RISC-V XBitmanip Extension

Document Version 0.35-draft

Editor: Clifford Wolf
Symbiotic GmbH
clifford@symbioticeda.com

April 24, 2018

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): Allen Baum, Steven Braeger, Michael Clark, Po-wei Huang, Luke Kenneth Casson Leighton, 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.
- 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 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 instruction that already operate on bits, the minimal set of instructions need to 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

- 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.

The sections on encodings are mere placeholders.

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

The `clz` operation counts the number of 0 bits before the first 1 bit (counting from the most significant bit) in the source register. This is related to the "integer logarithm". It takes a single register as input and operates on the entire register. 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 after the last 1 bit. 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 computes the number of 1 bits in a register. It takes a single register as input and operates on the entire register.

This operation counts the total number of set bits in the register.

```
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
???????	src2	src1	ANDCW	dest	OP-32

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, it shifts in ones. This can be used in mask creation or bit-field insertions, for example.

These instructions are exactly the same as the equivalent logical shift operations, except the shift shifts in ones values.

```
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

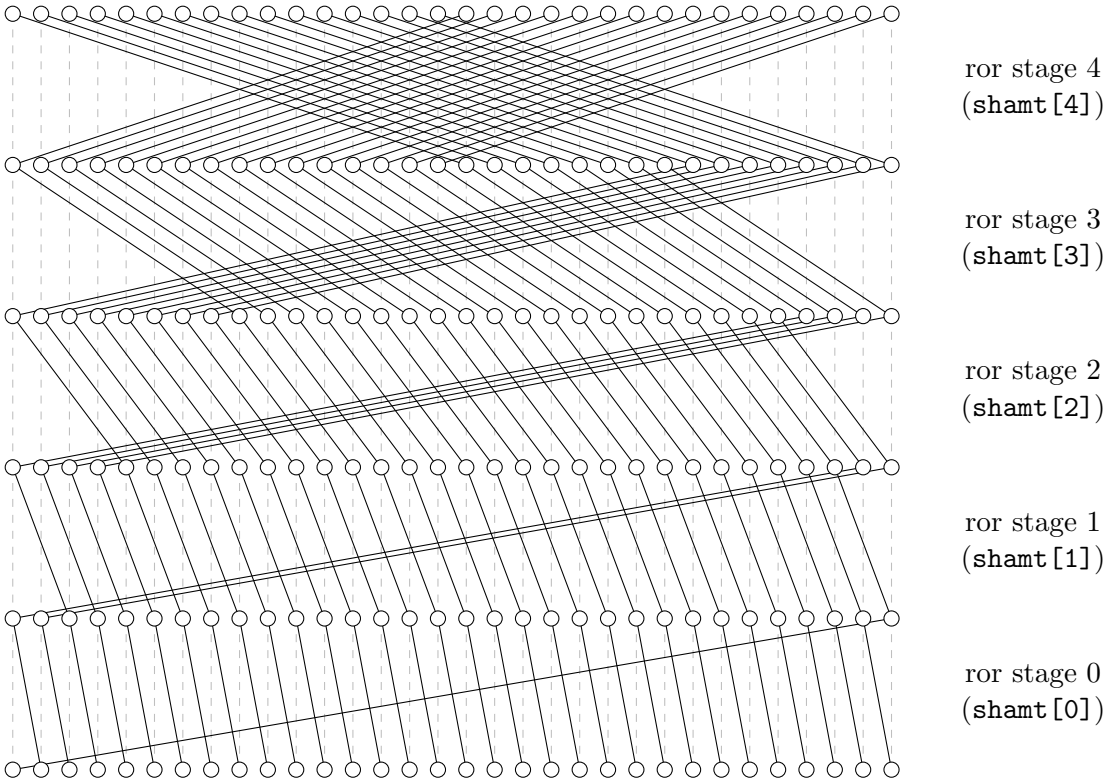


Figure 2.1: `rot` permutation network

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 opcodes can be encoded using the bit pattern that would otherwise encode an "Arithmetic Left Shift" (which is an operation that does not exist).

