RISC-V XBitmanip Extension

Document Version 0.34-draft

Editor: Clifford Wolf
Symbiotic GmbH
clifford@symbioticeda.com

April 18, 2018

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): Steven Braeger, Rex McCrary, Po-wei Huang, 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 burocratic 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 straight forward, 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 Zeros (`clz`)

This 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 ˜0, 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;
}
```

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

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

2.2 Count Bits Set (pcnt)

The purpose of this instruction is to compute 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	PCNTW	dest	OP-IMM-32
????????????	src	PCNT	dest	OP-IMM

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

2.3 Generalized Reverse (grev, grevi)

The purpose of this instruction is to provide a single hardware instruction that can implement all of byte-order swap, bitwise reversal, short-order-swap, word-order-swap (RV64I), nibble-order swap, bitwise reversal in a byte, etc, all from a single hardware instruction. It takes in a single register value and an immediate that controls which function occurs, through controlling the levels in the recursive tree at which reversals occur.

This operation iteratively checks each bit immed_i from i=0 to XLEN-1, in XLEN stages, and if the corresponding bit of the 'function_select' immediate is true for the current stage, swaps each adjacent pair of 2^i bits in the register.

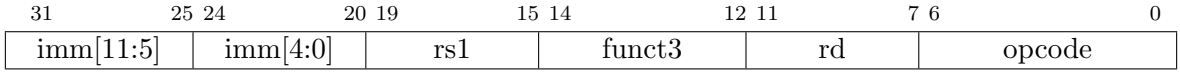
`grevi` 'butterfly' implementation in C on various architectures

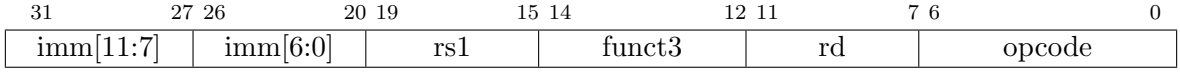
```
uint32_t grevi32(uint32_t rs1, int12_t immed)
{
    uint32_t x = rs1;
    uint5_t k = (uint12_t)immed;
    if (k &  1) x = ((x & 0x55555555) << 1) | ((x & 0xAAAAAAAA) >> 1);
    if (k &  2) x = ((x & 0x33333333) << 2) | ((x & 0xCCCCCCCC) >> 2);
```

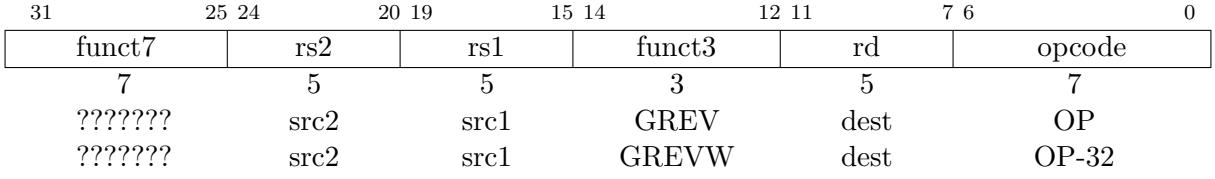
```
    if (k &  4) x = ((x & 0x0F0F0F0F) << 4) | ((x & 0xF0F0F0F0) >> 4);
    if (k &  8) x = ((x & 0x00FF00FF) << 8) | ((x & 0xFF00FF00) >> 8);
    if (k & 16) x = ((x & 0x0000FFFF) <<16) | ((x & 0xFFFF0000) >> 16);
    return x;
}

