# RISC-V BitManip

ISA Extensions, Hardware, Tools, Applications

# Thanks to our task group members!

Thanks to the 98 individual members of the task group for their valuable contributions. We also like to thank the following companies for their support in varying capacity.

# RISC-V BitManip Task Group

We define a set of RISC-V standard extensions for with the goal of:

- increasing performance (via fewer instructions & branch-reduced code)
- decreasing code size (via fewer instructions)
- decreasing power (via fewer and more specialized instructions)

by adding bit-manipulation instructions to the ISA.

# BitManip Design Criteria

- No added state
  - BitManip is only using GPRs
- No new instruction formats*
  - Use existing 32-bit R-type and I-type instruction formats
  - Use existing 32-bit R4-type instruction format for ternary instructions
  - Use existing shift-immediate encoding space for new shift-like instructions with immediates
- New instructions must replace at least 3 existing instructions*
  - For patterns of 2 instructions high-end cores can fuse instructions to macro-ops
- Hardware Simplicity
  - New instructions must be easy to implement with reasonable hardware cost
  - ISC-licensed (BSD-style) Verilog reference implementations are provided to this end

# BitManip Design Criteria -- Bending the rules a bit

- No new instruction formats
  - We add integer ternary instructions in Zbt (will not be included in "B")
  - Uses R4-type instruction format for non-immediate version
  - Add the obvious format for immediate funnel shift instruction
- New instructions must replace at least 3 existing instructions
  - We make a few exceptions for instructions that ...
    1. are used reasonably frequently,
    2. are cheap in terms of encoding space,
    3. <u>and</u> are very cheap in terms of HW cost.

# BitManip Grouping

Proposed grouping based on

- Implementation Cost (size)
- Usage in particular applications

Individual “Zb?” extensions

“Base” Bitmanip instructions in “Zbb”

“B” will be a large “best of” selection

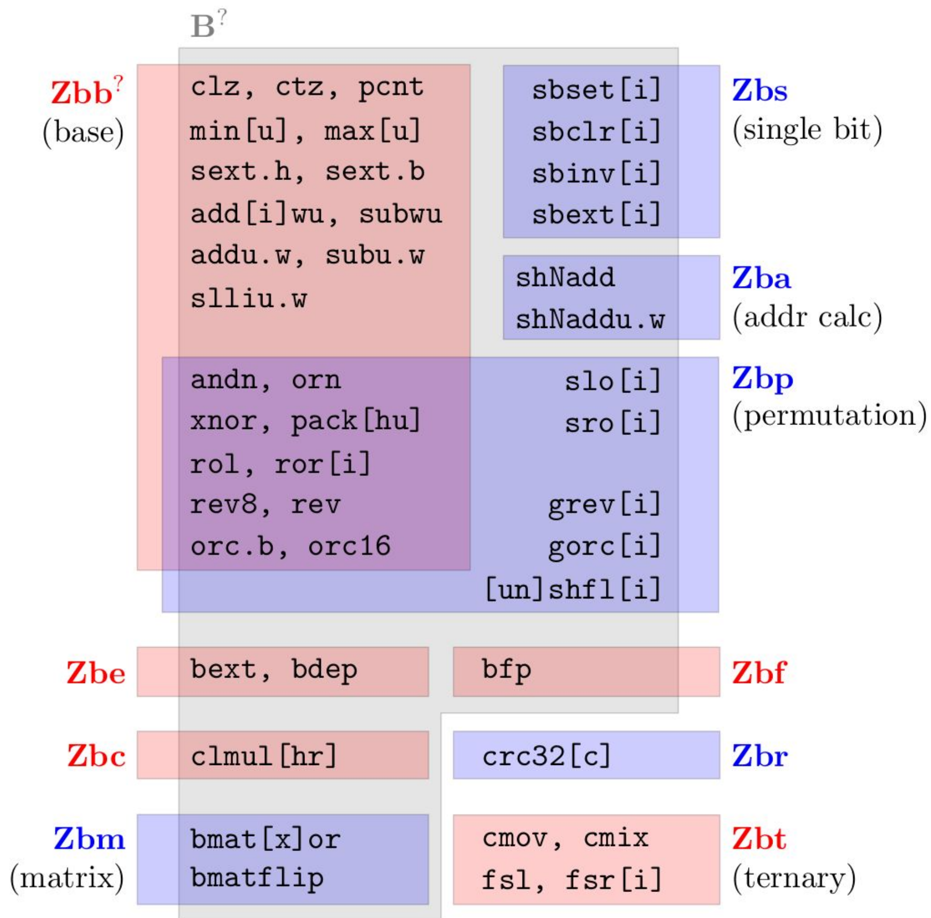
“B” will not contain “Zbt”

Where are these instructions useful to you?

- Operation systems
- Algorithms
- Applications

Final grouping will be decided in consultation with

**Platform Task Group**



# Roadmap: Where do we go from here?

- Reference Implementations
  - Hardware
  - cycle-aware models
- Compiler support
- Performance analysis - what helps and where
- Reformat Specification to fit in with other extensions
- Submit for ratification:
  - All at once, or
  - Most needed subsets?

# Verilog Reference Implementation

On the following slides we compare area of our BitManip implementation against Rocket MulDiv, using the following cell library:

| Module | Instructions |
|---|---|
| rvb_bextdep | bext bdep grev gorc shfl unshfl |
| rvb_clmul | clmul clmulr clmulh |
| rvb_shifter | sll srl sra slo sro rol ror fsl fsr slliu.w sbset sbclr sbinv sbext bfp |
| rvb_bmatxor | bmatxor bmator |
| rvb_simple | min max minu maxu andn orn xnor pack cmix cmov addiwu addwu subwu adduw subuw |
| rvb_bitcnt | clz ctz pcnt bmatflip |
| rvb_crc | crc32.[bhwd] crc32c.[bhwd] |
| rvb_full | All of the above |

| Cell | Gate Count |
|---|---|
| NOT | 0.5 |
| NAND | 1 |
| NOR | 1 |
| XOR | 3 |
| XNOR | 3 |
| DFF | 4 |

| Cell | Gate Count |
|---|---|
| AOI3 | 1.5 |
| OAI3 | 1.5 |
| AOI4 | 2 |
| OAI4 | 2 |
| NMUX | 2.5 |
| MUX | 3 |

Regarding timing we evaluate the longest paths for rvb full and rocket-chip MulDiv, measured in gate delays:

| | RV32 | RV64 |
|---|---|---|
| rvb_full | 30 | 57 |
| MulDiv | 43 | 68 |

# Verilog Reference Implementation

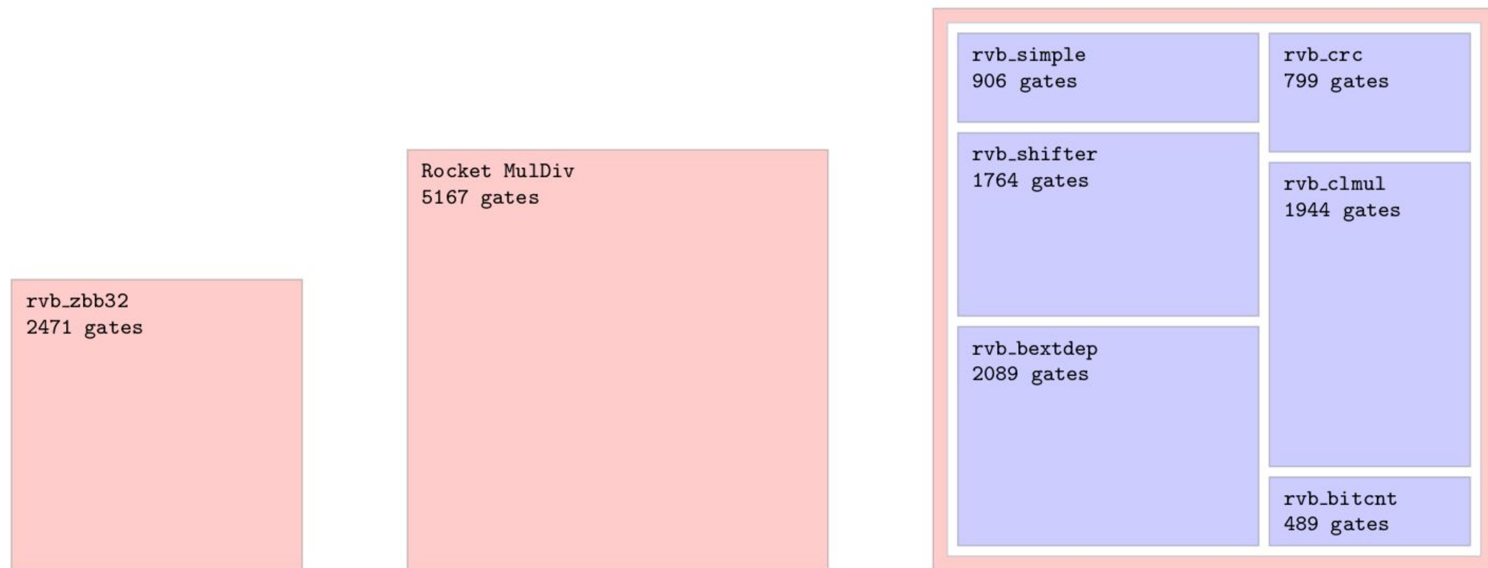



Figure 3.1: Area of 32-bit Rocket MulDiv core (center) compared to a complete implementation of all 32-bit instructions proposed in this specification (right), and the 32-bit "Zbb" extension (left).