Table 2.1: Rotate Encodings

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

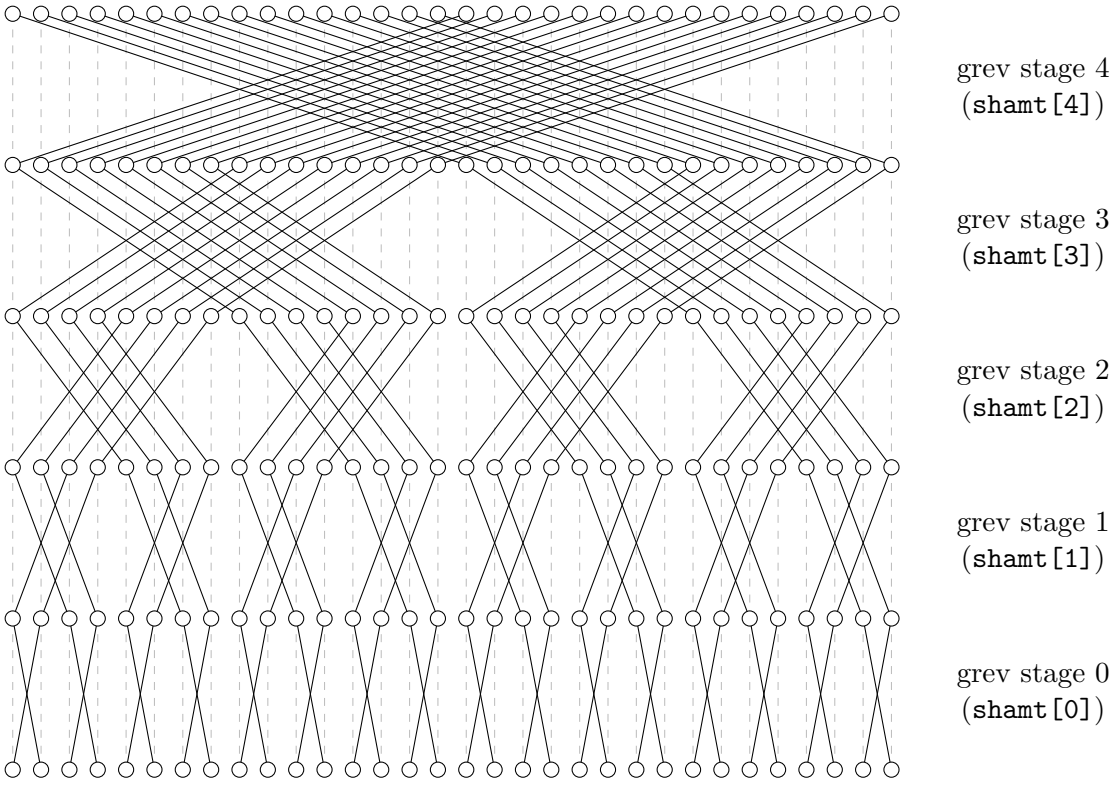


Figure 2.2: `grev` permutation network

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 & 0x5555555555555555ull) <<  1) |
                        ((x & 0xAAAAAAAAAAAAAAAAull) >>  1);
    if (shamt &  2) x = ((x & 0x3333333333333333ull) <<  2) |
                        ((x & 0xCCCCCCCCCCCCCCCCull) >>  2);
    if (shamt &  4) x = ((x & 0x0F0F0F0F0F0F0F0Full) <<  4) |
                        ((x & 0xF0F0F0F0F0F0F0F0ull) >>  4);
    if (shamt &  8) x = ((x & 0x00FF00FF00FF00FFull) <<  8) |
                        ((x & 0xFF00FF00FF00FF00ull) >>  8);
    if (shamt & 16) x = ((x & 0x0000FFFF0000FFFFull) << 16) |
                        ((x & 0xFFFF0000FFFF0000ull) >> 16);
    if (shamt & 32) x = ((x & 0x00000000FFFFFFFFull) << 32) |
                        ((x & 0xFFFFFFFF00000000ull) >> 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 `rot` 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
???????	src2	src1	GREVW	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
?????	mode	src	GREVI	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
???????	mode	src	GREVIW	dest	OP-IMM-32

`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`.