uint64_t grevi64(uint32_t rs1, int12_t immed)
{
    uint64_t x = rs1;
    uint6_t k = (uint12_t)immed;
    if (k &  1) x = ((x & 0x5555555555555555) << 1) | ((x & 0xAAAAAAAAAAAAAAAA) >> 1);
    if (k &  2) x = ((x & 0x3333333333333333) << 2) | ((x & 0xCCCCCCCCCCCCCCCC) >> 2);
    if (k &  4) x = ((x & 0x0F0F0F0F0F0F0F0F) << 4) | ((x & 0xF0F0F0F0F0F0F0F0) >> 4);
    if (k &  8) x = ((x & 0x00FF00FF00FF00FF) << 8) | ((x & 0xFF00FF00FF00FF00) >> 8);
    if (k & 16) x = ((x & 0x0000FFFF0000FFFF) <<16) | ((x & 0xFFFF0000FFFF0000) >> 16);
    if (k & 32) x = ((x & 0x00000000FFFFFFFF) <<32) | ((x & 0xFFFFFFFF00000000) >> 32);
    return x;
}
```

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







`grev` is encoded as standard R-type opcode and `grevi` is encoded as standard I-type opcode.

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 sloi(uint_xlen_t rs1, int12_t immed)
{
    int shamt = immed & (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);
}

uint_xlen_t sroi(uint_xlen_t rs1, int12_t immed)
{
    int shamt = immed & (XLEN-1);
    return ~(~rs1 >> shamt);
}
```

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

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
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

`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.

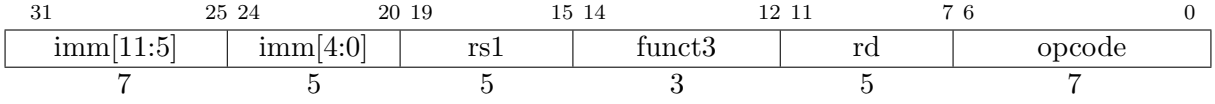
2.5 Rotate (Left/Right) (`rol, ror, rori`)

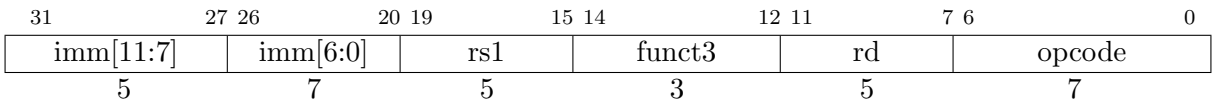
These instructions are similar to shift-logical operations from the base spec, except they shift in the values from the opposite side of the register, in order. This is also called 'circular shift'.

```
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));
}

int_xlen_t rori(uint_xlen_t rs1, int12_t immed)
{
    int shamt = immed & (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	RORW	dest	OP-32
11?????	src2	src1	ROLW	dest	OP-32
11?????	src2	src1	ROR	dest	OP
11?????	src2	src1	ROL	dest	OP

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 And-with-complement (`andc`)

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

This is an R-type instruction that implements the and-with-complement operation `x=(a & ~b)` with an assembly format of `andc ro,r1,r2`.

Additionally, the four positive bitwise operations can all be implemented with the same ALU area (e.g. as a single microcode instruction) using a scheme described in the implementation commentary, and fused versions can be implemented as well.

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

This instruction should be encoded similarly to the instruction in the base spec. The exact instruc-

tion encoding is to be decided, however.

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

This is an R-type instruction that implements the generic bit extract and bit deposit functions. Similar implementation can be referred to as gather/scatter or pack/unpack.

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

`BDEP[W] rd,rs1,rs2` writes LSB justified bits from rs1 to rd using deposit mask in rs2.

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

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

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

This instruction should be encoded similarly to the R-type instructions in the base spec. The exact instruction encoding is to be decided, however.

2.8 Compressed instructions (`c.not, c.neg, c.clz, 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 opecodes in other ISAs must be constructed from smaller atoms in RISC-V XBitmanip code. But implementations might chose 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, `clz` and `brev` (an alias for `grevi rd, rs, -1`, i.e. bitwise reversal) are very common atoms for building common bit manipulation operations. So it is helpful to have compressed opcodes for them as well.

For example, the equivalent to the x86 instruction `BSR` (find index of highest set bit) can be implemented using the following 64 bit sequence that does not spill other registers (and thus is a prime candidate for macro-op-fusion):