# Verilog Reference Implementation

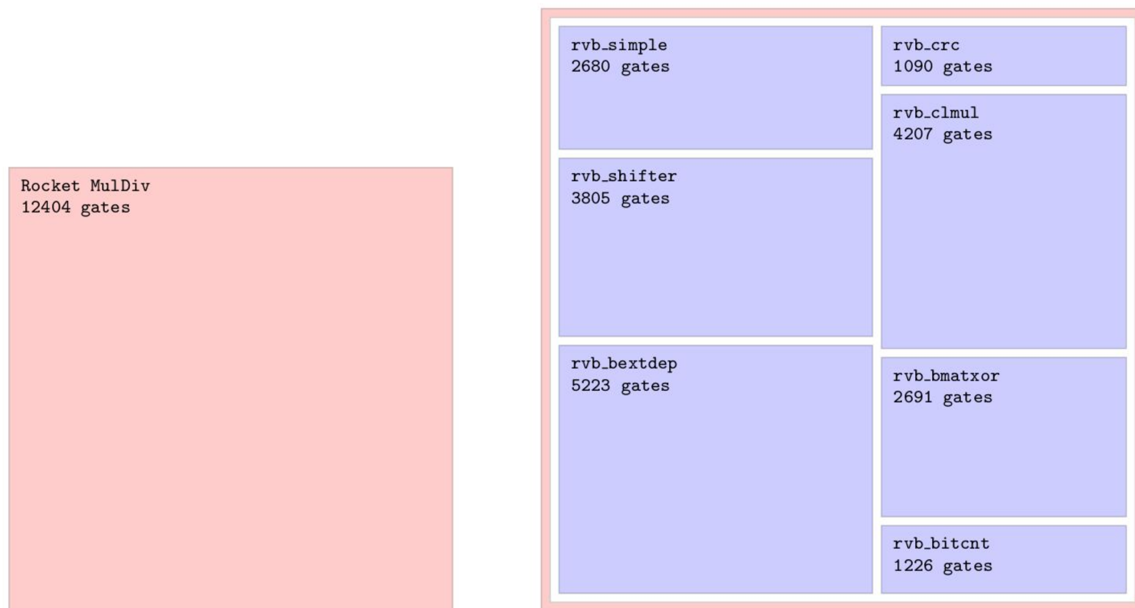



Figure 3.2: Area of 64-bit Rocket MulDiv core (left) compared to a complete implementation of all 64-bit instructions proposed in this specification (right).

# GNU Tool Chain Support

Binutils/GDB Support:

- Hand coded binutils almost done, all BitManip instructions fully supported
  - Only a few pseudo-instructions are still missing
- CGEN based assembler/disassembler and GDB simulator available for v0.92
  - see CGEN Tutorial

GCC C/C++ Support:

- Semantic information added, so instruction selection picks up bitmanip automatically
  - some existing RTL instructions special-cased, e.g. $2^n$ immediate uses `sbseti`
- Bitmanip intrinsics implemented (more to be done)

RISC-V ISA Sim (aka “Spike”):

- Full support for BitManip instructions

See “riscv-bitmanip” branch in the respective github repositories.

# Clang/LLVM Tool Chain Support

C/C++ support for v0.92

- BitManip instructions added to the integrated assembler
- Same set of intrinsics as GCC
- Additional codegen pattern matching for bitmanip instructions
  - 16% code size reduction in Embench nettle-aes benchmark at -Os

Not yet added to `-march`. Invocation:

```
riscv32-unknown-elf-clang -Xclang -target-feature -Xclang +b ...
```

# C Intrinsics for RISC-V BitManip

Use #include <rvintrin.h> to access our C intrinsics. Usage example:

```
#include <rvintrin.h>

int find_nth_set_bit(unsigned int value, int n) {
    return _rv32_ctz(_rv32_bdep(1 << n, value));
}
```

- Use _rv32_* for intrinsics operating on “int32_t”.
- Use _rv64_* for intrinsics operating on “int64_t”.
- Use _rv_* for intrinsics operating on “long”.

Set “#define RVINTRIN EMULATE” for plain-C “emulation” functions instead. For example, for writing and testing algorithms on non-RISC-V platforms.

# Instruction encodings and used encoding space

| funct7 | funct3 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 30\|29\|27\|25 | 001 | 101 | 000 | 010 | 011 | 100 | 110 | 111 |
| -00-0-0 | SLL | SRL | ADD | SLT^ | SLTU^ | XOR^ | OR^ | AND^ |
| -10-0-0 | | SRA | SUB | | | XNOR^ | ORN^ | ANDN^ |
| -00-0-1 | MULH^(2) | DIVU (2) | MUL | MULHSU^ | MULHU^ | DIV | REM | REMU |
| -10-0-1 | (2) | (2) | | | | | | |
| -00-1-0 | SHFL (4) | UNSHFL | ADDU.W (1) | | BMATOR^ | PACK | BEXT | PACKH^ |
| -10-1-0 | SBCLR | SBEXT | SUBU.W (1) | | BMATXOR^ | PACKU | BDEP | BFP |
| -00-1-1 | CLMUL (2) | MAX^(2) | ADDWU (1) | CLMULR | CLMULH | MIN^ | MINU^ | MAXU^ |
| -10-1-1 | (2) | (2) | SUBWU (1) | | | | | |
| -01-0-0 | SLO | SRO | (SH4ADD) | SH1ADD | | SH2ADD | SH3ADD | |
| -11-0-0 | ROL (3) | ROR | | | | | | |
| -01-0-1 | (2) | (2) | | | | | | |
| -11-0-1 | (2) | (2) | | | | | | |
| -01-1-0 | SBSET | GORC | | | | | | |
| -11-1-0 | SBINV | GREV | | | | | | |
| -01-1-1 | (2) | (2) | | | | | | |
| -11-1-1 | (2) | (2) | | | | | | |

This table shows ¼ of the OP/OP-32 code space for binary instructions. (Bits 31 and 28 are always zero so far.)

Bit 26 is used to select ternary instructions.

See BitManip spec for details.

# Opcode Assignment

- The BitManip Specification includes proposed instruction encodings
  - Be aware these are subject to change
- The actual encodings will be decided by the **ISA Standing Committee**
  - We are *breaking ground* in being their first customer
  - We are submitting a proposal on a flow for opcode assignment on the tech mailing list
    - Please review and provide your feedback
- We will be closing the loop with other task groups that have/need similar operations to come up with a instructions that meet the needs of each domain while sharing mnemonics and encodings
  - Crypto scalar extensions
  - Packed-SIMD

# clz, ctz, pcnt (Zbb)

```
clz rd, rs
```

count number of zero bits at the MSB end of rs

```
ctz rd, rs
```

count number of zero bits at the LSB end of rs

```
pcnt rd, rs
```

count number of set bits in rs

clz/ctz return XLEN when rs is zero

# sext.h, sext.b (Zbb)

- `sext.b rd, rs`
  - sign extend the value in the 8 LSB bits
- `sext.h rd, rs`
  - sign extend the value in the 16 LSB bits
- Pseudo instructions for the remaining cases
  - `sext.w rd, rs` = `addiw rd, rs, 0`
  - `zext.b rd, rs` = `andi rd, rs, 255`
  - `zext.h rd, rs` = `pack[w] rd, rs, zero`
  - `zext.w rd, rs` = `pack rd, rs, zero`

# min[u], max[u] (Zbb)

```
min[u] rd, rs1, rs2
```

the [un]signed minimum of rs1 and rs2

```
max[u] rd, rs1, rs2
```

the [un]signed maximum of rs1 and rs2

# slo[i], sro[i], (Zbb, Zbp)

- `slo rd, rs1, rs2`
- `sloi rd, rs, imm`
- `sro rd, rs1, rs2`
- `sroi rd, rs, imm`

shift left/right but shift-in one bits instead of zero bits

# andn, orn, xnor (Zbb, Zbp)

```
andn rd, rs1, rs2
orn  rd, rs1, rs2
xnor rd, rs1, rs2
```

like `and/or/xor` but with rs2 inverted

Using the same instruction bit (bit 30) that already selects add/sub, and thus already controls an inverter on rs2.

