RISC-V XBitmanip Extension

Document Version 0.33

Editors: Clifford Wolf[1], Rex McCrary [2]
[1]Symbiotic GmbH, [2]Advanced Micro Devices, Inc.
clifford@symbioticeda.com, rmccrary@amd.com

April 7, 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

1.1 ISA Extension Proposal Scoring Criteria

Any proposed changes to the ISA should be evaluated or scored according to the following criteria. Proposals which score higher according to the criteria should be considered as higher priority than proposals that score lower. Proposals that score consistently low should be rejected. Each of he five categories have equal weighting and each contribute up to 20 points toward an instructions total score from 0 to 100.

- 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 without new instruction formats. "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.

1.2.1 Next steps

- Allocate instruction opcodes for the instructions so we can build experimental cores implementing the extension and compilers and other software tools for generating RISC-V program utilizing the instruction. For now it would be sufficient to allocate those opcodes in one of the major opcodes reserved for cutsom extensions.
- 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.

1.3 Summary of Instructions

The following machine instruction are listed below, followed by a table that lists pseudo instructions.

Table 1.1: Machine Instructions

Instructions	Primary Applications
clz	Count Leading Zeros
pcnt	Population Count
grevi	Generalized Reverse
slo	Shift left Ones
sloi	Shift left Ones Immediate
sro	Shfit Right Ones
sroi	Shift Right Ones Immediate
rol	Rotate Left
ror	Rotate Right
rori	Rotate Right Immediate
andc	AND Complement

Instructions	Primary Applications
bext	Bit Extract (Gather)
bdep	Bit Depsoit (Scatter)

1.4 Summary of Psuedo Ops

The following ops are supported by replacing with one or more machine operations. See instruction details for examples.

Table 1.2: Psuedo Ops

Instructions	Primary Applications	No. of Instr
flb, fls	Find Last Bit Set, Find Last Set: clz, sub	2
ctz	Count Trailing Zeros: grevi, clz	2
fbs, ffs	First Bit Set, Find First Set: grevi, clz	2
roli	Rotate Left Immediate: rori with adjusted immedate value	1
bfdep	Bit Field Deposit	1
bfext	Bit Field Extract	1

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.

```
// clz C implementation
uint_xlen_t clz(uint_xlen_t rs1)
{
    for (int count = 0; count < XLEN; count++)
        if ((rs1 << count) >> (XLEN-1))
            return count;
    return XLEN;
}
```

2.1.1 Encoding

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

`clz` is encoded as a standard I-type opcode, with a single source register and a 12-bit signed immediate.

2.1.2 Related Pseudo-instructions

The common operation of 'finding the index of the highest bit set' (also called `ilog2` or `bsr` or `find last set`) is computed as `XLEN-clz(rs1)`.

Counting trailing zeros is easily implemented by combining `clz` with the `grevi` instruction.

2.1.3 Criteria

The criteria evaluation is shown below:

- These operations all fit easily into the RISCV instruction encoding and philosophy.
- The hardware to implement them is fairly simple, and can be done in a logarithmic number of stages in parallel.
- They all have current compiler and standard library support, and are standardized intrinsics in many of those.
- The threshold criteria gets a little complicated: Every one of these operations can be implemented in 2-3 instructions in terms of each-other, so we only need one of them in hardware, and the rest can be pseudo-instructions.

However, implementing any of these instructions can also be done with a combination of `ctz` and a bitwise reverse (`brev`) in 1-2 instructions. Similarly, `ctz` can be implemented in terms of these functions with a `brev` in 1-2 instructions.

Without `brev` AND `ctz`, however, there is no simple way to construct this without a lot of instructions. The algorithm above in C is a nearly asymptotically optimal implementation. It seems to pass the threshold criteria in this case.

Benchmarks and Applications TBD

2.1.4 References

https://en.wikipedia.org/wiki/Find_first_set#CLZ

https://fgiesen.wordpress.com/2013/10/18/bit-scanning-equivalencies/

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.