```
bsr rsd:
  c.clz rsd
  c.neg rsd
  addi rsd, rsd, 32
```

Likewise, the AMD TBM extension adds 10 x86 bit manipulation instructions. The equivalent RISC-V code sequences can utilize `c.not` in half of the cases.

The compressed instructions `c.not, c.neg, c.clz, 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.CLZ
011	0	11	rs1′/rd′	0	01	C.BREV

This four instructions fit nicely in the reserved space in C.LUI/C.ADDI16SP. They only occupy 0.13% of the ≈ 15.6 bits wide RVC encoding space.

Chapter 3

Alternatives to `bext` and `bdep`

There are issues with `bext` and `bdep`. One is the high hardware cost of the unit performing the operations. Another is the open question of patents potentially preventing implementations to use the best-known single-cycle implementations.

Various multi-cycle implementations have been proposed, but preliminary evaluations of real-world bit manipulation tasks show that multi-cycle `bext` and `bdep` often are not fast enough to outperform algortihms that use sequences of shifts and bit masks.

However, some operations that would benefit from `bext` and `bdep` would be very hard to implement without them. Some examples are Bit scatter/gather (compress/uncompress) operations, goats-and-sheeps, and calulating the index of the Nth set bit in a word. 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
brev a0, a0
clz a0, a0
```

The instruction encoding space used by `bext` and `bdep` is minimal, i.e. not much is gained by preserving the opcode space occupied by the two instructions. So one possible solution would be to make `bext` and `bdep` optional features that may be emulated by software, and add fast implementations of the much simpler `shuffle` and `unshuffle` instructions described below. Those instructions simplify some of the most common cases that would otherwise benefit from a fast `bext` and `bdep` unit.

For small cores this would be an immense simplification because they would not need to implement `bext` and `bdep` in hardware. For large cores that provide `bext` and `bdep` hardware, adding the instructions below is only a minor additional effort. But this additional effort would not be in vain because the instructions below are somewhat orthoganal to `bext` and `bdep`, so that overall the combined feature set of `bext` and `bdep` and the instructions below would be even more powerful than `bext` and `bdep` alone.

A compiler would decide to use `bext` or `bdep` based on the optimization profile for the concrete

processor it is optimizing for, 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.

3.1 Generalized Bit Permutations (`shuffle, unshuffle`)

This instructions perform a bit permutation on the value in rs1. Which bit permutation is performed is defined by the control word in rs2 (Table 3.1).

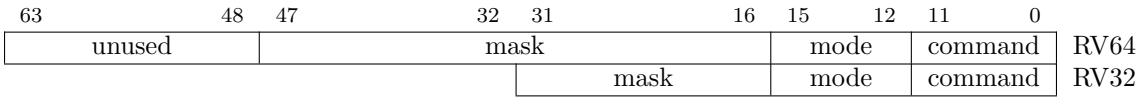


Table 3.1: `shuffle, unshuffle` control word

This spec only defines command = 0. Non-zero command values are reserved for future use. An implementation that does not support a given command must return 0 in rd. Support for command 0 is mandatory. Command values 24-31 are reserved for non-standard extensions (NSE).

Command 0 implements functions that are required for computing zip and unzip operations, butterfly stages or entire butterfly networks, omega stages or networks, flip stages or network, and similar operations. Tables 3.2 and 3.3 list the operations performed by command 0. Note that this functions can all use the existing butterfly network that implements the GREV instructions.

Mode	Description
0000	zip + butterfly(mask, 0)
0001	zip + butterfly(mask, 1)
0010	zip + butterfly(mask, 2)
0011	zip + butterfly(mask, 3)
0100	zip + butterfly(mask, 4)
0101	zip + butterfly(mask, 5) (RV64/128; NSE)
0110	zip + butterfly(mask, 6) (RV128; NSE)
0111	reserved for future standard extensions
1000	butterfly(mask, 0)
1001	butterfly(mask, 1)
1010	butterfly(mask, 2)
1011	butterfly(mask, 3)
1100	butterfly(mask, 4)
1101	butterfly(mask, 5) (RV64/128; NSE)
1110	butterfly(mask, 6) (RV128; NSE)
1111	reserved for future standard extensions

Table 3.2: Modes for shuffle command 0

Modes that are reserved for future standard or non-standard extensions must return 0 in rd on implementations that do not support those future extensions.

`shuffle` with rs2=0 (x0) implements just a zip operation (butterfly is disabled because mask=0), and `unshuffle` with rs2=0 implements just an unzip operation.