# sh[123]add, sh[123]addu.w (Zba)

`sh[123]add`

like `add`, but left-shift rs1 by 1, 2, or 3 bits before addition

`sh[123]addu.w  (RV64)`

like sh[123]`add`, but with bits XLEN-1:32 of rs1 zeroed before the shift

These instructions are most useful for address calculations when accessing arrays of 16-bit, 32-bit, or 64-bit elements.

# add[i]wu, subwu, addu.w, subu.w, slliu.w (Zbb, RV64)

`addu.w  subu.w`

like `add`/`sub`, but with bits XLEN-1:32 of rs2 zeroed out

`add[i]wu, subwu`

like `add`/`sub`, but with bits XLEN-1:32 of result zeroed out

`slliu.w`

like `slli`, but with bits XLEN-1:32 of rs1 zeroed out

# pack, packu, packh (Zbb)

`pack rd, rs1, rs2`

concatenate the LSB halves of rs1 and rs2 to form rd

`packu rd, rs1, rs2`

concatenate the MSB halves of rs1 and rs2 to form rd

`packh rd, rs1, rs2`

concatenate the LSB bytes of rs1 and rs2 and zero-extend to form rd

# sbset[i], sbclr[i], sbinv[i], sbext[i] (Zbs)

```
sbset rd, rs1, rs2      sbseti rd, rs, imm
sbclr rd, rs1, rs2      sbclri rd, rs, imm
sbinv rd, rs1, rs2      sbinvi rd, rs, imm
```

take the value of rs1/rs and set/clear/invert bit rs2/imm

```
sbext rd, rs1, rs2      sbexti rd, rs, imm
```

return bit rs2/imm from rs1/rs (zero extended)

# Bit permutation instructions

Bit permutations instructions reorder the bits in a register. The new location of each bit (index) is a function the current location (index).

- Rotate
  - `index' = (index +/- shamt) mod XLEN`
- Generalized Shuffle `[un]shfl[i]`
  - `index' = permute(index)`
- Generalized Reverse
  - `index' = index XOR bitmask`

# Bit permutation instructions: Rotate

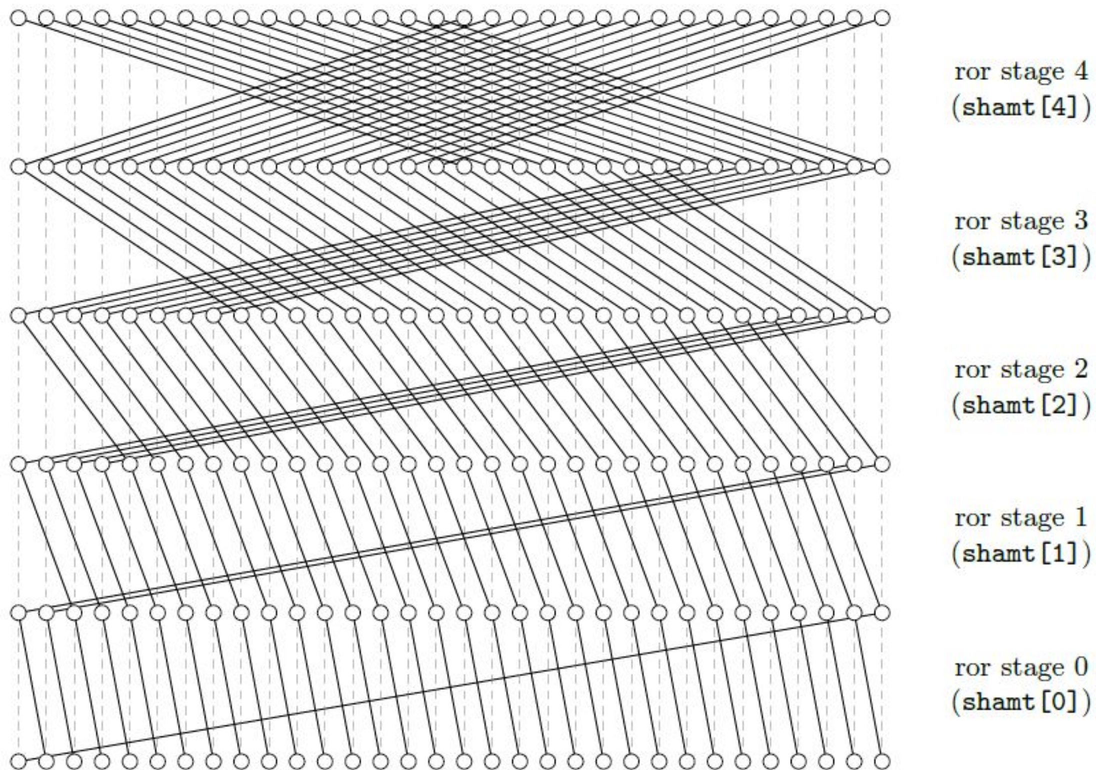
RV32, RV64:

- `ror rd, rs1, rs2`
- `rol rd, rs1, rs2`
- `rori rd, rs1, imm`

RV64 only:

- `rorw rd, rs1, rs2`
- `rolw rd, rs1, rs2`
- `roriw rd, rs1, imm`

# Rotate Right



# Generalized Shuffle (shfl, unshfl, shfli, unshfli, zip, unzip)

RV32, RV64:

- `shfl rd, rs1, rs2`
- `unshfl rd, rs1, rs2`
- `shfli rd, rs1, imm`
- `unshfli rd, rs1, imm`

RV64 only:

- `shflw rd, rs1, rs2`
- `unshflw rd, rs1, rs2`

The x86 PUNPCK[LH]* MMX/SSE/AVX instructions perform similar operations as

- `shfli rd, rs, -8`
- `shfli rd, rs, -16`

# Generalized Shuffle: index bit swapping

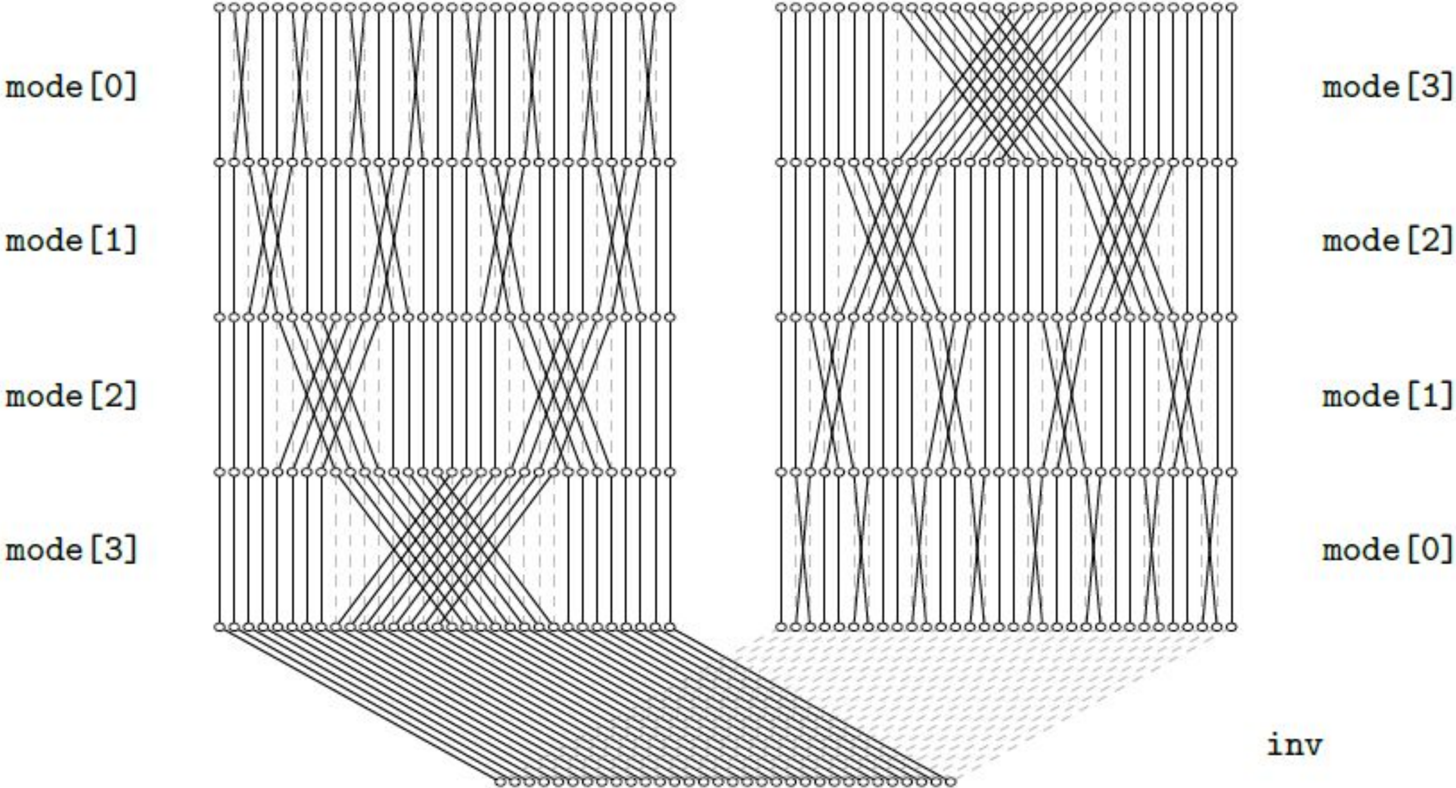
1. The generalized shuffle instruction has a *mode* operand with log2(XLEN)-1 bits
   a. This mode defines the permutation operation
   b. Thus only a subset of the permutations are available to a single shfl/unshfl
2. Each bit indicates that the corresponding index bit and its neighbor are to be swapped
3. Many shuffles are not their own inverse as each index-bit swap is performed on the output of the previous swap
   a. shfl is performed on the index bits from MSB to LSB
      i. mode[3]: swap i[4] ⇔ i[3]
      ii. mode[2]: swap i[3] ⇔ i[2]
      iii. mode[1]: swap i[2] ⇔ i[1]
      iv. mode[0]: swap i[1] ⇔ i[0]
   b. unshfl is performed on the index bits from LSB to MSB
      i. mode[0]: swap i[1] ⇔ i[0]
      ii. mode[1]: swap i[2] ⇔ i[1]
      iii. mode[2]: swap i[3] ⇔ i[2]
      iv. mode[3]: swap i[4] ⇔ i[3]