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])$/`. See Table 2.2.

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
~~0000 0~~	no-op	*reserved*
~~0000 1~~	no-op	*reserved*
0001 0	`i[1] -> i[0]`	`zip.n, unzip.n`
~~0001 1~~	*equivalent to 0001 0*	*reserved*
0010 0	`i[2] -> i[1]`	`zip2.b, unzip2.b`
~~0010 1~~	*equivalent to 0010 0*	*reserved*
0011 0	`i[2] -> i[0]`	`zip.b`
0011 1	`i[2] <- i[0]`	`unzip.b`
0100 0	`i[3] -> i[2]`	`zip4.h, unzip4.h`
~~0100 1~~	*equivalent to 0100 0*	*reserved*
~~0101 0~~	`i[3] -> i[2], i[1] -> i[0]`	*reserved*
~~0101 1~~	*equivalent to 0101 0*	*reserved*
0110 0	`i[3] -> i[1]`	`zip2.h`
0110 1	`i[3] <- i[1]`	`unzip2.h`
0111 0	`i[3] -> i[0]`	`zip.h`
0111 1	`i[3] <- i[0]`	`unzip.h`
1000 0	`i[4] -> i[3]`	`zip8, unzip8`
~~1000 1~~	*equivalent to 1000 0*	*reserved*
~~1001 0~~	`i[4] -> i[3], i[1] -> i[0]`	*reserved*
~~1001 1~~	*equivalent to 1001 0*	*reserved*
~~1010 0~~	`i[4] -> i[3], i[2] -> i[1]`	*reserved*
~~1010 1~~	*equivalent to 1010 0*	*reserved*
~~1011 0~~	`i[4] -> i[3], i[2] -> i[0]`	*reserved*
~~1011 1~~	`i[4] <- i[3], i[2] <- i[0]`	*reserved*
1100 0	`i[4] -> i[2]`	`zip4`
1100 1	`i[4] <- i[2]`	`unzip4`
~~1101 0~~	`i[4] -> i[2], i[1] -> i[0]`	*reserved*
~~1101 1~~	`i[4] <- i[2], i[1] <- i[0]`	*reserved*
1110 0	`i[4] -> i[1]`	`zip2`
1110 1	`i[4] <- i[1]`	`unzip2`
1111 0	`i[4] -> i[0]`	`zip`
1111 1	`i[4] <- i[0]`	`unzip`

Table 2.2: RV32 modes 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:

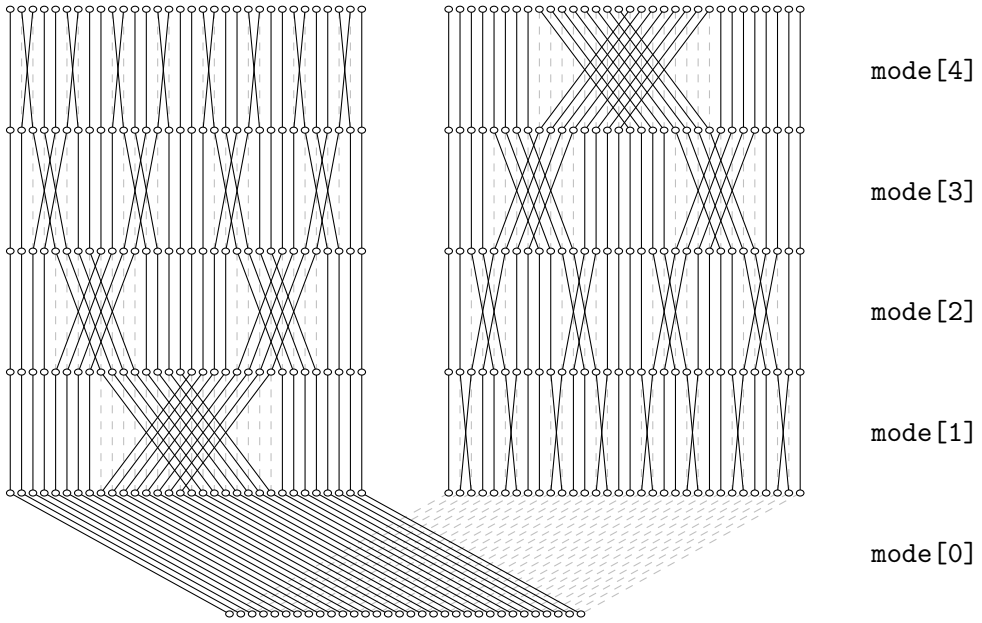


Figure 2.3: `gzip` permutation network without "flip" stages

```
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;
}

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;
}
```

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.

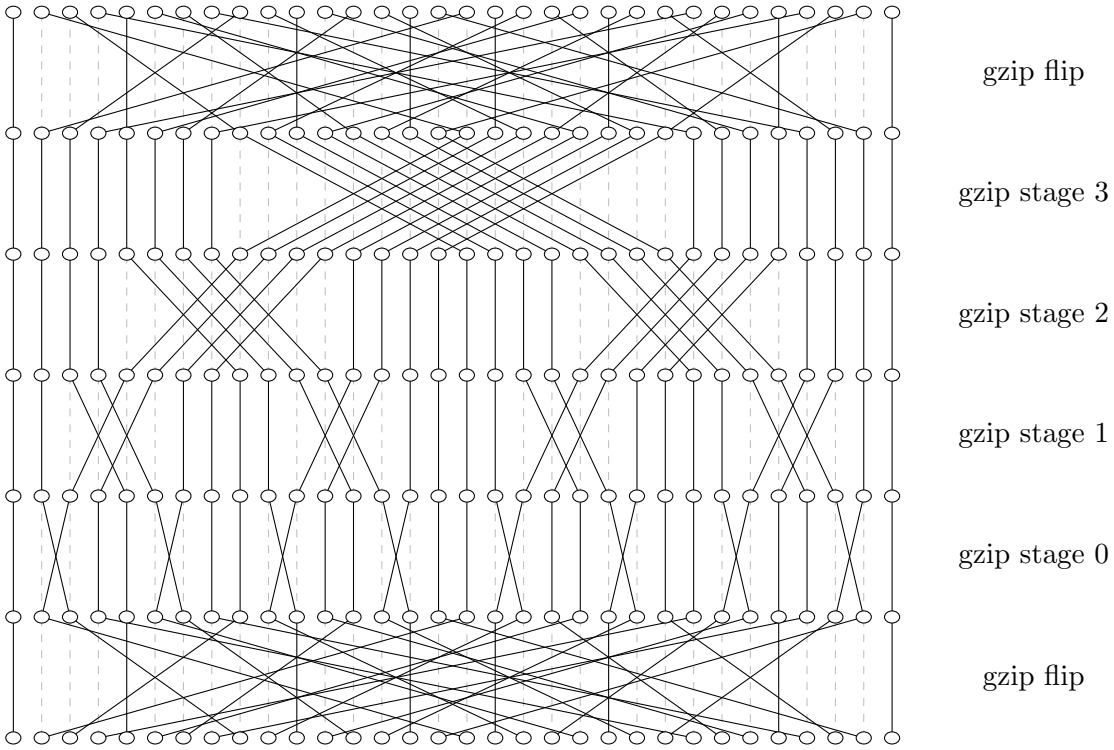
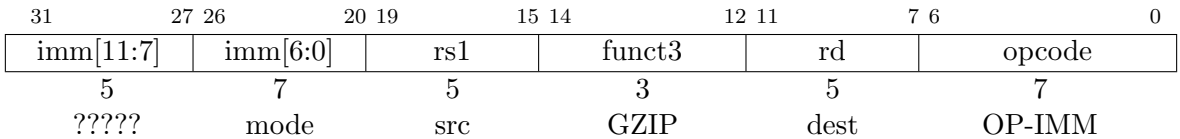


