshfl/shfli/unshfli`. It takes 5 cycles for one operation.
- `simplegrev` — A simple single-cycle implementation of `grev`/`grevi`.
- `simplegzip` — A simple single-cycle implementation of `shfl/unshfl/shfli/unshfli`.
- `simplebitcnt` — A simple single-cycle implementation of `clz`, `ctz`, and `pcnt`.
- `simplebfxp` — A simple single-cycle implementation of the XBitfield `bfxp` instruction (see Chapter **??**). It requires an external rotate-shift implementation.
- `simplebextdep` — A straight-forward multi-cycle implementation of `bext/bdep`. It takes up to 32 cycles for one operation (number of set mask bits).
- `smartbextdep` — A single-cycle implementation of `bext/bdep` using the method described in [11].

We implemented those cores with an ASIC cell library containing NOT, NAND, NOR, AOI3, OAI3, AOI4, and OAI4 gates. For the gate counts below we count NAND and NOR as 1 gate, NOT as 0.5 gates, AOI3 and OAI3 as 1.5 gates, and AOI4 and OAI4 as 2 gates.

We also implemented the cores using a simple FPGA architecture with 4-LUTs (and no dedicated CARRY or MUX resources).

Module	Gates	Depth	4-LUTs	Depth	FFs
`ror`	458	7	160	5	32
`tinygrev`	215	5	113	3	43
`simplegrev`	458	7	160	5	32
`tinygzip`	292	6	124	4	42
`simplegzip`	373	6	158	4	32
`simplebitcnt`	619	42	149	13	6
`simplebfxp`	463	14	130	6	32
`simplebextdep`	794	35	318	13	131
`smartbextdep`	1542	28	532	10	32
`Bitmanip`	2992	42	999	13	102
`Rocket MulDiv`	5068	49	1432	24	120

Table 4.3: Area and logic depth of Bitmanip reference cores compared to simple 32-bit rotate shift and Rocket MulDiv. The entry `Bitmanip` is just the total of `simplegrev` + `simplegzip` + `simplebitcnt` + `smartbextdep`. Not included in `Bitmanip` is the cost for adding shift-ones and rotate-shift support to the existing ALU shifter and the cost for additional decode and control logic.

Table 4.3 and Figure 4.1 show the area and logic depth of our reference cores (and the simple 32-bit rotate shift and Rocket MulDiv for comparison).

`smartbextdep` could share resources with `simplegrev` (it contains two butterfly circuits) and `simplebitcnt` (it contains a prefix adder network). We have not explored those resource sharing options in our reference cores.

Chapter 5

Evaluation

IMPORTANT NOTE: Some of the discussions below refer to an older draft of the RISC-V Bitmanip extension and are now out-of-date.

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.

Most assembler routines in this chapter are written as if they were ABI functions, i.e. arguments are passed in a0, a1, ... and results are returned in a0. Registers a0, a1, ... are also used for spilling Registers t0, t1, ... are used for pre-computed masks to be used with `bext`/`bdep`.

Some of the assembler routines below can not or should not overwrite their first argument. In those cases the arguments are passed in a1, a2, ... and results are returned in a0.

The main motivation behind this chapter is to show that all the common bit manipulation tasks can be performed in few instructions using Bitmanip. (In many cases RV32I/RV64I instructions are already sufficient.) In most cases the sequences are short enough to allow large cores to macro-op-fuse them into a single instruction. For this reason we also focus on code snippets that do not spill any registers, as this further simplifies macro-op fusion.

There likely will be a separate RISC-V standard for recommended sequences for macro-op fusion. The macros listed here are merely for demonstrating that suitable sequences exist. We do not advocate for any of those sequences to become "standard sequences" for macro-op fusion.

5.1 Bitfield extract

Extracting a bit field of length `len` at position `pos` can be done using two shift operations (or equivalently using the `bfext` pseudo instruction, see Section **??**).