# Generalized shuffle: First two index-bit swaps

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input index | 11111 | 11110 | 11101 | 11100 | 11011 | 11010 | 11001 | 11000 | 10111 | 10110 | 10101 | 10100 | 10011 | 10010 | 10001 | 10000 | 01111 | 01110 | 01101 | 01100 | 01011 | 01010 | 01001 | 01000 | 00111 | 00110 | 00101 | 00100 | 00011 | 00010 | 00001 | 00000 | mode[3] |
| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | i[4]<=>i[3] |
| output index | 11111 | 11110 | 11101 | 11100 | 11011 | 11010 | 11001 | 11000 | 01111 | 01110 | 01101 | 01100 | 01011 | 01010 | 01001 | 01000 | 10111 | 10110 | 10101 | 10100 | 10011 | 10010 | 10001 | 10000 | 00111 | 00110 | 00101 | 00100 | 00011 | 00010 | 00001 | 00000 | |

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input index | 11111 | 11110 | 11101 | 11100 | 11011 | 11010 | 11001 | 11000 | 10111 | 10110 | 10101 | 10100 | 10011 | 10010 | 10001 | 10000 | 01111 | 01110 | 01101 | 01100 | 01011 | 01010 | 01001 | 01000 | 00111 | 00110 | 00101 | 00100 | 00011 | 00010 | 00001 | 00000 | mode[2] |
| | 31 | 30 | 29 | 28 | 23 | 22 | 21 | 20 | 27 | 26 | 25 | 24 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 7 | 6 | 5 | 4 | 11 | 10 | 9 | 8 | 3 | 2 | 1 | 0 | i[3]<=>i[2] |
| output index | 11111 | 11110 | 11101 | 11100 | 10111 | 10110 | 10101 | 10100 | 11011 | 11010 | 11001 | 11000 | 10011 | 10010 | 10001 | 10000 | 01111 | 01110 | 01101 | 01100 | 00111 | 00110 | 00101 | 00100 | 01011 | 01010 | 01001 | 01000 | 00011 | 00010 | 00001 | 00000 | |

# Generalized Shuffle Diagram



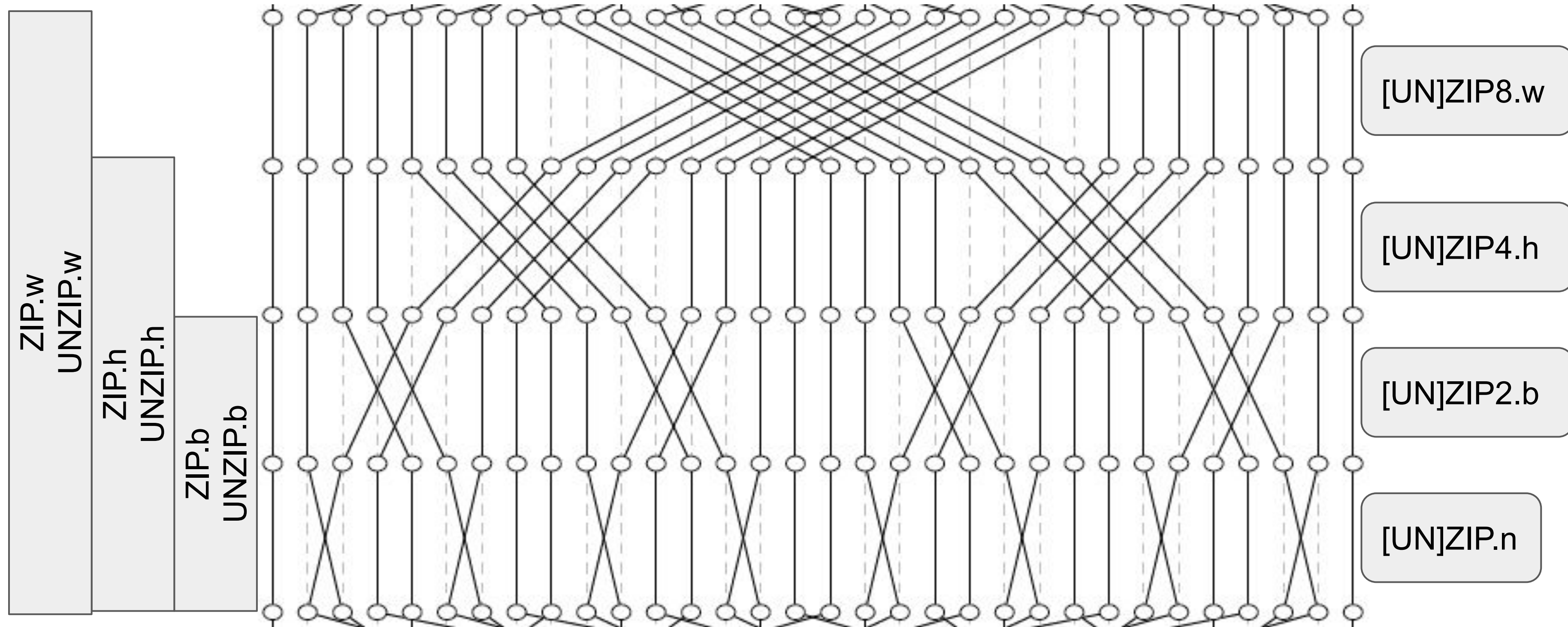
# Generalized Shuffle: index-bit permutations

All permutations are possible within log2(XLEN) shfl/unshfl instructions

1. There are log2(XLEN)! permutations
   a. For XLEN=32: log2(32) => 5; 5! => 120 permutations
   b. For XLEN=64: log2(64) => 6; 6! => 720 permutations
2. Each shfl/unshfl instruction supports (log2(XLEN)-1)! Permutations
   a. For XLEN=32: log2(32)-1 => 4; 4! => 24 permutations
   b. For XLEN=64: log2(64)-1 => 5; 5! => 120 permutations
3. log2(XLEN) shfl/unshfl operations to hit *all* permutations
   a. For XLEN=32: 5 * 24   => 120 permutations
   b. For XLEN=64: 6 * 120 => 720 permutations

Fortunately, all of the **[un]zip** variants can be done in a single shfl/unsfl

# Generalized Shuffle: [UN]ZIP pseudo instructions



# Shuffle Example: Swapping bits 0 and 1

| Instruction | State (XLEN=8) | Bit-Index Op |
|---|---|---|
| *initial value* | 7 6 5 4 3 2 1 0 | — |
| `rori a0, a0, 2` | 1 0 7 6 5 4 3 2 | $i' := i - 2$ |
| `unshfli a0, a0, -1` | 1 7 5 3 0 6 4 2 | $i' := \text{ror}(i)$ |
| `roli a0, a0, 1` | 7 5 3 0 6 4 2 1 | $i' := i + 1$ |
| `shfli a0, a0, -1` | 7 6 5 4 3 2 0 1 | $i' := \text{rol}(i)$ |

Table 2.2: Breakdown of the **`ror+[un]shfl`** sequence for swapping the two LSB bits of a word, using XLEN=8 for simplicity.