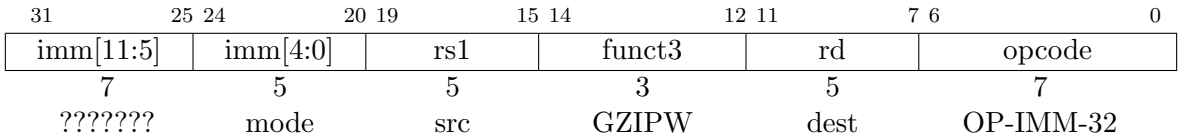
Figure 2.4: `gzip` permutation network with "flip" stages

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
add a0, a1
slli a1, a0, 8
add a0, a1
slli a1, a0, 16
add a0, a1
```

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





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

`BDEP rd,rs1,rs2` 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 c = 0, m = 1, mask = rs2;
    while (mask) {
        uint_xlen_t b = mask & -mask;
        if (rs1 & b)
            c |= m;
        mask -= b;
        m <<= 1;
    }
    return c;
}

uint_xlen_t bdep(uint_xlen_t rs1, uint_xlen_t rs2)
{
    uint_xlen_t c = 0, m = 1, mask = rs2;
    while (mask) {
        uint_xlen_t b = mask & -mask;
        if (rs1 & m)
            c |= b;
        mask -= b;
        m <<= 1;
    }
    return c;
}
```

Implementations might choose to use smaller multi-cycle implementations of `bext` and `bdep`. 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.

15 14 13	12	11 10	9 8 7	6 5 4 3 2	1 0	
011	nzimm[9]	2		nzimm[4|6|8:7|5]	01	C.ADDI16SP *(~~RES, nzimm=0~~)*
011	nzimm[17]	rd≠{0, 2}		nzimm[16:12]	01	C.LUI *(~~RES, nzimm=0~~; HINT, rd=0)*
011	0	00	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*

This 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.

2.10 Pseudo instructions and macros

RISC-V is a RISC instruction set. Unless there is a good argument against it, we try not to assign dedicated opcodes for complex operations that are just easily macro-op fusable short sequences of already existing instructions, especially if compressed instructions can be used to keep the length of those sequences reasonably short.

The assembler should provide pseudo-instructions for some of the sequences that are implemented using dedicated instructions in some CISC bit-maniuplation instruction sets. The sections describing sequences where the assembler should provide pseudo-instructions are titled "pseudo instruction", whereas sections that just informally describe useful sequences are titles "macro".

Many of the code snippets below can utilize compressed instructions. But for simplicity we use uncompressed instruction mnemonics in the assembler listings. Some of the macros spill to temporary registers. `t0, t1, t2, ...` is used for spilling in the assembler listings. The input register are referred to by `rs1, rs2, ...` and the output register is `rd`.