2.2.1 Functionality definition

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

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

2.2.2 Encoding

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

`pcnt` is encoded as a standard I-type opcode, with one source register and an immediate. There is no `pcnti`, because it only has one argument, so it would be redundant.

Implementation Commentary

`pcnt` can be implemented in parallel, similarly to an adder.

2.2.3 Related Pseudo-Instructions

```
//odd parity
parity rOut,rIn:
    pcnt rPopulation,rIn
    andi rOut,rPopulation,0x1
```

2.2.4 Extensions: `pcnt.w` (R64I+), `pcnt.d` (R128I+)

On RV64B+, the appropriate versions of pcnt that operate on sub-portions of the register are defined. On these platforms, the input source is set to zero for the high portions of the register. See `addi.w` and `addi.d` in the R64I/R128I specifications.

2.2.5 Justification

This operation is one of the 'core' bitwise instructions that is nearly universal on other ISAs. It is supported as an intrinsic on GCC, LLVM, and MSVC, and is supported in hardware on a plethora of platforms.

2.2.6 References

https://en.wikipedia.org/wiki/Hamming_weight

https://graphics.stanford.edu/~seander/bithacks.html

2.3 Generalized Reverse (`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.

2.3.1 Functionality definition

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

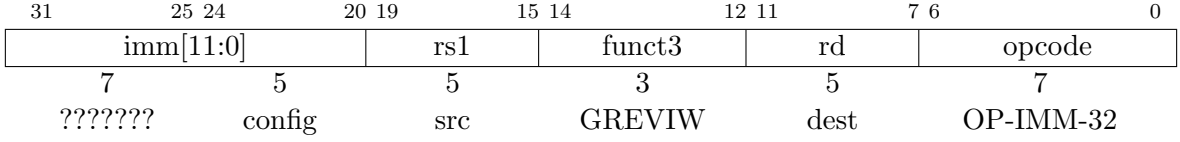
```
uint32_t grevi32(uint32_t rs1,int12_t immed)
{
    uint32_t x = rs1;
    uint5_t k = (uint12_t)immed;
    if(k &  1) x = ((x & 0x55555555) << 1) | ((x & 0xAAAAAAAA) >> 1);
    if(k &  2) x = ((x & 0x33333333) << 2) | ((x & 0xCCCCCCCC) >> 2);
    if(k &  4) x = ((x & 0x0F0F0F0F) << 4) | ((x & 0xF0F0F0F0) >> 4);
    if(k &  8) x = ((x & 0x00FF00FF) << 8) | ((x & 0xFF00FF00) >> 8);
```

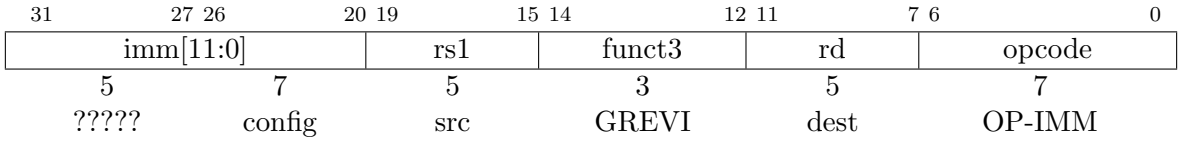
```
    if(k & 16) x = ((x & 0x0000FFFF) <<16) | ((x & 0xFFFF0000) >> 16);
    return x;
}

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+.

2.3.2 Encoding





`grevi` is encoded as a standard I-type opcode with one source register and one immediate.

There is no similar `grev`, R-type opcode with a dynamic argument for the function select, because it seems unlikely that functionality would ever be useful compared to the hardware implementation complexity required, and since the function select parameter only ranges from `0:XLEN-1`, it could be easily implemented as an XLEN-sized jump table by the compiler.

Implementation Commentary

When implementing this circuit, the masks and shifts should be used to select wires and the k bits should be used to AND out the computation at each stage. It uses log2(XLEN) stages of parallel circuitry (e.g., a "butterfly circuit").

2.3.3 Related Pseudo-Instructions

Related Pseudoinstruction List