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

5.2 MIX/MUX pattern

A MIX pattern selects bits from `a0` and `a1` based on the bits in the control word `a2`.

```
and a0, a0, a2
andc a1, a1, a2
or a0, a0, a1
```

A MUX operation selects word `a0` or `a1` based on if the control word `a2` is zero or nonzero, without branching.

```
snez a2, a2
neg a2, a2
and a0, a0, a2
andc a1, a1, a2
```

Or when `a2` is already either 0 or 1:

```
neg a2, a2
and a0, a2
andc a1, a1, a2
or a0, a0, a1
```

Alternatively, a core might fuse a conditional branch that just skips one instruction with that instruction to form a fused conditional macro-op.

A core with support for ternary instructions provides dedicated instructions for those operations: `cmix` and `cmov`. All examples below use those dedicated instructions. For cores without support for ternary operations, those instances of `cmix` and `cmov` must be replaced by the above code patterns.

5.3 Bit scanning and counting

Counting leading ones:

```
not a0, a0
clz a0, a0
```

Counting trailing ones:

```
not a0, a0
ctz a0, a0
```

Counting bits cleared:

```
not a0, a0
pcnt a0, a0
```

(This is better than XLEN-pcnt because RISC-V has no “reverse-subtract-immediate” operation.)

Odd parity:

```
pcnt a0, a0
andi a0, a0, 1
```

Even parity:

```
pcnt a0, a0
addi a0, a0, 1
andi a0, a0, 1
```

(Using `addi` here is better than using `xori`, because there is a compressed opcode for `addi` but none for `xori`.)

5.4 Test, set, and clear individual bits

Extracting bit N:

```
srli a0, a0, N
andi a0, a0, 1
```

Branching on bit N set:

```
slli a0, a0, (XLEN-N-1)
bltz a0, <bit_n_set>
```

Branching on bit N clear:

```
slli a0, a0, (XLEN-N-1)
bgez a0, <bit_n_clear>
```

Setting bit N (note that this `li` is only one instruction on RV32, except when N=11 which requires two instructions):

```
li a1, (1 << N)
or a0, a0, a1
```

Setting bit N without spilling:

```
rori a0, a0, N
ori a0, a0, 1
rori a0, a0, 32-N
```

(Or simply "`ori a0, 1 << N`" if N is sufficiently small.)

Clearing bit N (note that this `li` is only one instruction on RV32, except when N=11 which requires two instructions):

```
li a1, (1 << N)
andc a0, a0, a1
```

Clearing bit N without spilling:

```
rori a0, a0, N
andi a0, a0, -2
rori a0, a0, XLEN-N
```

(Or simply "`andi a0, ~(1 << N)`" if N is sufficiently small.)

Setting bit N to the value in `a1` (assuming a1 already is either 0 or 1):

```
rori a0, a0, N
andi a0, a0, -2
or a0, a0, a1
rori a0, a0, XLEN-N
```

5.5 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.

5.5.1 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 $\geq$XLEN can easily be built with `fsl` and `fsr`.

5.5.2 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.

5.5.3 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, N, a0, a2
  ret
```

5.6 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).

5.6.1 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 $log_2(\text{XLEN})$ butterfly operations.

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

5.6.2 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.

5.6.3 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.

5.6.4 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]

5.7 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.

5.7.1 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
```

5.7.2 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
```

5.7.3 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 $\mathtt{i}'$`[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 $\mathtt{i'[5:0]} = \mathtt{i[5:0]} + N \pmod{64}$.

The GREV operation with shamt N is $\mathtt{i'[5:0]} = \mathtt{i[5:0]}$ 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:

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

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

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

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.

5.7.4 Rotating Bitcubes

Let's define a bitcube as a $4 \times 4 \times 4$ cube with $x = \mathtt{i[1:0]}$, $y = \mathtt{i[3:2]}$, and $z = \mathtt{i[5:4]}$. 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
```

5.8 Rank and select

Rank and select are fundamental operations in succinct data structures [17].