2.10.1 MIX/MUX Macros

MIX Operation

A MIX operation selects bits from `rs1` and `rs2` based on the bits in the control word `rs3`.

```
and t0, rs1, rs3
andc t1, rs2, rs3
or rd, t0, t1
```

MUX Operation

A MUX operation selects word `rs1` or `rs2` based on if the control word `rs3` is zero or nonzero, without branching.

```
snez t0, rs3
neg t0, t0
and t1, rs1, t0
andc t2, rs2, t0
or rd, t1, t2
```

Or when `rs3` is already either 0 or 1:

```
neg t0, rs3
and t1, rs1, t0
andc t2, rs2, t0
or rd, t1, t2
```

2.10.2 Bit-field extract and deposit

Pseudo instruction `bfext`

Extract the continous bit field starting at `pos` with length `len` from `rs`:

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

Macros for bit-field deposit

Deposit `len` bits from `rs2` at `pos` in `rd`, remaining bits in `rd` are filled from `rs1`.

Assuming `rs1[pos+len-1:pos]`=0 and `rs2[XLEN-1:pos]`=0:

```
slli t0, rs2, pos
or rd, rs1, t0
```

Otherwise masking and/or shift operations should be used to clear the extra bits in `rs1` and `rs2` first. On a machine with fast `bdep`, the `bdep` instruction can be used to shift and mask `rs2` in one instruction using at the same mask that is also used to mask `rs1`:

```
li t0, ((1 << len)-1) << pos
andc t1, rs1, t0
bdep t0, rs2, t0
or rd, t0, t1
```

2.10.3 Pseudo instructions for bit scanning and counting

Pseudo instructions for counting leading/trailing ones

```
clo rd, rs   ->   not rd, rs
                  clz rd, rd

cto rd, rs   ->   not rd, rs
                  ctz rd, rd
```

Pseudo instruction for counting bits cleared

```
pcntn rd, rs   ->   not rd, rs
                    pcnt rd, rd
```

Pseudo instructions for parity

Odd parity:

```
oparity rd, rs   ->   pcnt rd, rs
                      andi rd, rd, 1
```

Even parity:

```
eparity rd, rs   ->   pcnt rd, rs
                      addi rd, rd, 1
                      andi rd, rd, 1
```

2.10.4 Pseudo instructions using `grevi`

On RV32:

```
brev    rd, rs   ->    grevi rd, rs, 31    ; bitwise reverse
brev.h  rd, rs   ->    grevi rd, rs, 15    ; reverse bits in each 16 bit half-word
brev.b  rd, rs   ->    grevi rd, rs,  7    ; reverse bits in each  8 bit byte

bswap   rd, rs   ->    grevi rd, rs, 24    ; reverse the byte order
bswap.h rd, rs   ->    grevi rd, rs,  8    ; swap bytes in each 16 bit half-word

hswap   rd, rs   ->    grevi rd, rs, 16    ; swap the two 16 bit half-words
```

On RV64:

```
brev    rd, rs   ->    grevi rd, rs, 63    ; bitwise reverse
brev.w  rd, rs   ->    grevi rd, rs, 31    ; reverse bits in each 32 bit word
brev.h  rd, rs   ->    grevi rd, rs, 15    ; reverse bits in each 16 bit half-word
brev.b  rd, rs   ->    grevi rd, rs,  7    ; reverse bits in each  8 bit byte

bswap   rd, rs   ->    grevi rd, rs, 56    ; reverse the byte order
bswap.w rd, rs   ->    grevi rd, rs, 24    ; reverse byte order in each 32 bit word
bswap.h rd, rs   ->    grevi rd, rs,  8    ; swap bytes in each 16 bit half-word
```

```
  hswap   rd, rs   ->    grevi rd, rs, 48    ; reverse order of 16 bit half-words
  hswap.w rd, rs   ->    grevi rd, rs, 16    ; swap 16 bit half-words in each 32 bit word

  wswap   rd, rs   ->    grevi rd, rs, 32    ; swap the two 32 bit words
```

2.10.5 Macros for bit permutations

Butterfly operations

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

```
  grevi t0, a0, (1 << N)
  and t0, t0, a1
  andc a0, a0, a1
  or a0, a0, t0
```

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 t0, a1, 1
  or a1, a1, t0

bfly_msk_1:
  zip2 a1, a1
  slli t0, a1, 2
  or a1, a1, t0

bfly_msk_2:
  zip4 a1, a1
  slli t0, a1, 4
  or a1, a1, t0

...
```

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

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

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

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

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

Omega-Flip Networks

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

```
zip a0, a0
grevi t0, a0, 1
and t0, t0, a1
andc a0, a0, a1
or a0, a0, t0
```

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

```
grevi t0, a0, 1
and t0, t0, a1
andc a0, a0, a1
or a0, a0, t0
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.

