RISC-V XBitmanip Extension

Document Version 0.36-draft

Editor: Clifford Wolf
Symbiotic GmbH
`clifford@symbioticeda.com`

May 12, 2018

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, Bruce Hoult, Po-wei Huang, Rex McCrary, and Clifford Wolf.

Contents

Chapter 1

Introduction

This is the RISC-V XBitmanip Extension draft spec. Originally it was the B-Extension draft spec, but the work group got dissolved for bureaucratic reasons in November 2017.

It is currently an independently maintained document. We'd happily donate it to the RISC-V foundation as starting point for a new B-Extension work group, if there will be one.

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 a *significant* savings in terms of clocks or instructions. As a heuristic, any proposal should replace at least four instructions. An instruction that only replaces three 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 XBitmanip Extension

In the proposals provided in this section, the C code examples are for illustration purposes. They are not optimal implementations, but are intended to specify the desired functionality. See Chapter 6 for fast C code for use in emulators.

The sections on encodings are mere placeholders.

2.1 Count Leading/Trailing Zeros (`clz`, `ctz`)

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;
}
```

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
????????????	src	CLZ	dest	OP-IMM
????????????	src	CTZ	dest	OP-IMM
????????????	src	CLZW	dest	OP-IMM-32
????????????	src	CTZW	dest	OP-IMM-32

One possible encoding for `clz` and `ctz` is as standard I-type opcodes somewhere in the brownfield surrounding the shift-immediate instructions.

2.2 Count Bits Set (`pcnt`)

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. [6, 5]

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

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
????????????	src	PCNT	dest	OP-IMM
????????????	src	PCNTW	dest	OP-IMM-32

One possible encoding for `pcnt` is as a standard I-type opcode somewhere in the brownfield surrounding the shift-immediate instructions.

2.3 And-with-complement (`andc`)

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` (`c.not`) with the base ALU operation. (Which can fit in 32 bit when using two compressed instructions.) Only and-with-complement occurs frequently enough to warrant a dedicated instruction.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
???????	src2	src1	ANDC	dest	OP

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

These instructions are similar to shift-logical operations from the base spec, except instead of shifting in zeros, they shifts 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);
}
```

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
10?????	src2	src1	SRO	dest	OP
10?????	src2	src1	SLO	dest	OP
10?????	src2	src1	SROW	dest	OP-32
10?????	src2	src1	SLOW	dest	OP-32

31 27	26 20	19 15	14 12	11 7	6 0
imm[11:7]	imm[6:0]	rs1	funct3	rd	opcode
5	7	5	3	5	7
10???	shamt	src	SLOI	dest	OP-IMM
10???	shamt	src	SROI	dest	OP-IMM

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
10?????	shamt	src	SLOIW	dest	OP-IMM-32
10?????	shamt	src	SROIW	dest	OP-IMM-32

`s(l/r)o(i)` is encoded similarly to the logical shifts in the base spec. However, the spec of the entire family of instructions is changed so that the high bit of the instruction indicates the value to be inserted during a shift. This means that a `sloi` instruction can be encoded similarly to an `slli` instruction, but with a 1 in the highest bit of the encoded instruction. This encoding is backwards compatible with the definition for the shifts in the base spec, but allows for simple addition of a ones-insert.

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.

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

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));
}

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

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
11?????	src2	src1	ROR	dest	OP
11?????	src2	src1	ROL	dest	OP
11?????	src2	src1	RORW	dest	OP-32
11?????	src2	src1	ROLW	dest	OP-32

31 27	26 20	19 15	14 12	11 7	6 0
imm[11:7]	imm[6:0]	rs1	funct3	rd	opcode
5	7	5	3	5	7
11???	shamt	src	RORI	dest	OP-IMM

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
11?????	shamt	src	RORIW	dest	OP-IMM-32