The mechanics of this sequence is closely related to the fact that `rol(ror(x-2)+1)` is a function that maps 1 to 0 and 0 to 1 and every other number to itself. (With rol and ror denoting 1-bit rotate left and right shifts respectively.)

# Bit permutation instructions: Generalized Reverse

RV32, RV64:

- `grev rd, rs1, rs2`
- `grevi rd, rs1, imm`

RV64 only:

- `grevw rd, rs1, rs2`
- `greviw rd, rs1, imm`

# Generalized Reverse: rev pseudo instructions

| | | Bit Index | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| shamt | pOp | **31** | **30** | **29** | **28** | **27** | **26** | **25** | **24** | **23** | **22** | **21** | **20** | **19** | **18** | **17** | **16** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| 00000 | noop | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 00001 | rev.p | 30 | 31 | 28 | 29 | 26 | 27 | 24 | 25 | 22 | 23 | 20 | 21 | 18 | 19 | 16 | 17 | 14 | 15 | 12 | 13 | 10 | 11 | 8 | 9 | 6 | 7 | 4 | 5 | 2 | 3 | 0 | 1 |
| 00010 | rev2.n | 29 | 28 | 31 | 30 | 25 | 24 | 27 | 26 | 21 | 20 | 23 | 22 | 17 | 16 | 19 | 18 | 13 | 12 | 15 | 14 | 9 | 8 | 11 | 10 | 5 | 4 | 7 | 6 | 1 | 0 | 3 | 2 |
| 00011 | rev.n | 28 | 29 | 30 | 31 | 24 | 25 | 26 | 27 | 20 | 21 | 22 | 23 | 16 | 17 | 18 | 19 | 12 | 13 | 14 | 15 | 8 | 9 | 10 | 11 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 |
| 00100 | rev4.b | 27 | 26 | 25 | 24 | 31 | 30 | 29 | 28 | 19 | 18 | 17 | 16 | 23 | 22 | 21 | 20 | 11 | 10 | 9 | 8 | 15 | 14 | 13 | 12 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 |
| 00101 | | 26 | 27 | 24 | 25 | 30 | 31 | 28 | 29 | 18 | 19 | 16 | 17 | 22 | 23 | 20 | 21 | 10 | 11 | 8 | 9 | 14 | 15 | 12 | 13 | 2 | 3 | 0 | 1 | 6 | 7 | 4 | 5 |
| 00110 | rev2.b | 25 | 24 | 27 | 26 | 29 | 28 | 31 | 30 | 17 | 16 | 19 | 18 | 21 | 20 | 23 | 22 | 9 | 8 | 11 | 10 | 13 | 12 | 15 | 14 | 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 |
| 00111 | rev.b | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 01000 | rev8.h | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| 01001 | | 22 | 23 | 20 | 21 | 18 | 19 | 16 | 17 | 30 | 31 | 28 | 29 | 26 | 27 | 24 | 25 | 6 | 7 | 4 | 5 | 2 | 3 | 0 | 1 | 14 | 15 | 12 | 13 | 10 | 11 | 8 | 9 |
| 01010 | | 21 | 20 | 23 | 22 | 17 | 16 | 19 | 18 | 29 | 28 | 31 | 30 | 25 | 24 | 27 | 26 | 5 | 4 | 7 | 6 | 1 | 0 | 3 | 2 | 13 | 12 | 15 | 14 | 9 | 8 | 11 | 10 |
| 01011 | | 20 | 21 | 22 | 23 | 16 | 17 | 18 | 19 | 28 | 29 | 30 | 31 | 24 | 25 | 26 | 27 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 12 | 13 | 14 | 15 | 8 | 9 | 10 | 11 |
| 01100 | rev4.h | 19 | 18 | 17 | 16 | 23 | 22 | 21 | 20 | 27 | 26 | 25 | 24 | 31 | 30 | 29 | 28 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 11 | 10 | 9 | 8 | 15 | 14 | 13 | 12 |
| 01101 | | 18 | 19 | 16 | 17 | 22 | 23 | 20 | 21 | 26 | 27 | 24 | 25 | 30 | 31 | 28 | 29 | 2 | 3 | 0 | 1 | 6 | 7 | 4 | 5 | 10 | 11 | 8 | 9 | 14 | 15 | 12 | 13 |
| 01110 | rev2.h | 17 | 16 | 19 | 18 | 21 | 20 | 23 | 22 | 25 | 24 | 27 | 26 | 29 | 28 | 31 | 30 | 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 | 9 | 8 | 11 | 10 | 13 | 12 | 15 | 14 |
| 01111 | rev.h | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 10000 | rev16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 10001 | | 14 | 15 | 12 | 13 | 10 | 11 | 8 | 9 | 6 | 7 | 4 | 5 | 2 | 3 | 0 | 1 | 30 | 31 | 28 | 29 | 26 | 27 | 24 | 25 | 22 | 23 | 20 | 21 | 18 | 19 | 16 | 17 |
| 10010 | | 13 | 12 | 15 | 14 | 9 | 8 | 11 | 10 | 5 | 4 | 7 | 6 | 1 | 0 | 3 | 2 | 29 | 28 | 31 | 30 | 25 | 24 | 27 | 26 | 21 | 20 | 23 | 22 | 17 | 16 | 19 | 18 |
| 10011 | | 12 | 13 | 14 | 15 | 8 | 9 | 10 | 11 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 28 | 29 | 30 | 31 | 24 | 25 | 26 | 27 | 20 | 21 | 22 | 23 | 16 | 17 | 18 | 19 |
| 10100 | | 11 | 10 | 9 | 8 | 15 | 14 | 13 | 12 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 27 | 26 | 25 | 24 | 31 | 30 | 29 | 28 | 19 | 18 | 17 | 16 | 23 | 22 | 21 | 20 |
| 10101 | | 10 | 11 | 8 | 9 | 14 | 15 | 12 | 13 | 2 | 3 | 0 | 1 | 6 | 7 | 4 | 5 | 26 | 27 | 24 | 25 | 30 | 31 | 28 | 29 | 18 | 19 | 16 | 17 | 22 | 23 | 20 | 21 |
| 10110 | | 9 | 8 | 11 | 10 | 13 | 12 | 15 | 14 | 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 | 25 | 24 | 27 | 26 | 29 | 28 | 31 | 30 | 17 | 16 | 19 | 18 | 21 | 20 | 23 | 22 |
| 10111 | | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 11000 | rev8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| 11001 | | 6 | 7 | 4 | 5 | 2 | 3 | 0 | 1 | 14 | 15 | 12 | 13 | 10 | 11 | 8 | 9 | 22 | 23 | 20 | 21 | 18 | 19 | 16 | 17 | 30 | 31 | 28 | 29 | 26 | 27 | 24 | 25 |
| 11010 | | 5 | 4 | 7 | 6 | 1 | 0 | 3 | 2 | 13 | 12 | 15 | 14 | 9 | 8 | 11 | 10 | 21 | 20 | 23 | 22 | 17 | 16 | 19 | 18 | 29 | 28 | 31 | 30 | 25 | 24 | 27 | 26 |
| 11011 | | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 12 | 13 | 14 | 15 | 8 | 9 | 10 | 11 | 20 | 21 | 22 | 23 | 16 | 17 | 18 | 19 | 28 | 29 | 30 | 31 | 24 | 25 | 26 | 27 |
| 11100 | rev4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 11 | 10 | 9 | 8 | 15 | 14 | 13 | 12 | 19 | 18 | 17 | 16 | 23 | 22 | 21 | 20 | 27 | 26 | 25 | 24 | 31 | 30 | 29 | 28 |
| 11101 | | 2 | 3 | 0 | 1 | 6 | 7 | 4 | 5 | 10 | 11 | 8 | 9 | 14 | 15 | 12 | 13 | 18 | 19 | 16 | 17 | 22 | 23 | 20 | 21 | 26 | 27 | 24 | 25 | 30 | 31 | 28 | 29 |
| 11110 | rev2 | 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 | 9 | 8 | 11 | 10 | 13 | 12 | 15 | 14 | 17 | 16 | 19 | 18 | 21 | 20 | 23 | 22 | 25 | 24 | 27 | 26 | 29 | 28 | 31 | 30 |
| 11111 | rev | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

grev[i] (Zbp)

| RISC-V | ARM | X86 |
|---|---|---|
| `rev` | RBIT | — |
| `rev8.h` | REV16 | — |
| `rev8.w` | REV32 | — |
| `rev8` | REV | BSWAP |

Table 2.5: Comparison of bit/byte reversal instructions

| Prefix | Mask | Suffix | Mask | |
|---|---|---|---|---|
| `rev` | 111111 | — | 111111 | |
| `rev2` | 111110 | `.w` | 011111 | (`w` = word) |
| `rev4` | 111100 | `.h` | 001111 | (`h` = half word) |
| `rev8` | 111000 | `.b` | 000111 | (`b` = byte) |
| `rev16` | 110000 | `.n` | 000011 | (`n` = nibble) |
| `rev32` | 100000 | `.p` | 000001 | (`p` = pair) |

Table 2.4: Naming scheme for `grevi` pseudo-instructions. The prefix and suffix masks are ANDed to compute the immediate argument.