Mode	Description
0000	butterfly(mask, 0) + unzip
0001	butterfly(mask, 1) + unzip
0010	butterfly(mask, 2) + unzip
0011	butterfly(mask, 3) + unzip
0100	butterfly(mask, 4) + unzip
0101	butterfly(mask, 5) + unzip (RV64/128; NSE)
0110	butterfly(mask, 6) + unzip (RV128; NSE)
0111	reserved for future standard extensions
1000	reserved for future standard extensions
1001	reserved for future standard extensions
1010	reserved for future standard extensions
1011	reserved for future standard extensions
1100	reserved for future standard extensions
1101	reserved for future standard extensions
1110	reserved for future standard extensions
1111	reserved for future standard extensions

Table 3.3: Modes for unshuffle command 0

The `zip` (aka "shuffle") operation interleaves the bits of the lower and upper half of its argument. The `unzip` (aka "unshuffle") instruction performs the inverse.

In other words, `zip` performs a rotate left shift on the bit indices, and `unzip` performs a rotate right shift on the bit indices. Performing `zip` log_2(XLEN) times is the identity. Performing it log_2(XLEN) − 1 times is equivalent to one execution of `unzip`.

```
uint_xlen_t zip(uint_xlen_t rs1)
{
    uint_xlen_t x = 0;
    for (int i = 0; i < XLEN/2; i++) {
        x |= ((rs1 >> i) & 1) << (2*i);
        x |= ((rs1 >> (i+XLEN/2)) & 1) << (2*i+1);
    }
    return x;
}

uint_xlen_t unzip(uint_xlen_t rs1)
{
    uint_xlen_t x = 0;
    for (int i = 0; i < XLEN/2; i++) {
        x |= ((rs1 >> (2*i)) & 1) << i;
        x |= ((rs1 >> (2*i+1)) & 1) << (i+XLEN/2);
    }
    return x;
}
```

The butterfly operation performs a single butterfly stage N (i.e. the `grev` operation with argument

2^N), which performs XLEN/2 pairwise bit swaps. But unlike `grev` the individual bit swaps are conditional and the XLEN/2 bits in mask determine which bit swaps are taken.

```
uint_xlen_t swapbits(uint_xlen_t x, int p, int q)
{
    assert(p < q);
    x = x ^ ((x & (1 << p)) << (q-p));
    x = x ^ ((x & (1 << q)) >> (q-p));
    x = x ^ ((x & (1 << p)) << (q-p));
    return x;
}

uint_xlen_t butterfly(uint_xlen_t x, uint_xlen_t mask, int N)
{
    int a = 1 << N, b = 2*a;
    for (int i = 0; i < XLEN/2; i++) {
        int p = b*(i/a) + i%a, q = p + a;
        if ((mask >> i) & 1)
            x = swapbits(x, p, q);
    }
    return x;
}
```

Putting it all together:

```
uint_xlen_t shuffle(uint_xlen_t x, uint_xlen_t ctrl)
{
    uint_xlen_t mask = ctrl >> 16;
    bool dozip = !((ctrl >> 15) & 1);
    int stage = (ctrl >> 12) & 7;
    int cmd = ctrl & 0xfff;

    if (cmd != 0 || stage >= LOG2_XLEN)
        return 0;

    x = dozip ? zip(x) : x;
    x = butterfly(x, mask, stage);
    return x;
}

uint_xlen_t unshuffle(uint_xlen_t x, uint_xlen_t ctrl)
{
    uint_xlen_t mask = ctrl >> 16;
    bool dounzip = !((ctrl >> 15) & 1);
    int stage = (ctrl >> 12) & 7;
    int cmd = ctrl & 0xfff;
```

```
    if (cmd != 0 || stage >= LOG2_XLEN || !dounzip)
        return 0;

    x = butterfly(x, mask, stage);
    x = unzip(x);
    return x;
}
```

On RV32, a control word for command 0 can be loaded using a single `lui` instruction. At most $2 \cdot log_2(\text{XLEN}) - 1$ shuffle or unshuffle operations are required to perform an arbitrary bit permutation. Most bit permutations that arise from real-world applications can be implemented in shorter sequences.

Commands in the range 1-2047 with the upper bits (mask/mode) set to zero can be loaded with a single `li` instruction. Note that there is no requirement for future non-zero commands to perform bit-permutations or even reversible operations. But if they do, and if they are not their own inverse, then `shuffle` and `unshuffle` on the same control word should implement inverse operatations. (This is the case for command 0. Note that a butterfly operation without zip or unzip is its own inverse.)