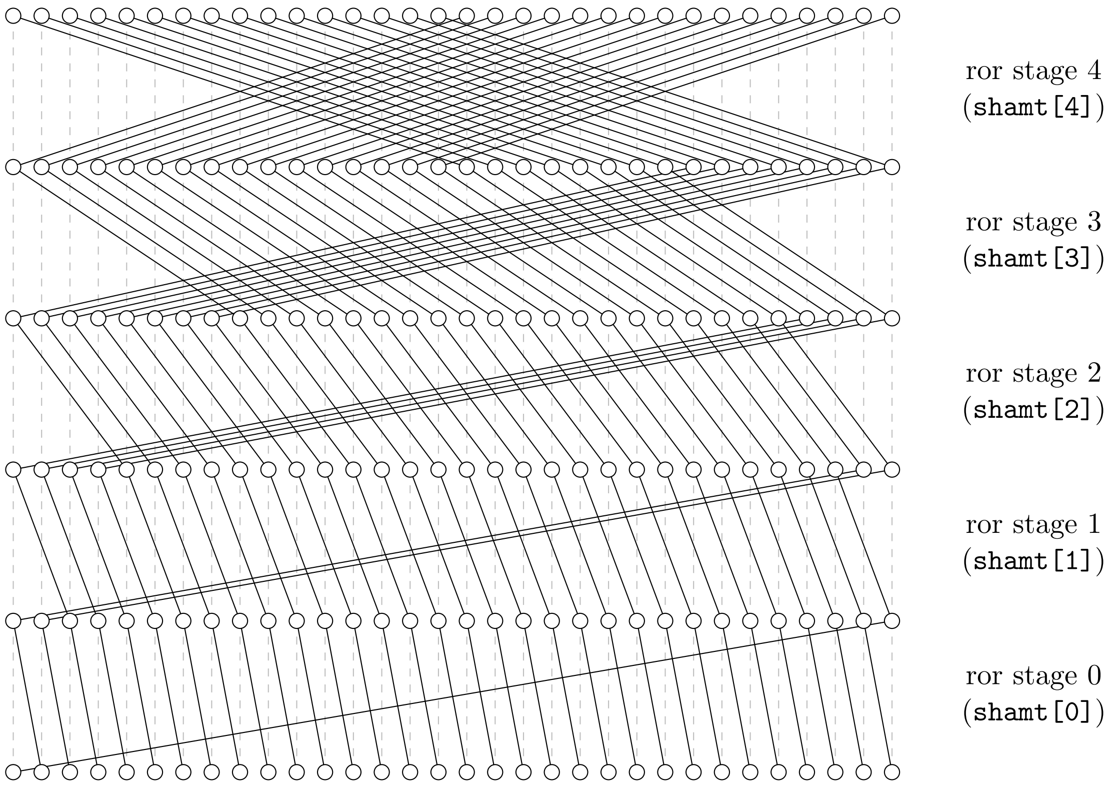

Figure 2.1: `ror` permutation network

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 31 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

2.6 Generalized Reverse (`grev, grevi`)

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

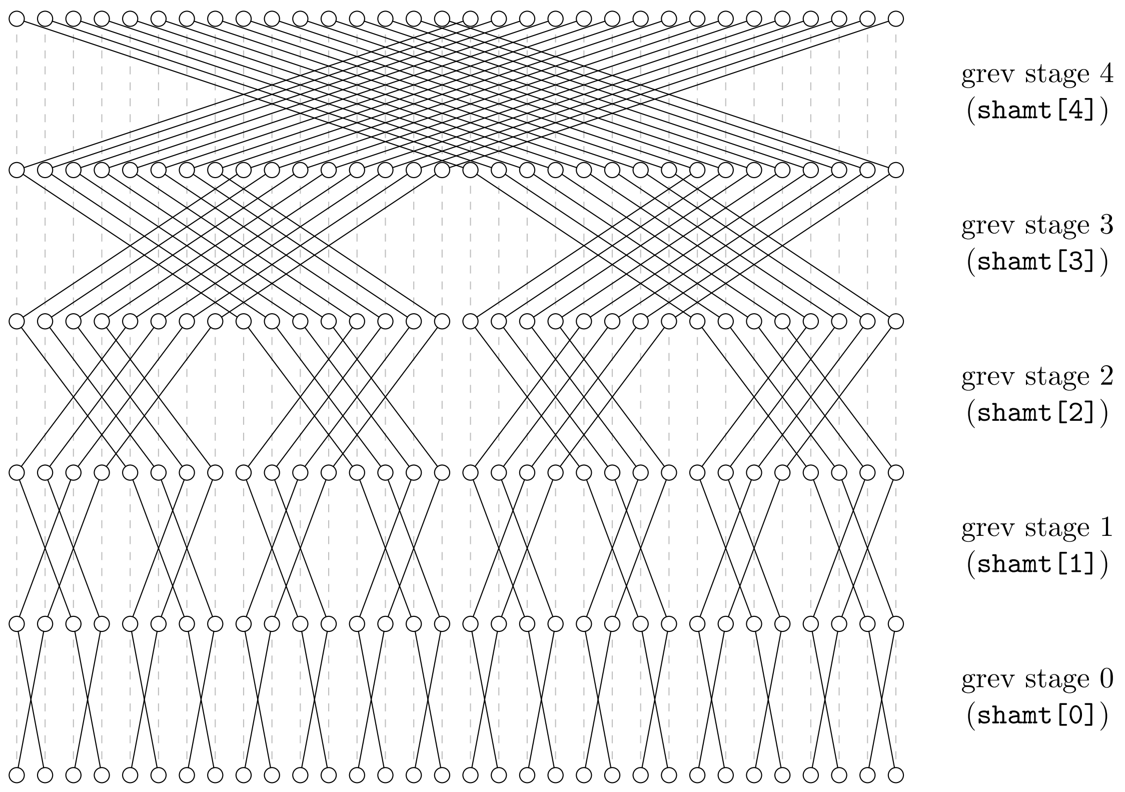

Figure 2.2: `grev` permutation network

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

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

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

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

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

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
???????	src2	src1	GREV	dest	OP

31 27	26 20	19 15	14 12	11 7	6 0
imm[11:7]	imm[6:0]	rs1	funct3	rd	opcode
5	7	5	3	5	7
?????	mode	src	GREVI	dest	OP-IMM

`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".

2.7 Generalized zip/unzip (`gzip`)

`gzip` is the third bit permutation instruction in XBitmanip, after `rori` and `grevi`. 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.

This set of operation is best understood as operation on the bit indices. For reference, rotate shift adds `shamt` to the bit index (modulo XLEN), and generalized reversed performs an XOR between

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

the bit index and `shamt`. Generalized zip/unzip performs a roate left shift (zip) or rotate right shift (unzip) on a continous range of bits in the bit index. Every arbitrary permutation of the bit index bits can be achieved by applying `gzip` repeatedly.

The `gzip` instruction uses an I-type encoding similar to `grevi`. There are XLEN different generalized zip operations, some of which are reserved because they are no-ops, or equivalent to other modes, or encode for obscure combinations of other modes. The bit pattern for the non-reserved modes match the regular expression `/^0*(10+|11+0*[01])$/`, or in words: Bits `mode[LOG2_XLEN-1:1]` must only contain one continous range of 1 bits, and if only one bit in `mode[LOG2_XLEN-1:1]` is set then `mode[0]` must be cleared. See Tables 2.2 and 2.3.