# Comparing permutation networks: rotate and GREV

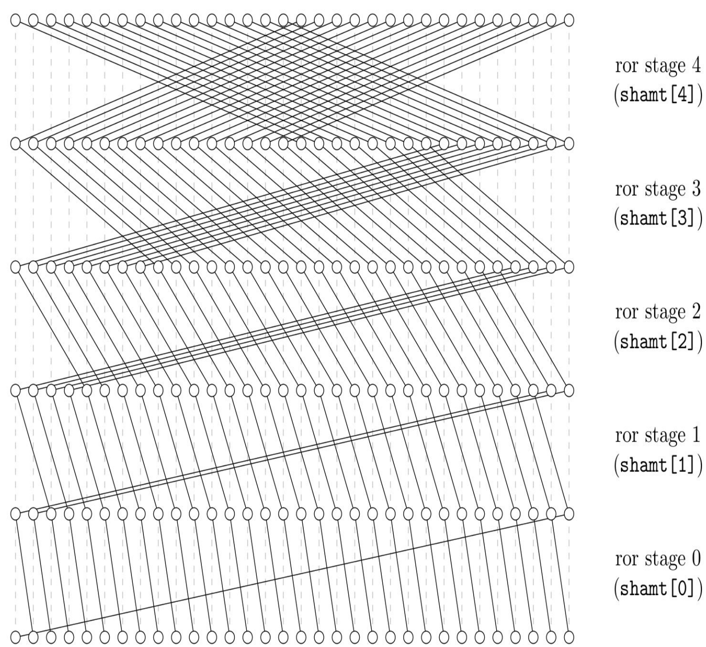



Figure 2.1: `ror` permutation network

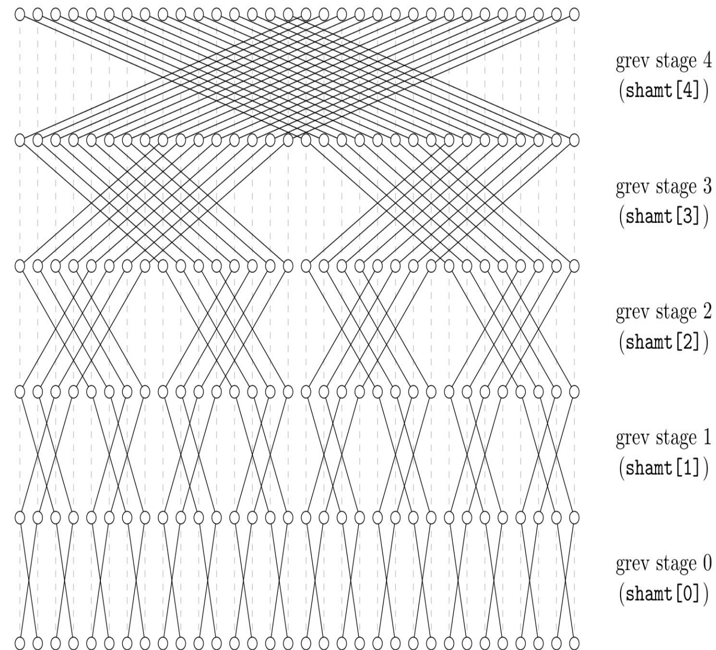



Figure 2.2: `grev` permutation network

# gorc[i] (Zbp)

Like grev but OR pairs of bits instead of swapping them.

This is most useful for checking if naturally aligned bitfields (such as the bytes in a word) are zero or non-zero. For example, counting trailing non-zero bytes in a0:

```
gorci a0, a0, 7      (= orc.b a0, a0)
not a0, a0
ctz a0, a0
srli a0, a0, 3
```

Or checking if a0 contains any zero bytes:

```
gorci a0, a0, 7      (= orc.b a0, a0)
addi a0, a0, 1
bnez a0, found_zero_byte
```

“Zbb” contains orc.b and orc16.

# bext, bdep (Zbe)

```
bext rd, rs1, rs2
```

extract bits marked with ones in rs2 from rs1 and compress to LSB end

```
bdep rd, rs1, rs2
```

expand bits from LSB end of rs1 to line up with the set bits in rs2

These instructions are equivalent to PEXT/PDEP in x86 BMI2.

However, they are not new inventions. We even find instructions like these in some 60s mainframe architectures.

# bext, bdep (Zbe)

For example, extracting a 28-bit value from the 7 LSB bits in each byte of a 32-bit word:

```
li t0, 0x7f7f7f7f
bext a0, a0, t0
```

The following code efficiently calculates the index of the 10th set bit in a0 using bdep:

```
sbseti t0, zero, 9
bdep a0, t0, a0
ctz a0, a0
```

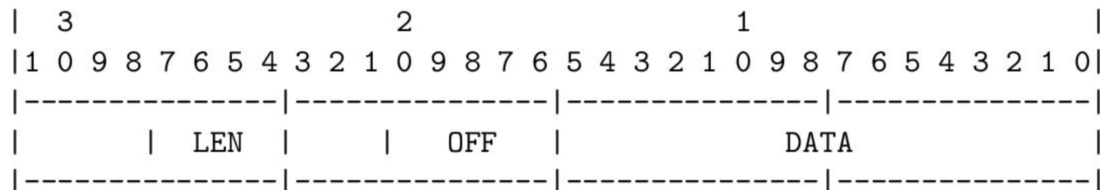
The following code computes the (1-indexed) indices of the set bits in the input byte, and stores those indices in the LSB nibbles of the output word, with the rest of the output word set to zero:

```
li t0, 0x87654321
andi a0, a0, 0xff
zip a0, a0
zip a0, a0
orc.n a0, a0
and a1, a0, t0
bext a0, a1, a0
```

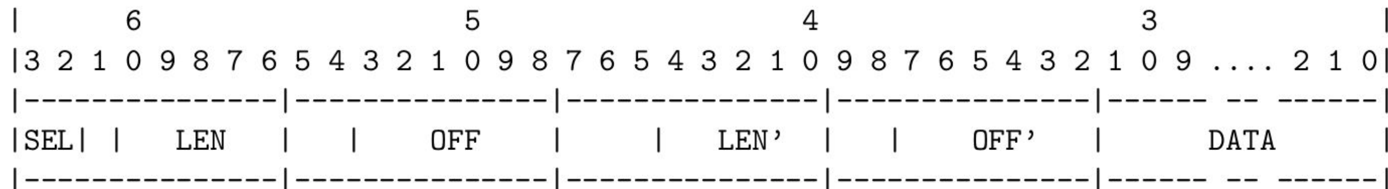
# bfp (Zbf)

Bit Field Place (BFP). Place up to XLEN/2 bits from rs2 in rs1. The upper half of rs2 contains position and length of the bit field.

The layout of the control word in rs2 is as follows for RV32. `LEN=0` encodes for `LEN=16`.




And on RV64 (with `LEN=0` encoding for `LEN=32`):




When `SEL=10` then `LEN` and `OFF` are used, otherwise `LEN'` and `OFF'` are used.

# bfp (Zbf)

Bit Field Place (bfp) is usually used as part of a fuse-able three instruction sequence.

Placing bits from `a0` in `a1`, with results in `t0` on RV32:

```
addi t0, zero, {length[3:0], offset[7:0]}
pack t0, a0, t0
bfp t0, a1, t028
```

And on RV64:

```
lui t0, zero, {3’b 100, length[4:0], offset[7:0], 4’b 0000}
pack t0, a0, t0
bfp t0, a1, t0
```

# bmat[x]or, bmatflip (Zbm, RV64)

These RV64-only instructions implement bit-matrix operations for 8x8 bit matrices.