```
// reverse the bits in a register (this works on all platforms)
rev rOut,rIn:
    grevi rOut,rIn,-1

// reverse the bits in each byte, but leave the bytes in the same order.
rev.b rOut,rIn:
    grevi rOut,rIn,7

// reverse the bits in each halfword, but leave the halfwords in the same order
rev.h rOut,rIn:
    grevi rOut,rIn,15

// reverse the bits in each word, but leave the words in the same order (e.g. RV64IB+)
rev.w rOut,rIn:
    grevi rOut,rIn,31

// reverse the bits in each dword, but leave the dwords in the same order (e.g. RV128IB+)
rev.d rOut,rIn:
    grevi rOut,rIn,63

// reverse the byte order of an entire register
bswap rOut,rIn:
    grevi rOut,rIn,-8

// reverse the byte order of each halfword in the register, but leave them in the same
// order (e.g, on RV32IB, this reverses the byte order of the high and lo halfwords)
bswap.h rOut,rIn:
    grevi rOut,rIn,8

// reverse the byte order each word in the register (e.g, on RV64IB/RV128IB, this
// reverses the byte order of each of the 2/4 words)
bswap.w rOut,rIn:
    grevi rOut,rIn,24

// reverse the byte order of each dword in the register (e.g, on RV128IB, reverse the
// byte order of the high and lo words)
bswap.d rOut,rIn:
    grevi rOut,rIn,56

// reverse the halfword order of an entire register
hswap rOut,rIn:
    grevi rOut,rIn,-16
```

```
// reverse the halfword order of each word in the register
hswap.w rOut,rIn:
    grevi rOut,rIn,16

// reverse the halfword order of each dword in the register (e.g. RV128IB+, there are 2
// dwords)
hswap.d rOut,rIn:
    grevi rOut,rIn,48

// reverse the word order of an entire register (only meaningful on RV64IB+)
wswap rOut,rIn:
    grevi rOut,rIn,-32

// reverse the word order of each dword in the register (e.g. RV128IB+, there are 2 dwords)
wswap.d rOut,rIn:
    grevi rOut,rIn,32

// swap the nibbles of each byte in the whole register (useful for converting to/from
// bigendian hex)
nswap.b rOut,rIn:
    grevi rOut,rIn,4

// Etc.  A total of XLEN different possible operations can be constructed.
```

2.3.4 Justification

Quoted from page 102 of Hacker's Delight:

https://books.google.com/books?id=iBNKMspIlqEC&lpg=PP1&pg=RA1-SL20-PA2#v=onepage&q&f=false

> For k=31, this operation reverses the bits in a word. For k=24, it reverses the bytes in a word. For k=7, it reverses the bits in each byte, without changing the positions of the bytes. For k=16, it swaps the left and right halfwords of a word, and so on. In general, it moves the bit at positing m to position m XOR k. It can be implemented in hardware very similarly to the way a rotate shifter is usually implemented (five stages of MUX, with each stage controlled by a bit of the shift amount k)
>
> This strongly suggests that this generic instruction could allow us to use a great deal of other bitwise operations as pseudo-instructions.

2.3.5 References

Hackers Delight, Chapter 7.1, "Generalized Bit Reversal" in

https://books.google.com/books?id=iBNKMspIlqEC&lpg=PP1&pg=RA1-SL20-

PA2#v=onepage&q&f=false

http://hackersdelight.org/

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.

2.4.1 Functionality definition

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

```
// Implementation of ‘slo(i)‘ in C
uint_xlen_t slo(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int12_t amount=(rs2 & (XLEN-1));
    uint_xlen_t mask=((1 << amount)-1);
    return (rs1 << amount) | mask;
}
uint_xlen_t sloi(uint_xlen_t rs1, int12_t immed)
{
    int12_t amount=(immed & (XLEN-1));
    uint_xlen_t mask=((1 << amount)-1);
    return (rs1 << amount) | mask;
}