Reserving modes that encode for "obscure combinations of other modes" can help implementations that use different base permutations (or completely different mechanisms) to implement the `gzip` instruction. The reserved modes can be used to encode unary functions such as `ctz`, `clz`, and `pcnt`.

mode	Bit index rotations	Pseudo-Instruction
~~0: 0000 0~~	no-op	*reserved*
~~1: 0000 1~~	no-op	*reserved*
2: 0001 0	`i[1] -> i[0]`	`zip.n, unzip.n`
~~3: 0001 1~~	*equivalent to 0001 0*	*reserved*
4: 0010 0	`i[2] -> i[1]`	`zip2.b, unzip2.b`
~~5: 0010 1~~	*equivalent to 0010 0*	*reserved*
6: 0011 0	`i[2] -> i[0]`	`zip.b`
7: 0011 1	`i[2] <- i[0]`	`unzip.b`
8: 0100 0	`i[3] -> i[2]`	`zip4.h, unzip4.h`
~~9: 0100 1~~	*equivalent to 0100 0*	*reserved*
~~10: 0101 0~~	`i[3] -> i[2], i[1] -> i[0]`	*reserved*
~~11: 0101 1~~	*equivalent to 0101 0*	*reserved*
12: 0110 0	`i[3] -> i[1]`	`zip2.h`
13: 0110 1	`i[3] <- i[1]`	`unzip2.h`
14: 0111 0	`i[3] -> i[0]`	`zip.h`
15: 0111 1	`i[3] <- i[0]`	`unzip.h`
16: 1000 0	`i[4] -> i[3]`	`zip8, unzip8`
~~17: 1000 1~~	*equivalent to 1000 0*	*reserved*
~~18: 1001 0~~	`i[4] -> i[3], i[1] -> i[0]`	*reserved*
~~19: 1001 1~~	*equivalent to 1001 0*	*reserved*
~~20: 1010 0~~	`i[4] -> i[3], i[2] -> i[1]`	*reserved*
~~21: 1010 1~~	*equivalent to 1010 0*	*reserved*
~~22: 1011 0~~	`i[4] -> i[3], i[2] -> i[0]`	*reserved*
~~23: 1011 1~~	`i[4] <- i[3], i[2] <- i[0]`	*reserved*
24: 1100 0	`i[4] -> i[2]`	`zip4`
25: 1100 1	`i[4] <- i[2]`	`unzip4`
~~26: 1101 0~~	`i[4] -> i[2], i[1] -> i[0]`	*reserved*
~~27: 1101 1~~	`i[4] <- i[2], i[1] <- i[0]`	*reserved*
28: 1110 0	`i[4] -> i[1]`	`zip2`
29: 1110 1	`i[4] <- i[1]`	`unzip2`
30: 1111 0	`i[4] -> i[0]`	`zip`
31: 1111 1	`i[4] <- i[0]`	`unzip`

Table 2.2: RV32 modes and pseudo-instructions for `gzip` instruction

Like GREV and rotate shift, the `gzip` instruction can be implemented using a short sequence of atomic permutations, that are enabled or disabled by the mode (shamt) bits. But zip has one stage fewer than GREV and the LSB bit of mode controls the order in which the stages are applied:

```
uint32_t gzip32_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;
}
```

mode	Pseudo-Instruction
~~0: 00000 0~~	*reserved*
~~1: 00000 1~~	*reserved*
2: 00001 0	`zip.n, unzip.n`
~~3: 00001 1~~	*reserved*
4: 00010 0	`zip2.b, unzip2.b`
~~5: 00010 1~~	*reserved*
6: 00011 0	`zip.b`
7: 00011 1	`unzip.b`
8: 00100 0	`zip4.h, unzip4.h`
~~9: 00100 1~~	*reserved*
~~10: 00101 0~~	*reserved*
~~11: 00101 1~~	*reserved*
12: 00110 0	`zip2.h`
13: 00110 1	`unzip2.h`
14: 00111 0	`zip.h`
15: 00111 1	`unzip.h`
16: 01000 0	`zip8.w, unzip8.w`
~~17: 01000 1~~	*reserved*
~~18: 01001 0~~	*reserved*
~~19: 01001 1~~	*reserved*
~~20: 01010 0~~	*reserved*
~~21: 01010 1~~	*reserved*
~~22: 01011 0~~	*reserved*
~~23: 01011 1~~	*reserved*
24: 01100 0	`zip4.w`
25: 01100 1	`unzip4.w`
~~26: 01101 0~~	*reserved*
~~27: 01101 1~~	*reserved*
28: 01110 0	`zip2.w`
29: 01110 1	`unzip2.w`
30: 01111 0	`zip.w`
31: 01111 1	`unzip.w`
32: 10000 0	`zip16, unzip16`
~~33: 10000 1~~	*reserved*
~~34: 10001 0~~	*reserved*
~~35: 10001 1~~	*reserved*
~~36: 10010 0~~	*reserved*
~~37: 10010 1~~	*reserved*
~~38: 10011 0~~	*reserved*
~~39: 10011 1~~	*reserved*
~~40: 10100 0~~	*reserved*
~~41: 10100 1~~	*reserved*
~~42: 10101 0~~	*reserved*
~~43: 10101 1~~	*reserved*
~~44: 10110 0~~	*reserved*
~~45: 10110 1~~	*reserved*
~~46: 10111 0~~	*reserved*
~~47: 10111 1~~	*reserved*
48: 11000 0	`zip8`
49: 11000 1	`unzip8`
~~50: 11001 0~~	*reserved*
~~51: 11001 1~~	*reserved*
~~52: 11010 0~~	*reserved*
~~53: 11010 1~~	*reserved*
~~54: 11011 0~~	*reserved*
~~55: 11011 1~~	*reserved*
56: 11100 0	`zip4`
57: 11100 1	`unzip4`
~~58: 11101 0~~	*reserved*
~~59: 11101 1~~	*reserved*
60: 11110 0	`zip2`
61: 11110 1	`unzip2`
62: 11111 0	`zip`
63: 11111 1	`unzip`