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	SHUFFLE	dest	OP
???????	src2	src1	UNSHUFFLE	dest	OP
???????	src2	src1	SHUFFLEW	dest	OP-32
???????	src2	src1	UNSHUFFLEW	dest	OP-32

This instruction should be encoded similarly to the R-type instructions in the base spec. The exact instruction encoding is to be decided, however.

Pseudo instruction `bfly`

`shuffle` with mode[3]=1 (and mode[2:0] $< log_2(\text{XLEN})$) performs a butterfly operation. The assembler should provide a `bfly` pseudo-instruction for rd $\neq$ rs and constant `mask` and `N` (if `N` is omitted then `N`=0 is assumed):

```
bfly rd, rs, mask[, N]     ->     lui rd, ((mask << 4) | 8 | N)
                                  shuffle rd, rs, rd
```

(On RV64, longer sequences are required instead of `lui` to load the full 32 bit mask into `rd`.)

For example, an arbitrary RV32 bit permutation using a complete butterfly network:

```
bfly a1, a0, <maskA>, 4
bfly a2, a1, <maskB>, 3
bfly a0, a2, <maskC>, 2
bfly a1, a0, <maskD>, 1
bfly a2, a1, <maskE>, 0
bfly a0, a2, <maskF>, 1
bfly a1, a0, <maskG>, 2
bfly a2, a1, <maskH>, 3
bfly a0, a2, <maskI>, 4
```

Permutations arising from real-world applications can often be implemented using shorter sequences.

Pseudo instructions `omega` and `flip`

A zip operation followed by a butterfly(0) is commonly known as an *omega stage*.

A butterfly(0) operation followed by unzip (equivalent to an unzip operation followed by butterfly($\log_2$(XLEN) − 1)) is commonly known as a *flip stage*. The assember should provide appropiate pseudo-instructions for rd ≠ rs and constant `mask` and `N` (if `N` is omitted then `N`=0 is assumed):

```
omega rd, rs, mask[, N]     ->     lui rd, ((mask << 4) | N)
                                   shuffle rd, rs, rd

flip  rd, rs, mask[, N]     ->     lui rd, ((mask << 4) | N)
                                   unshuffle rd, rs, rd
```

For example, an arbitrary RV32 bit permutation using a complete omega-flip network:

```
omega a1, a0, <maskA>
omega a0, a1, <maskB>
omega a1, a0, <maskC>
omega a0, a1, <maskD>
omega a1, a0, <maskE>
flip  a0, a1, <maskF>
flip  a1, a0, <maskG>
flip  a0, a1, <maskH>
flip  a1, a0, <maskI>
flip  a0, a1, <maskJ>
```

(Either `<maskE>` or `<maskF>` can be zero, effectively turning either the last `omega` into a `zip` or the first `flip` into an `unzip`.)

As for butterfly networks, permutations arising from real-world applications can often be implemented using a much shorter sequence. Especially if `bfly`, `omega` and `flip` can be mixed arbitrarily.

Pseudo instructions `zip` and `unzip`

With a zero value as control word, `shuffle` and `unshuffle` perform simple zip or unzip operations. The assembler should provide pseudo-instructions for those cases (Table 3.4).

Pseudoinstruction	Base Instruction(s)
`zip rd, rs`	`shuffle rd, rs, zero`
`unzip rd, rs`	`unshuffle rd, rs, zero`