`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
```

5.9 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. [16, 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)`.

5.10 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 5 instructions:

```
uint64_t rfill_brev(uint64_t x)
{
  x = brev(x);         // GREVI
  x = (x - 1) & ~x;    // ADDI, ANDC
  x = ~x;              // NOT
  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;
}
```

5.11 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.

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:5]				imm[4:0]		S-type
imm[12|10:5]				imm[4:1|11]		B-type
imm[20|10:1|11|19:12]						J-type

15 14 13	12 11 10 9 8 7 6 5 4 3 2	1 0	
	imm[11|4|9:8|10|6|7|3:1|5]		CJ-type

```
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
```

Or using XBitfield:

```
decode_s:
  bfxp a0, a1, zero, 7, 5, 20
  bfxp a0, a1, a0, 25, 7, 25
  c.srai a0, 20
  ret

decode_b:
  bfxp a0, a1, zero, 7, 1, 30
  bfxp a0, a1, a0, 25, 6, 24
  bfxp a0, a1, a0, 8, 4, 20
  bfxp a0, a1, a0, 31, 1, 31
  c.srai a0, 19
  ret

decode_j:
  bfxp a0, a1, zero, 21, 10, 12
  bfxp a0, a1, a0, 20, 1, 22
  bfxp a0, a1, a0, 12, 8, 23
  bfxp a0, a1, a0, 31, 1, 31
  c.srai a0, 11
  ret

decode_cj:
  bfxp a0, a1, zero, 11, 1, 24
  bfxp a0, a1, a0, 9, 2, 28
  bfxp a0, a1, a0, 8, 1, 30
  bfxp a0, a1, a0, 7, 1, 26
  bfxp a0, a1, a0, 6, 1, 27
  bfxp a0, a1, a0, 3, 3, 21
  bfxp a0, a1, a0, 2, 1, 25
  bfxp a0, a1, a0, 12, 1, 31
  c.srai a0, 20
  ret
```

Chapter 6

Comparison with othert ISAs

IMPORTANT NOTE: Some of the discussions below refer to an older draft of the RISC-V Bitmanip extension and are now out-of-date.

6.1 Comparison with x86 Bit Manipulation ISAs

The following code snippets implement all instructions from the x86 bit manipulation ISA extensions ABM, BMI1, BMI2, and TBM using RISC-V code that does not spill any registers and thus could easily be implemented in a single instruction using macro-op fusion. (Some of them simply map directly to instructions in this spec and so no macro-op fusion is needed.) Note that shorter RISC-V code sequences are sometimes possible if we allow spilling to temporary registers.

ABM added x86 encodings for POPCNT, LZCNT, and TZCNT.[1] The difference between LZCNT and the 80386 instruction BSR, and between TZCNT and the 80386 instruction BSF, is that the new instructions return the operand size when the input operand is zero, while BSR and BSF were undefined in that case. The ABM instructions map 1:1 to Bitmanip instructions. Table 6.1 lists ABM instructions and regular x86 bit manipulation instructions.

BMI1 adds some instructions for trailing bit manipulations, an add-complement instruction, and a bit field extract instruction that expects the length and start position packed in one register operand. Our version expects the length in a0, start position in a1, and source value in a2. See Table 6.2.

BMI2 adds a few `*X` instructions that just perform the indicated operation without changing any flags. RISC-V does not use flags, so this instructions trivially just map to their regular RISC-V counterparts. In addition to those instructions, BMI2 adds bit extract and deposit instructions and an instruction to clear high bits above a given bit index. See Table 6.3.

[1] Depending on if you ask Intel or AMD you will get different opinion regarding whether LZCNT and/or TZCNT

x86 Instruction	Bytes x86	Bytes RV	RISC-V Code
POPCNT	5	4	`pcnt a0, a0`
LZCNT / BSR	5	4	`clz a0, a0`
TZCNT / BSF	5	4	`ctz a0, a0`
BSWAP	3	4	`bswap`
ROL	4	4	`roli`
ROR	4	4	`rori`
BT	5	4	`c.srl a0, N` `c.andi a0, 1`

BTC	5	16	`rori a0, N`
			`andi a1, a0, 1`
			`xori a0, a0, 1`
			`rori a0, XLEN-N`

BTR	5	16	`rori a0, N`
			`andi a1, a0, 1`
			`andi a0, a0, -2`
			`rori a0, XLEN-N`

BTS	5	16	`rori a0, N`
			`andi a1, a0, 1`
			`ori a0, a0, 1`
			`rori a0, XLEN-N`

Table 6.1: Comparison of x86+ABM with Bitmanip

Finally, TBM was a short-lived x86 ISA extension introduced by AMD in Piledriver processors, complementing the trailing bit manipulation instructions from BMI1. See Table 6.4.

6.2 Comparison with RI5CY Bit Manipulation ISA

The following section compares Bitmanip with the RI5CY bit manipulation instructions as documented in [8]. All RI5CY bit manipulation instructions (or something very close to their behavior) can be emulated with Bitmanip using 3 instructions or less.

RI5CY Instructions `p.extract`, `p.extractu`, `p.extractr`, and `p.extractur`

These four RI5CY instructions extract bit-fields. The non-`u`-versions sign-extend the extracted bit-field. This operations can be performed with two shift-immediate operations. It even fits in a 32-bit word when using compressed instructions (requires `rd` = `rs`).