Table 2.3: RV64 modes and pseudo-instructions for `gzip` instruction

```
uint32_t gzip32(uint32_t rs1, uint32_t rs2)
{
    uint32_t x = rs1;
    int mode = rs2 & 31;

    if (mode & 1) {
        if (mode &  2) x = gzip32_stage(x, 0x44444444, 0x22222222, 1);
        if (mode &  4) x = gzip32_stage(x, 0x30303030, 0x0c0c0c0c, 2);
        if (mode &  8) x = gzip32_stage(x, 0x0f000f00, 0x00f000f0, 4);
        if (mode & 16) x = gzip32_stage(x, 0x00ff0000, 0x0000ff00, 8);
    } else {
        if (mode & 16) x = gzip32_stage(x, 0x00ff0000, 0x0000ff00, 8);
        if (mode &  8) x = gzip32_stage(x, 0x0f000f00, 0x00f000f0, 4);
        if (mode &  4) x = gzip32_stage(x, 0x30303030, 0x0c0c0c0c, 2);
        if (mode &  2) x = gzip32_stage(x, 0x44444444, 0x22222222, 1);
    }

    return x;
}
```

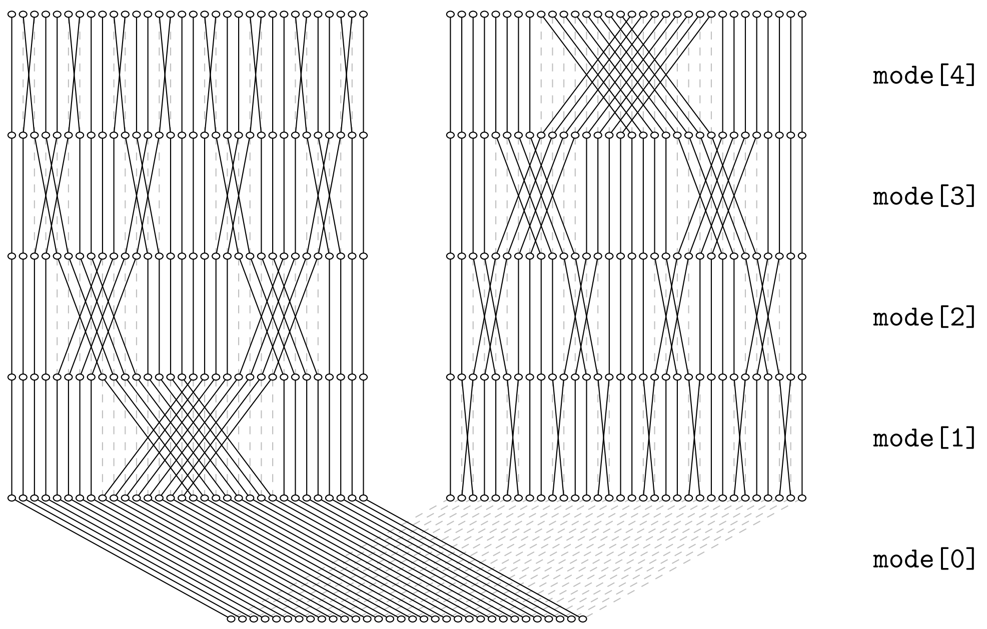

Figure 2.3: `gzip` permutation network without "flip" stages

Or for RV64:

```
uint64_t gzip64_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 gzip64(uint64_t rs1, uint64_t rs2)
{
    uint64_t x = rs1;
    int mode = rs2 & 63;

    if (mode & 1) {
        if (mode &  2) x = gzip64_stage(x, 0x4444444444444444LL,
                                           0x2222222222222222LL, 1);
        if (mode &  4) x = gzip64_stage(x, 0x3030303030303030LL,
                                           0x0c0c0c0c0c0c0c0cLL, 2);
        if (mode &  8) x = gzip64_stage(x, 0x0f000f000f000f00LL,
                                           0x00f000f000f000f0LL, 4);
        if (mode & 16) x = gzip64_stage(x, 0x00ff000000ff0000LL,
                                           0x0000ff000000ff00LL, 8);
        if (mode & 32) x = gzip64_stage(x, 0x0000ffff00000000LL,
                                           0x00000000ffff0000LL, 16);
    } else {
        if (mode & 32) x = gzip64_stage(x, 0x0000ffff00000000LL,
                                           0x00000000ffff0000LL, 16);
        if (mode & 16) x = gzip64_stage(x, 0x00ff000000ff0000LL,
                                           0x0000ff000000ff00LL, 8);
        if (mode &  8) x = gzip64_stage(x, 0x0f000f000f000f00LL,
                                           0x00f000f000f000f0LL, 4);
        if (mode &  4) x = gzip64_stage(x, 0x3030303030303030LL,
                                           0x0c0c0c0c0c0c0c0cLL, 2);
        if (mode &  2) x = gzip64_stage(x, 0x4444444444444444LL,
                                           0x2222222222222222LL, 1);
    }

    return x;
}
```

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

Alternatively `gzip` 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 gzip32_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 gzip32alt(uint32_t rs1, uint32_t rs2)
{
    uint32_t x = rs1;
    int mode = rs2 & 31;

    if (mode &  1)
        x = gzip32_flip(x);

    if ((mode & 17) == 16 || (mode &  3) ==  3)
        x = gzip32_stage(x, 0x00ff0000, 0x0000ff00, 8);

    if ((mode &  9) ==  8 || (mode &  5) ==  5)
        x = gzip32_stage(x, 0x0f000f00, 0x00f000f0, 4);

    if ((mode &  5) ==  4 || (mode &  9) ==  9)
        x = gzip32_stage(x, 0x30303030, 0x0c0c0c0c, 2);

    if ((mode &  3) ==  2 || (mode & 17) == 17)
        x = gzip32_stage(x, 0x44444444, 0x22222222, 1);

    if (mode &  1)
        x = gzip32_flip(x);

    return x;
}
```

Figure 2.4 shows the `gzip` permutation network with “flip” stages and Figure 2.3 shows the `gzip` 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
```