Table 3.4: Pseudo instructions `zip` and `unzip`

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

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

```
andi a0, a0, 0xff
zip a0, a0
zip a0, a0
slli a1, a0, 4
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.

Implementation notes

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.

Since future non-zero commands are optional they do not add to the complexity requirements of the command. A simple implementation can simply ignore them and return 0 in rd.

Chapter 4

Discussion and Evaluation

4.1 Frequently Asked Questions

Why `clz` instead of `ctz`? Why not XYZ?

The important thing is that we wanted to have a small ISA that only provides useful atoms for building most of the other "usual" bit manipulation instructions.

If we chose `clz` or `ctz` is not that important. But we want to pick only one and let the used build the other by combining the supported instruction with bitwise reverse (which is provided by `grevi`).

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

The `grevi` instruction can be used to build a wide range of common bit permutation instructions. For example on RV32:

Table 4.1: Some GREVI Pseudo-Instructions

Mode	Meaning on RV32
4	swap the nibbles of each byte
7	reverse the bits in each byte
15	reverse the bits in each halfword
16	swap the two half words halfword
24	reverse the byte order
31	bitwise reverse

If `grevi` were removed from this spec we would need to add a few new instructions in its place for operations such as bitwise reverse or endianness convertion.

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.

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]

4.2 Analysis of used encoding space

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

Table 4.2: XBitmanip encoding space ($log2$, i.e. in equivalent number of bits)

RV32		RV64		Instruction
2x	10	4x	10	CLZ, CLZW, 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
3x	15	6x	15	ANDC, BEXT, BDEP, ANDCW, BEXTW, BDEPW
4x	4	4x	4	C.NEG, C.NOT, C.CLZ, C.BREV

In the compressed encoding space XBitmanip needs the equivalent of a 6 bit encoding space, or

[1] http://svn.clifford.at/handicraft/2017/bitcode/
[2] https://github.com/cliffordwolf/bextdep

$\approx 0.13\%$ of the total ≈ 15.6 bits available. $(100/(2^{15.6-6}) \approx 0.13)$

On RV32, XBitmanip requires the equivalent of a ≈ 18.6 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.

$$\frac{log(2 \cdot 2^{10} + 12 \cdot 2^{15})}{log(2)} \approx 18.6$$

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 less than one-quarter of the encoding space of the `SLTIU` instruction.

$$\frac{log(4 \cdot 2^{10} + 20 \cdot 2^{15} + 4 \cdot 2^{16})}{log(2)} \approx 19.8$$

Chapter 5

Pseudo-Ops, Macros, Algorithms

This chapter contains a collection of psudo-ops, macros, and algorithms using the XBitmanip extension. 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.

5.1 GREVI Pseudo-Ops

Tables 5.1 and 5.2 list assembler pseudo-ops for commonly used `grevi` commands.

Pseudoinstruction	Base Instruction(s)	Meaning
`brev rd, rs`	`grevi rd, rs, 31`	bitwise reverse
`bswap rd, rs`	`grevi rd, rs, 24`	reverse the byte order
`bswap.h rd, rs`	`grevi rd, rs, 8`	reverse the byte order in each half word
`hswap rd, rs`	`grevi rd, rs, 16`	reverse the order of half words

Table 5.1: RV32 GREVI Pseudoinstructions.

Pseudoinstruction	Base Instruction(s)	Meaning
`brev rd, rs`	`grevi rd, rs, 63`	bitwise reverse
`bswap rd, rs`	`grevi rd, rs, 56`	reverse the byte order
`bswap.h rd, rs`	`grevi rd, rs, 8`	reverse the byte order in each half word
`bswap.w rd, rs`	`grevi rd, rs, 24`	reverse the byte order in each word
`hswap rd, rs`	`grevi rd, rs, 48`	reverse the order of half words
`hswap.w rd, rs`	`grevi rd, rs, 16`	reverse the order of half words in each word
`wswap rd, rs`	`grevi rd, rs, 32`	reverse the order of words

Table 5.2: RV64 GREVI Pseudoinstructions.

5.2 Bit Scanning Operations

Count Leading Zeros

This is trivially just the `clz` instruction:

```
clz a0, a0
```

Count Leading Ones

Simply invert before executing the `clz` instruction:

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

Find index of highest bit set (aka `ilog2`)

```
clz a0, a0
neg a0, a0
addi a0, a0, 32
```

Find index of lowest bit set

```
brev a0, a0
clz a0, a0
```

5.3 MIX/MUX Operations

MIX Operation

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

```
and t0, a1, a0
andc a0, a2, a0
or a0, a0, t0
```

MUX Operation

A MUX operation selects word a1 or a2 based on if the control word a0 is zero or nonzero, without branching. (The `snez` instruction is optional if a0 is already either 0 or 1.)