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.

Sheep-and-goats operation

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 (sheeps) to the MSB end of the word, without changing the order of sheeps or goats.

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

```
bext t0, a0, a1
not a1, a1
bext a0, a0, a1
pcnt a1, a1
ror a0, a0, a1
or a0, a0, t0
```

Any arbitrary bit permutation can be implement 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.

Chapter 3

Discussion

3.1 Frequently Asked Questions

`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[1] have show that `andc` is much more common in bit manipulation code than any other operators. 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

[1] http://svn.clifford.at/handicraft/2017/bitcode/

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 rotate shift.)

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.

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.[2]

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.[3])

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.

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.

SHUFFLE looks like a really expensive operation. Do we really need it?

Even though this instruction looks expensive, it is actually quite simple to implement. The butterfly operation can just reuse the butterfly circuit that is already present to support the `grev` instruction, and zip and unzip are very cheap to implement (just one additional word-wide mux each). The "return 0" part for nonzero commands and reserved modes is also very cheap.

Considering that SHUFFLE is very cheap to implement on top an existing GREV implementation, and considering that it only requires a single R-type instruction, and that software emulation

[2] https://github.com/cliffordwolf/bextdep
[3] https://github.com/cliffordwolf/bextdep

of similar functionality requires tens of instructions (and/or multiplications with large "magic constants"), it is a relatively good option.

With dedicated unary ZIP/UNZIP instructions it would be possible to emulate a single SHUFFLE instruction in under 10 instructions. For example, emulating a single OMEGA instruction (input in a0 and mask in a1):

```
grevi t0, a0, 0
zip a1, a1
and t0, t0, a1
andc a0, a0, a1
or a0, a0, t0
unzip a0, a0
```

Or emulating a single BFLY(2) instruction:

```
grevi t0, a0, 0
zip a1, a1
zip a1, a1
zip a1, a1
and t0, t0, a1
andc a0, a0, a1
or a0, a0, t0
```

This is not too bad, but considering that a single 32-bit permutation takes up to 9 of those, it is probably not a viable option for many bit permutations found in real-world applications.

3.2 Analysis of used encoding space

So how much encoding space is used by the XBitmanip extension?

Table 3.1: XBitmanip encoding space ($log2$, i.e. in equivalent number of bits)

RV32		RV64		Instruction
3x	10	6x	10	CLZ, CLZW, CTZ, CTZW, PCNT, PCNTW
1x	15	3x	15	GREV, GREVW, GREVIW
1x	15	1x	16	GREVI
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

RV32		RV64		Instruction
4x	15	4x	15	ANDC, BEXT, BDEP, SHUFFLE
		4x	15	ANDCW, BEXTW, BDEPW, SHUFFLEW
3x	4	3x	4	C.NEG, C.NOT, C.BREV

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 5.6 bit encoding space, or ≈ 0.1% of the total ≈ 15.6 bits available.

$$log_2(3 \cdot 2^4) \approx 5.585$$
$$100/(2^{15.585-5.585}) \approx 0.098$$

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.

$$log_2(3 \cdot 2^{10} + 13 \cdot 2^{15}) \approx 18.711$$

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

$$log_2(6 \cdot 2^{10} + 22 \cdot 2^{15} + 4 \cdot 2^{16}) \approx 19.911$$

Chapter 4

Evaluation, Algorithms

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 t0, t1, ... are used for spilling.

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.

4.1 Emulating 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 possible if we allow spilling to temporary registers.