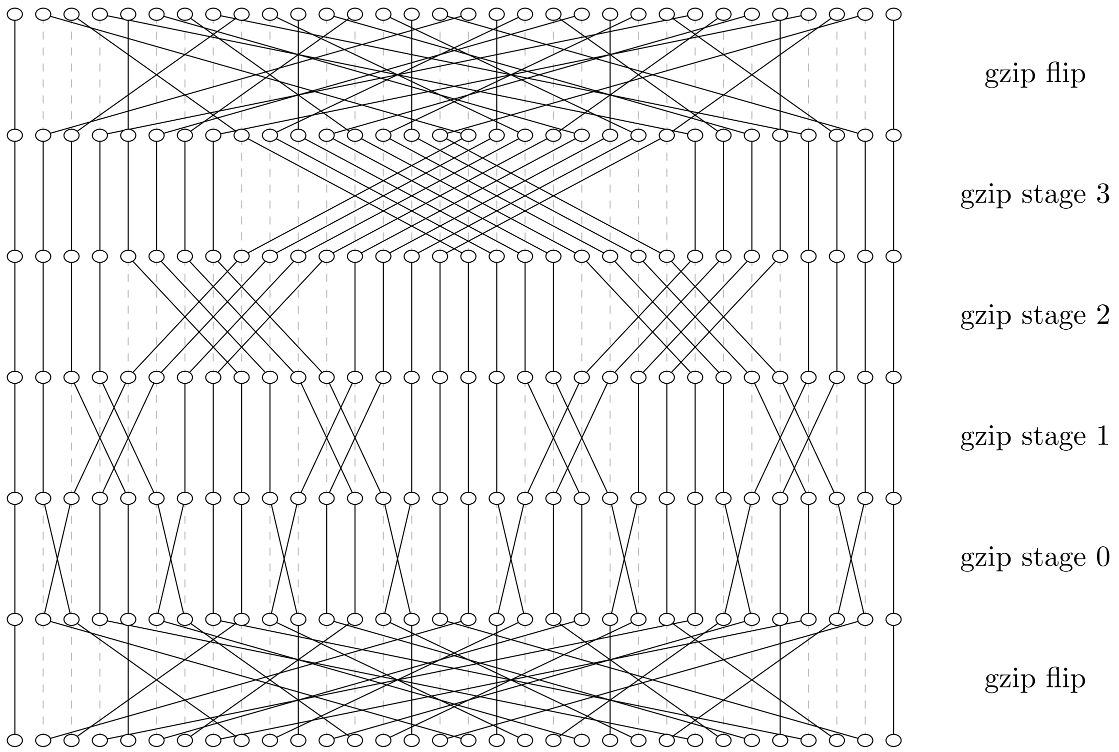

Figure 2.4: `gzip` permutation network with "flip" stages

```
slli a1, a0, 4
c.add a0, a1
slli a1, a0, 8
c.add a0, a1
slli a1, a0, 16
c.add a0, a1
```