```
snez a0, a0
neg a0, a0
and t0, a1, a0
andc a0, a2, a0
or a0, a0, t0
```

5.4 Emulating other Bit Manipulation ISAs using macro-op fusion

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 5.3: Emulating other Bit Manipulation ISAs using macro-op fusion

x86 Ext	x86 Instruction	Bytes x86	Bytes RV	RISC-V Code
ABM	`popcnt`	5	2	`c.pcnt a0`
	`lzcnt`	5	2	`c.clz a0`
BMI1	`andn`	5	4	`andc a0, a2, a1`

	`bextr` (regs)[1]	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`
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`

[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.

[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.

x86 Ext	x86 Instruction	Bytes x86	Bytes RV	RISC-V Code
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`

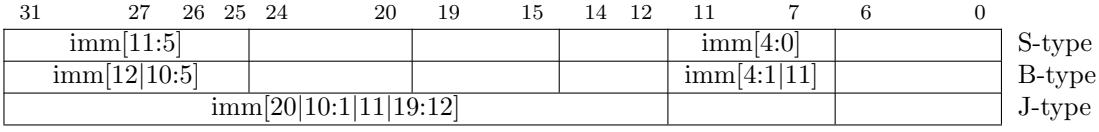
	`t1mskc`	5	10	`addi a0, a1, +1`
				`andc a0, a1, a0`
				`c.not a0`

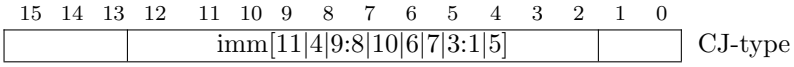
	`t1msk`	5	8	`addi a0, a1, -1` `andc a0, a0, a1`

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.

5.5 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:
  lui t0, 0x51574
  shuffle a0, a0, t0
  li t0, 0xeaa80055
  bext a0, a0, t0
  c.slli a0, 20
  c.srai a0, 19
  ret

// variant 1 (with shuffle/unshuffle/bext)
decode_j:
  lui t0, 0x0fff3
  unshuffle a0, a0, t0
  lui t0, 0x34349
  shuffle a0, a0, t0
  lui t0, 0x70fe4
  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/unshuffle)
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/unshuffle/bext)
decode_cj:
```

```
  rori a0, a0, 24
  lui t0, 0x399cb
  shuffle a0, a0, t0
  lui t0, 0xf2d30
  unshuffle a0, a0, t0
  li t0, 0x538a2046
  bext a0, a0, t0
  c.slli a0, 21
  c.srai a0, 20
  ret

// variant 2 (without shuffle/unshuffle/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
```

5.6 Butterfly Operations

The `grev`, `bext`, and `bdep` instructions are commonly implemented using butterfly circuits[3]. (Butterfly circuits are a powerful tool for implementing bit permutations.) However, sometimes one wants to use butterfly operations directly in a program and it is not immediately obvious how to

[3] http://programming.sirrida.de/bit_perm.html#butterfly

use GREVI and BDEP to perform butterfly operations.

One can easily implement a single butterfly stage using GREVI in just four instructions:

```
grevi t0, a0, N
and t0, t0, a1
andc a0, a0, a1
or a0, a0, t0
```

With `a0` containing the data input on entry and the result on exit. `N` is a power of two indicating which butterfly stage to implement, and `a1` contains an XLEN bitmask derived from the XLEN/2 butterfly control word.

This bitmask can either be pre-computed or, if it needs to be generated on-the-fly, it can easily be calculated with the help of BDEP. For example for the first butterfly stage (using a0 as input and output):

```
li t0, 0x55555555
bdep t0, a0, t0
slli a0, t0, 1
or a0, a0, t0
```

Likewise for the second butterfly stage:

```
li t0, 0x33333333
bdep t0, a0, t0
slli a0, t0, 2
or a0, a0, t0
```

For the third stage:

```
li t0, 0x0f0f0f0f
bdep t0, a0, t0
slli a0, t0, 4
or a0, a0, t0
```

For the fourth stage:

```
li t0, 0x00ff00ff
bdep t0, a0, t0
slli a0, t0, 8
or a0, a0, t0
```

And for the fifth stage:

```
li t0, 0x0000ffff
and t0, a0, t0
slli a0, t0, 16
or a0, a0, t0
```

Chapter 6

Temporary encoding using NSE opcode space

This is currently a non-standard extension so we need instruction encodings using "custom" opcodes. The long-term goal for this spec is to become the official B extension. So the encoding in this chapter is a temporary solution for the meantime, allowing us to build compilers and processors to evaluate the extension.