// Implementation of ‘sro(i)‘ in C
uint_xlen_t sro(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int12_t amount=(rs2 & (XLEN-1));
    uint_xlen_t mask=~((1 << (XLEN-amount))-1);
    return (rs1 >> amount) | mask;
}
uint_xlen_t sroi(uint_xlen_t rs1, int12_t immed)
{
    int12_t amount=(immed & (XLEN-1));
    uint_xlen_t mask=~((1 << (XLEN-amount))-1);
    return (rs1 >> amount) | mask;
}
```

2.4.2 Encoding

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
???????	shamt	src	SLOIW	dest	OP-IMM-32
???????	shamt	src	SROIW	dest	OP-IMM-32

31 27	26 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
5	7	5	3	5	7
?????	shamt	src	SLOI	dest	OP-IMM
?????	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
???????	src2	src1	SRO	dest	OP
???????	src2	src1	SLO	dest	OP
???????	src2	src1	SROW	dest	OP-32
???????	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.

Implementation Commentary

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.

2.4.3 Extensions: `s(l/r)o(i).w` (R64I+), `s(l/r)o(i).d` (R128I+)

On RV64B+, the appropriate versions of shift-ones that operate on sub-portions of the register are defined. On these platforms, the input source is set to zero for the high portions of the register, and the shift only occurs on the lower portions of the register. See `slli.w` and `slli.d` in the R64I/R128I specifications. See `slli.w` and `slli.d` in the R64I/R128I specifications.

2.4.4 Related Pseudo-Instructions

```
// builds a mask in the low-order bits up to a certain point
maski rOut,iWidth:
    sloi rOut,r0,iWidth

// builds a mask in the low-order bits up to a certain point
mask rOut,rIn:
    slo rOut,r0,rIn

// extracts a bitfield of length width from offset and moves it down to the bottom
bfextracti rOut,rIn,iWidth,iOffset:
    slli rTop,rIn,XLEN-iWidth-iOffset
    srli rOut,rTop,XLEN-iWidth

// updates a bitfield of length width at offset in rCurrent with the value from rIn
bfupdatei rOut,rCurrent,rIn,iWidth,iOffset:
    sloi rTop,rIn,<XLEN-iWidth>
    sroi rField,rTop,<XLEN-iWidth-iOffset>
    and rOut,rCurrent,rField

// sets a bitfield of length-width at offset to 1
bfseti rOut,rCurrent,iWidth,iOffset:
    sloi rMask,r0,iWidth
    slli rMask,rMask,iOffset
    or rOut,rCurrent,rMask

// sets a bitfield of length-width at offset to 0
bfcleari rOut,rCurrent,iWidth,iOffset:
    sloi rMask,r0,iWidth
    slli rMask,rMask,iOffset
    andc rOut,rCurrent,rMask
```

2.4.5 Justification

This instruction can be used to create masks, which is an incredibly common operation for modifying the bitfield structures.

2.4.6 References

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'.

2.5.1 Functionality definition

```
// Implementation of ‘rol‘ in C
uint_xlen_t rol(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int12_t amount=(rs2 & (XLEN-1));
    return (rs1 << amount) | (rsl >> (XLEN-amount));
}