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

Other `gzip` modes can be used to "fan-out" in blocks of 2, 4, 8, or 16 bit. `gzip` 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
```

31 27	26 20	19 15	14 12	11 7	6 0
imm[11:7]	imm[6:0]	rs1	funct3	rd	opcode
5	7	5	3	5	7
?????	mode	src	GZIP	dest	OP-IMM

There is no R-type instruction for `gzip`. It is an I-type only instruction. `gzip` can use the instruction encoding for "rotate left immediate".

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

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 algortihms 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.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
???????	src2	src1	BEXT	dest	OP
???????	src2	src1	BDEP	dest	OP
???????	src2	src1	BEXTW	dest	OP-32
???????	src2	src1	BDEPW	dest	OP-32

2.9 Compressed instructions (`c.not, c.neg, c.brev`)

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

In bitmanipulation code `not` and `neg` are very common operations. But there are no compressed encodings for those operations because there is no `c.xori` instruction and `c.sub` can not operate on `x0`.

Many bit manipulation operations that have dedicated opcodes in other ISAs must be constructed from smaller atoms in RISC-V XBitmanip code. But implementations might choose to implement them in a single micro-op using macro-op-fusion. For this it can be helpful when the fused sequences are short. `not` and `neg` are good candidates for macro-op-fusion, so it can be helpful to have compressed opcodes for them.

Likewise `brev` (an alias for `grevi rd, rs, -1`, i.e. bitwise reversal) is also a very common atom for building bit manipulation operations. So it is helpful to have a compressed opcode for this instruction as well.

The compressed instructions `c.not, c.neg, c.brev` must be supported by all implementations that support the C extension and XBitmanip.

These three instructions fit nicely in the reserved space in C.LUI/C.ADDI16SP. They only occupy 0.1% of the ≈ 15.6 bits wide RVC encoding space.

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	00	rs2′/rd′	0	01	C.NEG
011	0	01	rs1′/rd′	0	01	C.NOT
011	0	10	rs1′/rd′	0	01	C.BREV
011	0	11	—	0	01	*Reserved*

2.10 Bit-field extract pseudo instruction (`bfext`)

Extract the continous bit field starting at `pos` with length `len` from `rs` (with `pos` > 0, `len` > 0, and `pos` $+$ `len` $\leq$ XLEN).

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

If possible, an implementation should fuse this two shift operations into a single macro-op. (Some implementors have raised concerns about the lack of a dedicated bit field extract instruction with large immediate, especially for implementations that can not fuse instructions into macro-ops and/or implementations that do not support compressed instructions. See Chapter 5 for a brief discussion of why no such instruction is part of XBitmanip, and a short separate extension proposal that adds such an instruction.)

Chapter 3

Discussion

3.1 Frequently Asked Questions

Which instructions were considered but not included?

- A bit-field extract instruction. Unfortunately the encoding space required by such an instruction would be enourmous (for forward-looking support up to RV128 this would require a $2 = 14$ bits immedate. This was deemed too expensive considering that the same functionality can already be implemented using only two shift instructions. However, see Chapter 5 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 4.6.1). The `smartbextdep` implementation of `bext/bdep` (see Section 3.3) 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 enourmous, the instruction itself would only be useful in niche applications, and it would require implementors 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 4.6.1).

- A reverse-subract-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-subract-immediate operation is not very bit-manipulation specific, would require a much larger encoding space than the rest of XBitmanip (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 XBitmanip. See Section 4.2 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 XBitmanip.

- Instructions for testing, setting and clearing bits. This operations can be performed in few easily macro-op fusable 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 fusable instruction sequences (see Section 4.7).

- 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 endianess 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. [8] 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. [7]

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. [7])

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
  sll a0, a2
  c.or a0, a3
  ret
```

```
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 `rd`$'$ = `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 XLEN $- i$ to calulate the shift amount for the 2nd shift in a funnel shift (see Section 4.5, XLEN does not actually need to be added because the shift operations only uses the lower log_2(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 neccessary?

This criticism is justified.

1. `c.neg` contains `rd`$'$/`rs2`$'$ instead of `rd`$'$/`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`
1x	15	1x	15	`grev`
2x	15	2x	16	`grevi, gzip`
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 3.1: XBitmanip encoding space ($log2$, i.e. in equivalent number of bits)

adn 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.

3.2 Analysis of used encoding space

Table 3.1 lists all XBitmanip 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 XBitmanip 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 3.2. (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 XBitmanip 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 3.2: Reserved "C" encoding space as of Version 2.2 of the RISC-V User-Level ISA

On RV32, XBitmanip requires the equivalent of a ≈ 18.7 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 XBitmanip extension needs less than one-eighth of the encoding space of the `SLTIU` instruction.

$$log_2(13 \cdot 2^{15}) \approx 18.700$$

On RV64, XBitmanip requires the equivalent of a ≈ 19.8 bit encoding space in the uncompressed encoding space. I.e. the entire RV64 XBitmanip extension needs about one-quarter of the encoding space of the `SLTIU` instruction.

$$log_2(17 \cdot 2^{15} + 5 \cdot 2^{16}) \approx 19.755$$

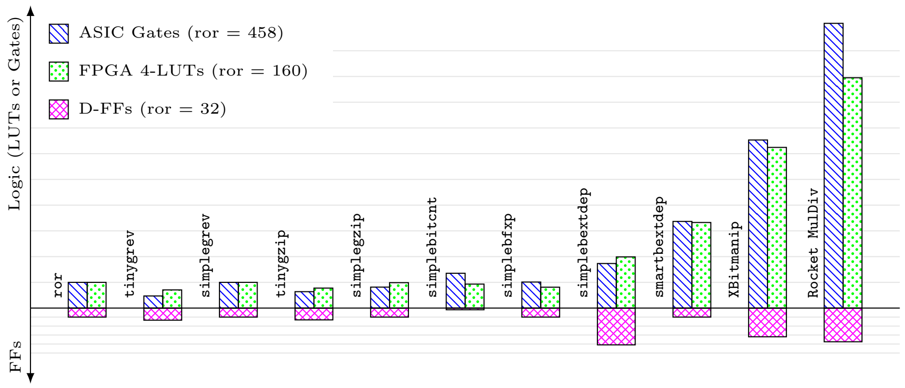

Figure 3.1: Relative area of XBitmanip reference cores compared to simple 32-bit rotate shift and Rocket MulDiv. The height of 1 LUT ≙ 2.86 Gates, 1 FF ≙ 5.00 Gates.

3.3 Area usage of reference implementations

We created RV32 implementations for the different compute cores necessary to implement XBitmanip (and XBitfield): `https://github.com/cliffordwolf/xbitmanip/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 XBitmanip (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 `gzip`. It takes 5 cycles for one operation.
- `simplegrev` — A simple single-cycle implementation of `grev`/`grevi`.
- `simplegzip` — A simple single-cycle implementation of `gzip`.
- `simplebitcnt` — A simple single-cycle implementation of `clz`, `ctz`, and `pcnt`.
- `simplebfxp` — A simple single-cycle implementation of the XBitfield `bfxp` instruction (see Chapter 5). 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 [4].

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.

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
`XBitmanip`	2992	42	999	13	102
`Rocket MulDiv`	5068	49	1432	24	120

Table 3.3: Area and logic depth of XBitmanip reference cores compared to simple 32-bit rotate shift and Rocket MulDiv. The entry `XBitmanip` is just the total of `simplegrev` + `simplegzip` + `simplebitcnt` + `smartbextdep`. Not included in `XBitmanip` 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.

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

Table 3.3 and Figure 3.1 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 4

Evaluation

This chapter contains a collection of short code snippets and algorithms using the XBitmanip 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 motivarion behind this chapter is to show that all the common bit manipulation tasks can be performed in few instructions using XBitmanip. (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.

4.1 Bitfield extract

Extracting a bit field of length `len` at position `pos` can be done using two shift operations (or equivialently using the `bfext` pseudo instruction, see Section 2.10).

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

The XBitfield `bfxp` instruction (see Chapter 5) performs this operation plus one more left shift and an OR operation in a single instruction.

4.2 MIX/MUX pattern

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

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

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

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

4.3 Bit scanning and counting

Counting leading ones:

```
c.not a0
clz a0, a0
```

Counting trailing ones:

```
c.not a0
ctz a0, a0
```

Counting bits cleared:

```
  c.not a0
  pcnt a0, a0
```

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

Odd parity:

```
  pcnt a0, a0
  c.andi a0, 1
```

Even parity:

```
  pcnt a0, a0
  c.addi a0, 1
  c.andi a0, 1
```

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

4.4 Test, set, and clear individual bits

Extracting bit N:

```
  c.srli a0, N
  c.andi a0, 1
```

Branching on bit N set:

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

Branching on bit N clear:

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

Setting bit N (note that this `li` is guaranteed to be only one instruction on RV32):

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

Setting bit N without spilling:

```
  rori a0, a0, N
  ori 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 guaranteed to be only one instruction on RV32):

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

Clearing bit N without spilling:

```
  rori a0, a0, N
  c.andi 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
  c.andi a0, -2
  c.or a0, a1
  rori a0, a0, 32-N
```

4.5 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.

This can be very useful for processing a stream of bit data that is not composed of units aligned to byte boundaries. (Especially when implemented in a single instruction, it can also be useful for misaligned memcopy when source and destination are not misaligned by the same offset.)

Performing a funnel right shift of `a0` (LSB) and `a1` (MSB) by the shift amount specified in `a2` (with $0 <$ `a2` $<$ XLEN):

```
  c.srl a0, a2
  c.neg a2
  c.sll a1, a2
  c.or a0, a1
```

When the shift amount is a compile-time constant, then a 32-bit funnel shift can be performed using two XBitfield `bfxp` instructions (see Chapter 5):

```
  bfxp a0, a0, zero, N, 32-N, 0
  bfxp a0, a1, a0, 0, N, 32-N
```

4.6 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).

4.6.1 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)
  c.and a2, a1
  andc a0, a0, a1
  c.or a0, a2
```

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
  c.or a1, a2

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

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

4.6.2 Using omega-flip networks

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

```
zip a0, a0
grevi a2, a0, 1
c.and a2, a1
andc a0, a0, a1
c.or a0, a2
```

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

```
grevi a2, a0, 1
c.and a2, a1
andc a0, a0, a1
c.or a0, a2
unzip a0, a0
```

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

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

4.6.3 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.

4.6.4 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
```

```
  c.not a1
  bext a0, a0, a1
  pcnt a1, a1
  ror a0, a0, a1
  c.or 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. [2, p. 152f, 162f]

4.7 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 XBitmanip instructions. Table 4.1 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 4.2.

BMI2 adds a few `*X` isnstructions 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 4.3.

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 4.4.

4.8 Comparison with RI5CY Bit Manipulation ISA

The following section compares XBitmanip with the RI5CY bit manipulation instructions as documented in [3]. All RI5CY bit manipulation instructions (or something very close to their behavior)

[1]Depending on if you ask Intel or AMD you will get different opinion regarding whether LZCNT and/or TZCNT are part of ABM or BMI1.

x86 Instruction	Bytes		RISC-V Code
	x86	RV	
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 4.1: Comparison of x86+ABM with XBitmanip

can be emulated with XBitmanip 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)
  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:
```

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 4.2: Comparison of x86 BMI1 with XBitmanip

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 4.3: Comparison of x86 BMI2 with XBitmanip

```
  sll rd, rs, rx
  sra rd, rd, ry

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

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
```

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 4.4: Comparison of x86 TBM with XBitmanip

`p.extractu` can be efficiently emulated with a single XBitfield `bfxp` instruction (see Chapter 5):

```
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 5), 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 5):

```
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 5):

```
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 XBitmanip 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.

4.9 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).

```
 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 bitboards operations are simple straight-forward operations, but mirroring and rotating bitboards can take up to 20 instructions on x86.

4.9.1 Mirroring bitboards

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

```
Flip horizontal:
 63 62 61 60 59 58 57 56    XBitmanip:
 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    XBitmanip:
  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    XBitmanip:
 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
```

4.9.2 Rotating bitboards

Using `gzip` a bitboard can be transposed easily:

```
Transpose:
  7 15 23 31 39 47 55 63    XBitmanip:
  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 [1]. With XBitmanip the rotate operation only takes 4 operations:

```
rotate_bitboard:
  bswap a0, a0
  zip a0, a0
  zip a0, a0
  zip a0, a0
```

4.9.3 Explanation

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

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

The GREV operation with shamt N is `i'[5:0]` $=$ `i[5:0]`$\mathrm{XOR}N$.

And a GZIP operation corresponds to a rotate left shift by one position of any continous region of `i[5:0]`.

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 it by 3 positions. This can easily done by rotate shifting the entire `i[5:0]` by one bit position (`zip`) three times.

4.9.4 Rotating Bitcubes

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

```
rotate_x:                  rotate_y:                  rotate_z:
  hswap a0, a0               brev.n a0, a0              nswap.h
  zip4 a0, a0                zip a0, a0                 zip.h a0, a0
  zip4 a0, a0                zip a0, a0                 zip.h a0, a0
                             zip4 a0, a0
                             zip4 a0, a0
```

4.10 Decoding RISC-V Immediates

The following code snippets decode and sign-extend the immedate 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
  gzip a0, a0, 14
  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 XBitmanip)
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 XBitmanip)
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
```

Chapter 5

XBitfield Extension

The most asked-for feature in XBitmanip is an instruction for bit field extract. XBitmanip recommends the usage of `slli` followed by `slri` for this operation and asks implementors to fuse this sequence into a single macro-op. However, there are valid concerns regarding this approach:

- Not all processors that might want to implement a fast bitfield extract can fuse macro-ops.
- In some architectures it might be hard to fuse the two shift instructions if one of them is using a 32-bit instruction encoding. Therefore the instruction can not be fused if `rd` $\neq$ `rs1`.
- `bext` and `bdep` can do a lot of the heavy lifting in applications that would otherwise require many individual bitflied-extract operations. But smaller cores might not provide a fast `bext`/`bdep` unit, or provide no hardware support for those instructions at all.