```
p.extract rd, rs, len, pos:
  slli rd, rs, (XLEN-pos-len)
```

are part of ABM or BMI1.

x86 Instruction	Bytes x86	Bytes RV	RISC-V Code
ANDN	5	4	`andc a0, a2, a1`

BEXTR (regs)	5	12	`c.add a0, a1`
			`slo a0, zero, a0`
			`c.and a0, a2`
			`srl a0, a0, a1`

BLSI	5	6	`neg a0, a1` `c.and a0, a1`
BLSMSK	5	6	`addi a0, a1, -1` `c.xor a0, a1`
BLSR	5	6	`addi a0, a1, -1` `c.and a0, a1`

Table 6.2: Comparison of x86 BMI1 with Bitmanip

x86 Instruction	Bytes x86	Bytes RV	RISC-V Code
BZHI	5	6	`slo a0, zero, a2` `c.and a0, a1`
PDEP	5	4	`bdep`
PEXT	5	4	`bext`
MULX	5	4	`mul`
RORX	6	4	`rori`
SARX	5	4	`sra`
SHRX	5	4	`srl`
SHLX	5	4	`sll`

Table 6.3: Comparison of x86 BMI2 with Bitmanip

```
  srai rd, rd, (XLEN-len)

p.extractu rd, rs, len, pos:
  slli rd, rs, (XLEN-pos-len)
  srli rd, rd, (XLEN-len)
```

The `r`-versions expect the bit-field size in bits 9:5 of the second source register and the bit-field start in bits 4:0. Instead we use two registers, `rx` $= XLEN - pos - len$ and `ry` $= XLEN - len$.

```
p.extractr:
  sll rd, rs, rx
  sra rd, rd, ry

p.extractur:
  sll rd, rs, rx
  srl rd, rd, ry
```

x86 Instruction	Bytes x86	Bytes RV	RISC-V Code
BEXTR (imm)	7	4	`c.slli a0, (32-START-LEN)` `c.srli a0, (32-LEN)`
BLCFILL	5	6	`addi a0, a1, 1` `c.and a0, a1`

BLCI	5	8	`addi a0, a1, 1`
			`c.not a0`
			`c.or a0, a1`

BLCIC	5	10	`addi a0, a1, 1`
			`andc a0, a1, a0`
			`c.not a0`

BLCMSK	5	6	`addi a0, a1, 1` `c.xor a0, a1`
BLCS	5	6	`addi a0, a1, 1` `c.or a0, a1`
BLSFILL	5	6	`addi a0, a1, -1` `c.or a0, a1`

BLSIC	5	10	`addi a0, a1, -1`
			`andc a0, a1, a0`
			`c.not a0`

T1MSKC	5	10	`addi a0, a1, +1`
			`andc a0, a1, a0`
			`c.not a0`

T1MSK	5	8	`addi a0, a1, -1` `andc a0, a0, a1`

Table 6.4: Comparison of x86 TBM with Bitmanip

Alternatively, instead of packing length and position into a register, we can create a mask in a register and then use this mask with `bext`. This has the advantage over the `sll`+`srl` sequence that the mask only needs to be generated once and can then be re-used many times, effectively implementing `p.extractur` in a single instruction.