Bit matrices are stored as 64-bit integers, in row-major order, LSB bit at (1,1).

`bmator` implements a matrix-matrix product, with AND for multiply and OR for addition. (Knuth calls this operation MOR.)

`bmatxor` implements a matrix-matrix product, with AND for multiply and XOR for addition. (Knuth calls this operation MXOR.)

`bmatflip` is an unary instruction that transposes such an 8x8 bit matrix.

Similar instructions can be found in x86 (GF2P8AFFINEQB) and Cray XMT.

# bmat[x]or, bmatflip (Zbm, RV64)

```
Using bmat[x]or to implement 64x64 bit matrix multiply:

Converting to/from block-matrix form:
    bm64c_baseisa:    1294 instructions
    bm64c_bitmanip:    507 instructions

Transposing a 64x64 matrix:
    bm64t_baseisa:   14613 instructions
    bm64t_baseisa2:   9857 instructions
    bm64t_bitmanip:   1224 instructions
    bm64t_bitmanip*:   194 instructions

Performing a 64x64 times 64x8 bit matrix multiply:
    bm64m_baseisa:    3099 instructions   (387 instr / dword)
    bm64m_bitmanip:    379 instructions   ( 47 instr / dword)
    bm64m_bitmanip*:   239 instructions   ( 29 instr / dword)
```

# clmul[hr] (Zbc)

`clmul` computes the “carry-less product” of the two arguments.

- Mechanically it is multiplication with XOR instead of ADD
- Mathematically it is multiplication of binary polynomials
  (binary polynomials = polynomial ring over GF(2))

`clmulr` computes `rev(clmul(rev(A), rev(B)))`.

`clmulh` computes the upper half of the product.
  Note that `clmulh(A, B) = clmulr(A, B) >> 1`.

Applications for carry-less product include CRCs, hashing, GCM, morton codes, and gray codes. Similar instructions are found in x86, SPARC, and TI C6000.

# crc32[c] (Zbr)

We define dedicated instructions for CRC32 and CRC32C.

Main advantages over use of CLMUL for CRC:

- slightly reduced area
- slightly improved performance
- significantly reduced power

Main disadvantage:

- only works with CRC32 and CRC32C polynomials

Smaller uC might want to implement Zbb+Zbr and larger cores might want to implement full “B” including CLMUL.

# crc32[c] (Zbr)

The CRC instructions are unary instructions. For example for CRC32C:

```
crc32c.b rd, rs
crc32c.h rd, rs      (equivalent to 2x crc32c.b)
crc32c.w rd, rs      (equivalent to 4x crc32c.b)
crc32c.d rd, rs      (equivalent to 8x crc32c.b, RV64-only)
```

Usage: XOR the new data with the LSB end of the CRC state, then execute the CRC instruction of the right length. For example, adding 32-bit from a1 to a CRC in a0:

```
xor a0, a0, a1
crc32c.w a0, a0
```

# CRC benchmark

Instruction counts for CRC32Q / CRC32C of a 1 kB block.

```
CRC32Q:
    crc32q_lookup   12293     … using lookup tables
    crc32q_barrett   2056     … using CLMUL barrett reduction
    crc32q_fold      1055     … using CLMUL folding method

CRC32C:
    crc32c_lookup   11270     … using lookup tables
    crc32c_barrett   2059     … using CLMUL barrett reduction
    crc32c_fold      1050     … using CLMUL folding method
    crc32c_instr      646     … using dedicated CRC instructions
```

# cmov, cmix (Zbt)

`cmov rd, rs2, rs1, rs3`

- return rs1 if rs2 is non-zero, otherwise return rs3
- thus this instruction implements `rd = rs2 ? rs1 : rs3`.

`cmix rd, rs2, rs1, rs3`

- use bits in rs2 to select bits in rs1 or rs3
- thus this instruction implements `rd = (rs2 & rs1) | (~rs2 & rs3).`

# fsl, fsr[i] (Zbt)

`fsl rd, rs1, rs3, rs2`

- rotate shift `{rs1, rs3}` left by `rs2` bits
- return the MSB half of the result

`fsr rd, rs1, rs3, rs2`

- rotate shift `{rs3, rs1}` right by `rs2` bits
- return the LSB half of the result

`fsri rd, rs1, rs3, imm`

- rotate shift `{rs3, rs1}` right by `imm` bits
- return the LSB half of the result

# macro-ops

For cores with support for macro-op fusion we explicitly recommend to fuse the following sequences:

- `slli+srli, slli+srai` extract bit-field
- `lui+addi, lui+addi+(pack|bfp)` load const, deposit bit-field
- `(addi|lui|packh)+pack+bfp` deposit bit-field

There are no dedicated instructions for these operations because they could not be encoded in regular 32-bit instruction formats in a reasonable way, and the community seems to overwhelmingly favor macro-ops over long instruction formats for these operations.

# Loading constants (RV64)

**sign-extended 32-bit:**

```
lui a0, imm
addiw a0, a0, imm
```

**zero-extended 32-bit:**

```
lui a0, imm
addiwu a0, a0, imm
```

or

```
addi a0, zero, imm
pack a0, a0, zero
```

**`0x WXYZ WXYZ WXYZ WXYZ`:**

```
lui a0, 0x0XYZW
orc16 a0, a0
```

**Any 64-bit value (spills t0):**

```
lui t0, imm
addiw t0, t0, imm

lui a0, imm
addiw a0, a0, imm
pack a0, a0, t0
```

# Packing Byte Vectors

**RV32:**

```
packh a0, a0, a1
packh a2, a2, a3
pack  a0, a0, a2
```

**RV64:**

```
packh a0, a0, a1
packh a2, a2, a3
packh a4, a4, a5
packh a6, a6, a7
packw a0, a0, a2
packw a4, a4, a6
pack  a0, a0, a4
```

# Packing bit-fields

There are different ways of packing bit-fields with the help of RISC-V BitManip instructions. A very efficient one is using `pack[hw]` to pack the data in one register, and then use `bext` to “compress” the result.

For example, packing a 16-bit RGB value in 5:6:5 format:

```
li    t0, 0x00f8fcf8

packh a0, a0, a1
pack  a0, a0, a2
bext  a0, a0, t0
```

# Packing bit-fields

Another way is using funnel shifts. Same 5:6:5 RGB example:

```
srli a2, a2, 3

slli a1, a1, XLEN-8
fsli a1, a2, a1, 6

slli a0, a0, XLEN-8
fsli a0, a1, a0, 6
```

# Byte Permutations

**Using rot, grev, and [un]shfl:**

- 24 ways of arranging the four bytes in a 32-bit word
  - requires at most 3 instructions (see table on the right)
- 40320 ways of arranging the eight bytes in a 64-bit word
  - requires at most 9 instructions (doesn't fit slide :)

**Using bmator:**

- bmator with a permutation matrix P can be used to
  - permute the 8 bytes in a0: bmator a0, P, a0
  - permute the 8 bits in each byte: bmator a0, a0, P

| Bytes | Instructions |
|---|---|
| A B C D | *initial byte order* |
| A B D C | ROR(24),SHFL(8),ROR(8) |
| A C B D | SHFL(8) |
| A C D B | ROR(8),GREV(8),SHFL(8) |
| A D B C | ROR(16),SHFL(8),ROR(24) |
| A D C B | ROR(8),GREV(8) |
| B A C D | ROR(8),SHFL(8),ROR(24) |
| B A D C | GREV(8) |
| B C A D | ROR(16),SHFL(8),ROR(8) |
| B C D A | ROR(24) |
| B D A C | GREV(8),SHFL(8) |
| B D C A | ROR(24),SHFL(8) |
| C A B D | ROR(8),GREV(24),SHFL(8) |
| C A D B | ROR(16),SHFL(8) |
| C B A D | ROR(8),GREV(24) |
| C B D A | SHFL(8),ROR(24) |
| C D A B | ROR(16) |
| C D B A | ROR(8),SHFL(8),ROR(8) |
| D A B C | ROR(8) |
| D A C B | SHFL(8),ROR(8) |
| D B A C | ROR(8),SHFL(8) |
| D B C A | GREV(24),SHFL(8) |
| D C A B | ROR(24),SHFL(8),ROR(24) |
| D C B A | GREV(24) |

# Arbitrary 64-bit Bit Permutations

Using 64x64 bit matrix multiply:

- 47 instructions per word (when processing groups of 8 words)
- (29 instructions when data is in block-matrix form)

Using Beneš network (aka “butterfly-reverse-butterfly networks”):

- 22 instructions, + 11 LD instructions for loading masks

Using Sheep-and-goats (SAG) operations:

- 36 instructions (= 6 SAG x 6 ops/SAG), + 6 LD for loading masks
- “Hacker’s Delight” SAG implementation is 340 ops for a single SAG

# Bitboards

Bitboards are 64-bit bitmasks that are used to represent part of the game state in chess engines (and other board game AIs). The bits in the bitmask correspond to squares on a $8 \times 8$ chess board:

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Many bitboard operations are simple straight-forward operations such as bitwise-AND, but mirroring and rotating bitboards can take up to 20 instructions on x86.