Therefore this chapter proposes the RISC-V XBitfield extension. Note that, unlike XBitmanip, **we do not propose XBitfield for adoption as official RISC-V ISA extension**.

The encoding for a proper 32-bit bitfield extract instruction, if encoded in a clean and future-safe way that scales all the way to RV128, would require a 14 bit immediate (7 bits for start and 7 bits for length), or the equivalent of 4 I-type instructions with full immediate length, or half of a major opcode, way too wasteful for an official RISC-V ISA extension, especially for something that can be encoded in the same space with two compressed instructions.

However, there is a tendency for implementors to fill the unused instruction encoding space with a myriad of complex instructions with large immediates. [3]

So might as well propose XBitfield, a custom, non-standard extension that uses almost the entire major opcode 1111011 (*custom-3*) for a single bit-field extract and place (`bfxp`) instruction, effectively packing 3 shift-immediate operations and one OR in one single instruction.

The main goal here is that individual implementors can use the same patches for compiler toolchains. By standardising one powerful but easy to implement instruction instead of a whole range of simpler instructions we keep the complexity of the extension low and ease adoption.

5.1 Bit-field extract and place (`bfxp`, `bfxpc`)

The bit-field extract and place instruction extracts a bitfield of length `len` starting at bit position `start`, creates a new value with the extraced bitfield at offset `dest`, and fills the other bits in the output with `rs2`.

The bit-field extract and place complement instruction does the same thing, but first complements `rs1`.

The immediate arguments `start` and `dest` are 5 bit unsigned immediates on RV32 and a 6 bit unsigned immediates on RV64.

The immediate argument `len` is an unsigned 5 bit immediates. On RV64 `len` = 0 encodes for `len` = 32. On RV32 `len` = 0 is reserved for other instructions.

Instruction words with `start` + `len` > XLEN or `dest` + `len` > XLEN are reserved for other instructions.

```
uint_xlen_t bfxp(uint_xlen_t rs1, uint_xlen_t rs2,
        unsigned start, unsigned len, unsigned dest)
{
    assert(start < XLEN && len < 32 && dest < XLEN);

    if (XLEN > 32 && len == 0)
        len = 32;

    assert(start + len <= XLEN);
    assert(dest + len <= XLEN);
    assert(len != 0);

    uint_xlen_t x = rs1;
    x <<= XLEN-start-len;
    x >>= XLEN-len;
    x <<= dest;

    uint_xlen_t y = ~uint_xlen_t(0);
    y <<= XLEN-start-len;
    y >>= XLEN-len;
    y <<= dest;

    return x | (rs2 & ~y);
}

uint_xlen_t bfxpc(uint_xlen_t rs1, uint_xlen_t rs2,
        unsigned start, unsigned len, unsigned dest)
{
    return bfxp(~rs1, rs2, start, len, dest);
}
```

The assembler mnemonics for `bfxp` and `bfxpc` are as follows.

```
bfxp  rd, rs1, rs2, start, len, dest
bfxpc rd, rs1, rs2, start, len, dest
```

`bfxp`/`bfxpc` occupies the *custom-3* major opcode:

31 18	17 15	14 11	10	9 7	6 0
imm	rs1′	imm	rs2*	rd′	1111011
14	3	4	1	3	7
`start[3]` `start[2]` `start[1]` `start[0]` `len[3]` `len[2]` `len[1]` `len[0]` `dest[5]` `dest[4]` `dest[3]` `dest[2]` `dest[1]` `dest[0]`		`start[5]` `start[4]` `len[4]` `bfxpc`			

With rs1′ and rd′ interpreted as in "C" opcodes (rs1 = rs1′+8, rd = rd′+8), and rs2* = 0 encoding for rs2 = x0 (zero) and rs2* = 1 encoding for rs2 = rd.

The bits from `start`, `len`, and `dest` are placed in the immediate fields in a manner that aims at maximizing the usefulness of the remaining brownfield in the *custom-3* opcode: On RV64, if instruction bits 14:12 are all set (funct3 = 111), then the instruction is reserved if any of the bits 31:24 are set.

5.2 ZBitfield extension using 48-bit encoding

The `bfxp`/`bfxpc` instructions can be cleanly encoded using a 48-bit instruction format. For example:

47 41	40 34	33 27	26 22	21 17	16 12	11 6	5 0
start	len	dest	rs2	rs1	rd	OP-BFXP	011111
start	len	dest	rs2	rs1	rd	OP-BFXPC	011111
7	7	7	5	5	5	6	6

There are no standard instruction formats for 48-bit instructions yet, and no RISC-V processors that support instructions > 32 bits. But when they become a thing, a `bfxp`/`bfxpc` ZBitfield extension using a 48-bit encoding should be considered.

5.3 Evaluation: Decoding RISC-V Immediates

The following code snippets decode and sign-extend the immedate 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:
  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

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 comprimise 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 XBitmanip instructions the C reference functions given in Chapter 2 are already reasonably efficient.

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 scattergather
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
????-??-??	0.36	—

Bibliography

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

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

[3] 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.

[4] 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.

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

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

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

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