```
p.extractur:
  slo rMask, zero, rLen
  sll rMask, rMask, rPos
  bext rd, rs, rMask
```

`p.extractu` can be efficiently emulated with a single XBitfield `bfxp` instruction (see Chapter ??):

```
p.extract rd, rs, len, pos:
  bfxp rd, rs, zero, pos, len, 0
```

RI5CY Instructions `p.insert` and `p.insertr`

These instructions OR the destination register with the `len` LSB bits from the source register, shifted up by `pos` bits. This can easily be achieved using three instructions and a temporary register `rt`:

```
p.insert rd, rs, len, pos, rt:
  slli rt, rs, (XLEN-len)
  srli rt, rt, (XLEN-len-pos)
  or rd, rd, rt
```

The `r`-version of the instruction expects `len` in bits 9:5 of the second source register and `pos` in bits 4:0. Instead we use two registers, `rx` $= XLEN - pos - len$ and `ry` $= XLEN - len$.

```
p.insertr:
  slli rt, rs, ry
  srli rt, rt, rx
  or rd, rd, rt
```

Alternatively, instead of packing length and position into a register, we can create a mask in a register and then use this mask with `bdep`. This has the advantage over the `sll`+`srl` sequence that the mask only needs to be generated once and can then be re-used many times.

```
p.extractur:
  slo rMask, zero, rLen
  sll rMask, rMask, rPos
  bdep rt, rs, rMask
  or rd, rd, rt
```

`p.insert` can be efficiently emulated with a single XBitfield `bfxp` instruction (see Chapter **??**), if target region in `rd` contains only zeros (`bfxp` clears the target region before performing the OR):

```
p.insert rd, rs, len, pos:
  bfxp rd, rs, rd, 0, len, pos
```

RI5CY Instructions `p.bclr` and `p.bclrr`

These instructions clear `len` bits starting with bit `pos`. Using a temporary register `rt`:

```
p.bclr rd, rs, len, pos, rt:
  sloi rt, zero, len
  slli rt, rt, pos
  andc rd, rs, rt
```

Or using two registers `rLen` and `rPos`:

```
p.bclrr rd, rs, rLen, rPos, rt:
  slo rt, zero, rLen
  sll rt, rt, rPos
  andc rd, rs, rt
```

If the mask in `rt` can be pre-computed then a single `andc` instruction can emulate `p.bclrr`, or a single `and`/`c.and` instruction if the mask is already inverted.

Or using `bfxp` with `rd` = `rs` (see Chapter **??**):

```
p.bclr rd, len, pos:
  bfxp rd, zero, rd, 0, len, pos
```

RI5CY Instructions `p.bset` and `p.bsetr`

These instructions set `len` bits starting with bit `pos`. They can be implemented similar to `p.bclr` and `p.bclrr` by replacing `andc` with `or`:

```
p.bset rd, rs, len, pos, rt:
  sloi rt, zero, len
  slli rt, rt, pos
  or rd, rs, rt

p.bsetr rd, rs, rLen, rPos, rt:
  slo rt, zero, rLen
  sll rt, rt, rPos
  or rd, rs, rt
```

If the mask in `rt` can be pre-computed then a single `or`/`c.or` instruction can emulate `p.bsetr`.

Or using `bfxpc` with `rd` = `rs` (see Chapter **??**):

```
p.bset rd, len, pos:
  bfxpc rd, zero, rd, 0, len, pos
```

RI5CY Instructions `p.ff1`, `p.cnt`, and `p.ror`

These instructions map directly to the Bitmanip instructions `ctz`, `pcnt`, and `ror`.

RI5CY Instructions `p.fl1`

This instruction returns the index of the last set bit in `rs`. If `rs` is 0, `rd` will be 0.

Using the arguably more useful definition that the operation should return -1 when `rs` is 0:

```
p.fl1 rd, rs:
  clz rd, rs
  neg rd, rd
  addi rd, rd, 31
```

Converting a -1 result to 0 to match the exact `p.fl1` behavior:

```
  slt rt, rd, zero
  add rd, rd, rt
```

RI5CY Instructions `p.clb`

This instruction counts the number of consecutive 1s or 0s from MSB. If `rs` is 0, `rd` will be 0.

Using the arguably more useful definition that the operation should return XLEN when `rs` is 0 or -1, and assuming `rd` $\neq$ `rs1`:

```
p.clb:
  srai rd, rs, XLEN-1
  xor rd, rd, rs
  clz rd, rd
```

Simply add `andi rd, rd, XLEN-1` if `rd` should be 0 when `rs` is 0 or -1.

6.3 Comparison with Cray XMT bit operations

Cray XMT is the 3rd generation of the Cray MTA architecture, a supercomputer using a barrel processor architecture. The Cray XMT instruction set contains a few bit manipulation instructions [5]. In this section we compare the Cray XMT bit manipulation instructions with Bitmanip.

Bitwise boolean operations

Cray XMT provides the following instructions for bitwise boolean operations: `BIT_AND`, `BIT_IMP`, `BIT_NAND`, `BIT_NIMP`, `BIT_NOR`, `BIT_OR`, `BIT_XNOR`, and `BIT_XOR`.

These can trivially emulated using basic RISC-V instructions. (Some of those XMT instructions have a direct RISC-V equivalent. The others can be emulated by combining the `not` pseudoinstruction with `and`, `or`, or `xor`.)

Count Leading/Trailing Zeros/Ones

The Cray XMT instructions `BIT_LEFT_ZEORS` and `BIT_RIGHT_ZEORS` are equivalent to the Bitmanip `clz` and `ctz` instructions.

The `BIT_LEFT_ONES` and `BIT_RIGHT_ONES` instructions can be emulated by combining the `not` pseudoinstruction with `clz` and `ctz`.

Mask generation

The Cray XMT instruction `BIT_MASK dest top bot` generates a bitmask that has the bits in the range [`bot` ... `top`] set and the rest cleared iff `bot` $\leq$ `top`, and those bits cleared and the rest set otherwise.

Similar masks can be generated using two instructions in Bitmanip, using the regular shift instructions and the "shift ones" instructions.

Cmix equivalent

The Cray XMT `BIT_MERGE` instruction is equivalent to the XTernarybits `cmix` instruction, or the `and-andc-or` MIX pattern.

Population count

The Cray XMT `BIT_TALLY` instruction and the Bitmanip `pcnt` instruction are equivalent.

Parity instructions

The Cray XMT `BIT_ODD_AND`, `BIT_ODD_NIMP`, `BIT_ODD_OR`, and `BIT_ODD_XOR` instruction perform the indicated bitwise boolean operation and then compute the parity of the result.

With Bitmanip the parity can be calculated with `pcnt dst, src` followed by `andi dst, dst, 1`.

Bit pack/unpack instruction

The Cray XMT `BIT_PACK` instruction and the Bitmanip `bext` instruction are equivalent.

The Cray XMT `BIT_UNPACK_1`, `BIT_UNPACK_2`, `BIT_UNPACK_3`, instruction sequence, when used as intended, is equivalent to the Bitmanip `bdep` instruction.

Bit matrix instructions

The Cray XMT `BIT_MAT_` instructions treat a 64-bit value as a 8x8 bit matrix.

`BIT_MAT_TRANSPOSE` is used to transpose such a bit matrix. With Bitmanip instructions on RV64 this bit permutation can be performed by applying the `zip` operation three times to the register holding the matrix (see also 5.7.2).

The Cray XMT instructions `BIT_MAT_OR` and `BIT_MAT_XOR` perform a matrix-matrix multiply between such bit matrices, where AND replaces scalar multiply and OR/XOR replaces scalar addition.

Bitmanip does not provide a similar operation. However, the two example applications given in the Cray XMT documentation [5, p. 81] are reversing the byte order in a word and reversing the bit order in each byte. In Bitmanip those operations are performed by the `bswap` and the `brev.b` `grevi`-pseudoinstructions.

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`

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] Pasquale Davide Schiavone Andreas Traber, Michael Gautschi. Ri5cy: User manual. `https://pulp-platform.org//wp-content/uploads/2017/11/ri5cy_user_manual.pdf`. Accessed: 2017-04-26.

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

[10] 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.

[11] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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