# Bitboards

```
Flip horizontal:
 63 62 61 60 59 58 57 56
 55 54 53 52 51 50 49 48
 47 46 45 44 43 42 41 40
 39 38 37 36 35 34 33 32
 31 30 29 28 27 26 25 24
 23 22 21 20 19 18 17 16
 15 14 13 12 11 10  9  8
  7  6  5  4  3  2  1  0
```

```
RISC-V Bitmanip:
   rev.b

x86:
   13 operations
```

```
Flip vertical:
  0  1  2  3  4  5  6  7
  8  9 10 11 12 13 14 15
 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31
 32 33 34 35 36 37 38 39
 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55
 56 57 58 59 60 61 62 63
```

```
RISC-V Bitmanip:
   rev8

x86:
   bswap
```

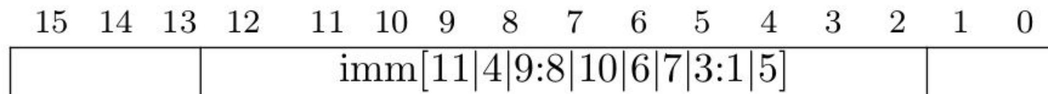
```
Rotate 180:
  7  6  5  4  3  2  1  0
 15 14 13 12 11 10  9  8
 23 22 21 20 19 18 17 16
 31 30 29 28 27 26 25 24
 39 38 37 36 35 34 33 32
 47 46 45 44 43 42 41 40
 55 54 53 52 51 50 49 48
 63 62 61 60 59 58 57 56
```

```
RISC-V Bitmanip:
   rev

x86:
   14 operations
```

```
Transpose:
  7 15 23 31 39 47 55 63
  6 14 22 30 38 46 54 62
  5 13 21 29 37 45 53 61
  4 12 20 28 36 44 52 60
  3 11 19 27 35 43 51 59
  2 10 18 26 34 42 50 58
  1  9 17 25 33 41 49 57
  0  8 16 24 32 40 48 56
```

```
RISC-V Bitmanip:
   zip, zip, zip
   (or just bmatflip)

x86:
   18 operations
```

# Decode RVC




```
// RV32IB
decode_cj:
li t0, 0x28800001
li t1, 0x000016b8
li t2, 0xb4e00000
li t3, 0x4b000000
bext a1, a0, t1
bdep a1, a1, t2
rori a0, a0, 11
bext a0, a0, t0
bdep a0, a0, t3
c.or a0, a1
c.srai a0, 20
ret
```

```
// Zbp only
decode_cj:
shfli a0, a0, 15
rori a0, a0, 28
shfli a0, a0, 2
shfli a0, a0, 14
rori a0, a0, 26
shfli a0, a0, 8
rori a0, a0, 10
unshfli a0, a0, 12
rori a0, a0, 18
unshfli a0, a0, 14
rori a0, a0, 28
shfli a0, a0, 6
rori a0, a0, 28
unshfli a0, a0, 15
slli a0, a0, 21
srai a0, a0, 20
ret
```

```
// RV32I
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
```

# Decode LEB128

```
uint64_t leb128_decode_bitmanip(uint64_t in)
{
        uint64_t mask = _rv64_orc8(0x80);
        long t = (_rv64_ctz(~_rv64_bext(in, mask)) << 3) + 7;
        return _rv64_bext(in & _rv64_slo(0, t), ~mask);
}
```

Litte Endian Base-128 (LEB128)

- Variable-length code for integers
- MSB bit in each byte = continue flag
- Used in
    - DWARF file format
    - WebAssembly portable binary
    - Dalvik Executable Format

```
uint64_t leb128_decode_reference(uint64_t in)
{
        uint64_t out = in & 127;
        long shamt = 7;

        while (in & 128) {
                in >>= 8;
                out |= (in & 127) << shamt;
                shamt += 7;
        }

        return out;
}
```

The functions on this slide don’t return the size of the encoded integer. But that’s merely an exercise in returning two values.

# Encode LEB128

```
uint64_t leb128_encode_bitmanip(uint64_t in)
{
        uint64_t mask = _rv64_orc8(0x80);
        uint64_t out = _rv64_bdep(in, ~mask);
        uint64_t t = (((uint64_t)!out)-1) >> (_rv64_clz(out) & 63);
        return out | (mask & t);
}
```

```
uint64_t leb128_encode_reference(uint64_t in)
{
        uint64_t out = in & 127;
        long shamt = 7;
        in >>= 7;

        while (in) {
                out |= (uint64_t)1 << shamt;
                out |= (uint64_t)(in & 127) << (shamt+1);
                shamt += 8;
                in >>= 7;
        }

        return out;
}
```

The functions on this slide don’t return the size of the encoded integer. But that’s merely an exercise in returning two values.

# Decoding UTF-8

```
// branch-free UTF-8 decoder using bitmanip instructions and misaligned load
long utf8_decode_bitmanip(const uint8_t *in, uint32_t *out, long len)
{
        uint32_t *p = out;
        uint32_t mask = 0x3f3f3f3f;

        for (int i = 0; i < len;) {
                uint32_t v = *(uint32_t*)(in+i);
                int bytes = _rv32_max(1, _rv32_clz(~(v << 24)));
                if (__builtin_expect(bytes > 4, 0)) error();
                v = _rv32_rev8(v) << bytes;
                v = v >> ((bytes-8*bytes) & 31);
                v = _rv32_bext(v, mask | (bytes-2));
                *(p++) = v;
                i += bytes;
        }

        return p - out;
}
```

20 Instructions / Unicode codepoint

# Getting started

- GitHub repository
  - `git clone https://github.com/riscv/riscv-bitmanip`

    ⇒ `riscv-bitmanip/bitmanip-draft.pdf`

- Building the toolchain
  - `sudo mkdir /opt/riscv64b; sudo chown $USER: /opt/riscv64b`
  - `cd tools; bash build-all.sh; bash riscv-gcc-demo.sh`
  - (this only builds the C toolchain, not the C++ toolchain)

- Running basic tests
  - `cd tests; bash run.sh`

- Running examples
  - `cd examples/crc; make`

- Running Verilog module sim
  - `cd verilog/rvb_bextbdep; make`

- Running Verilog system sim
  - `cd verilog/picorv32; make`
  - (requires riscv-compliance)

# Source for tool chains

- GCC GitHub repositories
  - `github.com/embecosm/riscv-gcc` or
  - `github.com/riscv/riscv-gcc`
  - in both cases use the `riscv-bitmanip` branch
- Binutils/GDB repository
  - `github.com/embecosm/riscv-binutils-gdb`
  - `riscv-bitmanip` branch for hand-coded assembler
  - `mblinov-bmi-work` branch for CGEN assembler/disassembler and GDB simulator
- Clang/LLVM monorepo
  - `github.com/embecosm/llvm-project`
  - as with GCC, use the `riscv-bitmanip` branch

# Compliance tests

- See `riscv-bitmanip/compliance/` for compliance test generator
- Integration into `riscv-compliance` is ongoing (see "`bitmanip`" branch)
- PicoRV32 demo passes RV32B compliance tests
- Spike passes RV32B/RV64B compliance tests

# Thanks to our task group members!

Thanks to the 98 individual members of the task group for their valuable contributions. We also like to thank the following companies for their support in varying capacity.

B?

**Zbb?** (base)
- clz, ctz, pcnt
- min[u], max[u]
- sext.h, sext.b
- add[i]wu, subwu
- addu.w, subu.w
- slliu.w

**Zbs** (single bit)
- sbset[i]
- sbclr[i]
- sbinv[i]
- sbext[i]

**Zba** (addr calc)
- shNadd
- shNaddu.w

**Zbb? / Zbp** (shared)
- andn, orn
- xnor, pack[hu]
- rol, ror[i]
- rev8, rev
- orc.b, orc16

**Zbp** (permutation)
- slo[i]
- sro[i]
- grev[i]
- gorc[i]
- [un]shfl[i]

**Zbe**
- bext, bdep

**Zbf**
- bfp

**Zbc**
- clmul[hr]

**Zbr**
- crc32[c]

**Zbm** (matrix)
- bmat[x]or
- bmatflip

**Zbt** (ternary)
- cmov, cmix
- fsl, fsr[i]

# Thanks!

# Questions?