Table 4.1: Emulating other Bit Manipulation ISAs using macro-op fusion

x86 Ext	x86 Instruction	Bytes x86	Bytes RV	RISC-V Code
ABM	`popcnt`	5	4	`pcnt a0, a0`
	`lzcnt`	5	4	`clz a0, a0`
BMI1	`andn`	5	4	`andc a0, a2, a1`
	`bextr` (regs)[1]	5	12	`c.add a0, a1` `slo a0, zero, a0`

[1] The BMI1 `bextr` instruction 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.

x86 Ext	x86 Instruction	Bytes x86	Bytes RV	RISC-V Code
				`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`
BMI2	`bzhi`	5	6	`slo a0, zero, a2` `c.and a0, a1`
	`mulx`[2]	5	4	`mul`
	`pdep`	5	4	`bdep`
	`pext`	5	4	`bext`
	`rorx`[2]	6	4	`rori`
	`sarx`[2]	5	4	`sra`
	`shrx`[2]	5	4	`srl`
	`shlx`[2]	5	4	`sll`
TBM	`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`

[2] The `*x` BMI2 isnstructions 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.

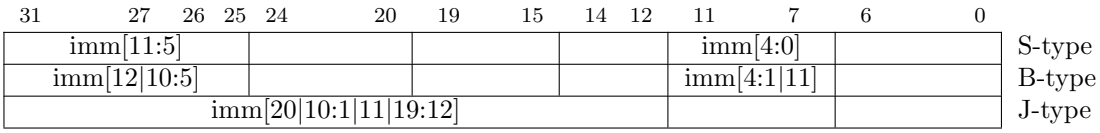
There will be a separate RISC-V standard for recommended sequences for macro-op fusion. The macros listed here are merely for demonstrating that suitable sequences exist. We do not advocate for any of those sequences to become "standard sequences" for macro-op fusion.

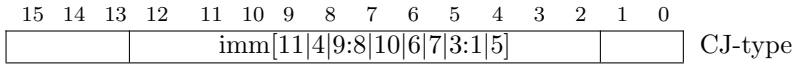
4.2 Emulating RI5CY Bit Manipulation ISA

TBD

4.3 Decoding RISC-V Immediates

The following code snippets decode 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.






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

decode_b:
  rori a0, a0, 8
  lui t0, 0x804eb
  shuffle a0, a0, t0
  li t0, 0x80fe0e01
  bext a0, a0, t0
  c.slli a0, 20
  c.srai a0, 19
  ret

// variant 1 (with shuffle/bext)
decode_j:
  lui t0, 0x0fffb
  shuffle a0, a0, t0
```

```
  lui t0, 0x0f40a
  shuffle a0, a0, t0
  lui t0, 0x70fec
  shuffle a0, a0, t0
  li t0, 0x8ff170fe
  bext a0, a0, t0
  c.slli a0, 12
  c.srai a0, 11
  ret

// variant 2 (with bext but without shuffle)
decode_j:
  li t0, 0x800ff000
  li a1, 0x00100000
  bext a2, a0, t0
  c.and a1, a0
  c.slli a0, a0, 1
  c.srli a0, a0, 22
  c.slli a2, 23
  c.slli a1, 2
  c.slli a0, 12
  c.or a0, a2
  c.or a0, a1
  c.srai a0, 11
  ret

// variant 1 (with shuffle/bext)
decode_cj:
  grevi a0, a0, 1
  lui t0, 0xebcac
  shuffle a0, a0, t0
  lui t0, 0xe3469
  shuffle a0, a0, t0
  li t0, 0x8bc20464
  bext a0, a0, t0
  c.slli a0, 21
  c.srai a0, 20
  ret

// variant 2 (without shuffle/bext)
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

Change History

Table 5.1: Summary of Changes

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)
????-??-??	0.35	Replace `shuffle` with generalized zip (`gzip`)