// Implementation of ‘ror(i)‘ in C
uint_xlen_t ror(uint_xlen_t rs1, uint_xlen_t rs2)
{
    int12_t amount=(rs2 & (XLEN-1));
    return (rs1 >> amount) | (rsl << (XLEN-amount));
}
int_xlen_t rori(uint_xlen_t rs1, int12_t immed)
{
    int12_t amount=(immed & (XLEN-1));
    return (rs1 >> amount) | (rs1 << (XLEN-amount));
}
```

2.5.2 Encoding

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
???????	shamt	src	ROLIW	dest	OP-IMM-32
???????	shamt	src	RORIW	dest	OP-IMM-32

31 27	26 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
5	7	5	3	5	7
?????	shamt	src	ROLI	dest	OP-IMM
?????	shamt	src	RORI	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
???????	src2	src1	RORW	dest	OP-32
???????	src2	src1	ROLW	dest	OP-32
???????	src2	src1	ROR	dest	OP
???????	src2	src1	ROL	dest	OP

`ror(i),rol(i)` 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:

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

Implementation Commentary

2.5.3 Extensions: `ro(r/l)(i).w` (R64I+),`ro(r/l)(i).d` (R128I+)

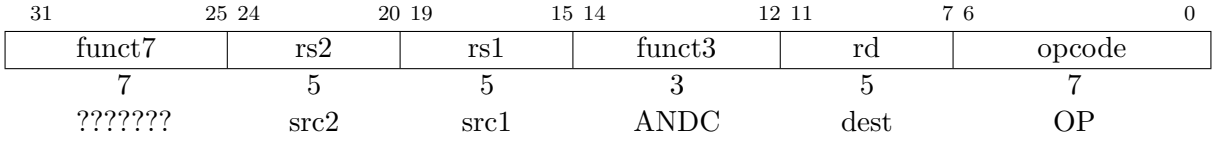
On RV64B+, the appropriate versions of rotate that operate on sub-portions of the register are defined. On these platforms, the input source is set to zero for the high portions of the register, and the shift only occurs on the lower portions of the register. See `slli.w` and `slli.d` in the R64I/R128I specifications.

2.5.4 Justification

This instruction is very useful for cryptography, hashing, and other operations.

2.5.5 References

2.6 And-with-complement (`andc`)



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.

2.6.1 Functionality definition

```
// Implementation of ‘andc‘ in C
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 instruction encoding is to be decided, however.

2.6.2 Related Pseudo-Instructions

```
// this selects the corresponding bit from the two arguments A and B, based on
// whether or not Control is high.
// out[i] = Control[i] ? A[i] : B[i]
mix rOut,rControl,rA,rB:
    and rTmp,rA,rControl
    andc rTmp2,rB,rControl
    or rOut,rTmp,rTmp2

// same as MIX but only use if the first bit of C is true (as in from a comparator)
// C can only be zero or 1, so if you need a SGT to implement nonzero comparator
// its 5 instructions in total.
mux rOut,rControl,rA,rB:
    neg rC2,rControl
    and rTmp,rA,rC2
    andc rTmp2,rB,rC2
    or rOut,rTmp,rTmp2
```

2.6.3 Justification

http://svn.clifford.at/handicraft/2017/bitcode/

2.6.4 References

http://www.hackersdelight.org/basics2.pdf

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

This is an R-type instruction that implements the generic bit erxtract 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.

2.7.1 Functionality definition

```
// Implementation of ‘bext‘ in C
uint_xlen_t bext(uint_xlen_t v, uint_xlen_t mask)
{
    uint32_t c = 0, m = 1;
    while (mask) {
        uint_xlen_t b = mask & -mask;
        if (v & b)
            c |= m;
        mask -= b;
        m <<= 1;
    }
    return c;
}

// Implementation of ‘bdep‘ in C
uint_xlen_t bdep32(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;
}
```

2.7.2 Encoding

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 instruction in the base spec. The exact instruction encoding is to be decided, however.

2.7.3 Related Pseudo-Instructions

TBD

2.7.4 Justification

http://svn.clifford.at/handicraft/2017/permsyn/

2.7.5 References

http://programming.sirrida.de/bit_perm.html#gather_scatter

Hackers Delight, Chapter 7.1, “Compress, Generalized Extract” in

https://books.google.com/books?id=iBNKMspIlqEC&lpg=PP1&pg=RA1-SL20-PA2#v=onepage&q&f=false

http://hackersdelight.org/

https://github.com/cliffordwolf/bextdep

Chapter 3

Change History

Table 3.1: 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