The temporary encoding uses the funct3=000 space in opcode *custom-0* for the uncompressed instructions and the nzuimm[5]=1, nzuimm[4:0]=0 space in C.SLLI for the compressed RV32 instructions.

On RV64 there is no NSE space in the C instruction listing, therefore no encoding for compressed bit manipulation instructions is defined on RV64. (The availability of compressed bit manipulation instructions on RV32 should be sufficient to evaluate the utility of those instructions.)

15 14 13	12	11 10	9 8 7	6 5 4 3 2	1 0	
000	nzuimm[5]	rs1/rd≠0		nzuimm[4:0]	10	C.SLLI *(HINT, rd=0; ~~RV32 NSE, nzuimm[5]=1~~)*
000	1	00	rs2′/rd′	0	10	C.NEG
000	1	01	rs1′/rd′	0	10	C.NOT
000	1	10	rs1′/rd′	0	10	C.CLZ
000	1	11	rs1′/rd′	0	10	C.BREV

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type

RV32 XBitmanip Instruction Set

0000000	00000	rs1	000	rd	0001011	CLZ
0001000	00000	rs1	000	rd	0001011	PCNT
0010000	rs2	rs1	000	rd	0001011	GREV
0010100	mode	rs1	000	rd	0001011	GREVI
0011000	rs2	rs1	000	rd	0001011	ANDC
0100000	rs2	rs1	000	rd	0001011	BEXT
0101000	rs2	rs1	000	rd	0001011	BDEP
1000100	shamt	rs1	000	rd	0001011	SLOI
1010100	shamt	rs1	000	rd	0001011	SROI
1110100	shamt	rs1	000	rd	0001011	RORI
1000000	rs2	rs1	000	rd	0001011	SLO
1010000	rs2	rs1	000	rd	0001011	SRO
1100000	rs2	rs1	000	rd	0001011	ROL
1110000	rs2	rs1	000	rd	0001011	ROR

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type

RV64 XBitmanip Instruction Set

0000000	00000	rs1	000	rd	0001011	CLZ
0000010	00000	rs1	000	rd	0001011	CLZW
0001000	00000	rs1	000	rd	0001011	PCNT
0001010	00000	rs1	000	rd	0001011	PCNTW
0010000	rs2	rs1	000	rd	0001011	GREV
0010010	rs2	rs1	000	rd	0001011	GREVW
001010	mode	rs1	000	rd	0001011	GREVI
0010110	mode	rs1	000	rd	0001011	GREVIW
0011000	rs2	rs1	000	rd	0001011	ANDC
0011010	rs2	rs1	000	rd	0001011	ANDCW
0100000	rs2	rs1	000	rd	0001011	BEXT
0100010	rs2	rs1	000	rd	0001011	BEXTW
0101000	rs2	rs1	000	rd	0001011	BDEP
0101010	rs2	rs1	000	rd	0001011	BDEPW
100010	shamt	rs1	000	rd	0001011	SLOI
1000110	shamt	rs1	000	rd	0001011	SLOIW
101010	shamt	rs1	000	rd	0001011	SROI
1010110	shamt	rs1	000	rd	0001011	SROIW
111010	shamt	rs1	000	rd	0001011	RORI
1110110	shamt	rs1	000	rd	0001011	RORIW
1000000	rs2	rs1	000	rd	0001011	SLO
1000010	rs2	rs1	000	rd	0001011	SLOW
1010000	rs2	rs1	000	rd	0001011	SRO
1010010	rs2	rs1	000	rd	0001011	SROW
1100000	rs2	rs1	000	rd	0001011	ROL
1100010	rs2	rs1	000	rd	0001011	ROLW
1110000	rs2	rs1	000	rd	0001011	ROR
1110010	rs2	rs1	000	rd	0001011	RORW

Chapter 7

Change History

Table 7.1: 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 scattergaher
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
????-??-??	0.34	Add GREV and compressed instructions
		Restructure document: Move discussions to extra sections
		Add FAQ, add analysis of used encoding space
		Add chapter on Pseudo-Ops, Macros, Algorithms
		Add chapter on temporary encoding using custom opcode space
		Add chapter on alternatives to bext/bdep