RISC-V ISA Profile of the XMSS C Implementation

Wrenna Robson*

February 2026

1 Context

XMSS (RFC 8391) is a stateful hash-based signature scheme standardised for post-quantum use. We are building a formally verifiable implementation in Jasmin, targeting x86-64 first (where the Jasmin compiler is mature) and RISC-V second (where the backend is under active development).

As a reference for the Jasmin port, we have written a C99 implementation of XMSS and XMSS-MT from scratch, following RFC 8391 including Errata 7900. This implementation is designed to be structurally portable to Jasmin: no VLAs, no heap allocation, no function pointers, no recursion, and all loop bounds derivable from parameter constants. It is this C implementation—not the upstream reference—that we analyse here.

Before writing RISC-V Jasmin code—or contributing to the Jasmin RISC-V backend—we need to know which ISA extensions XMSS actually exercises. This report answers that question by disassembling the compiled C implementation and classifying every instruction.

The key question is: *does XMSS require anything beyond RV64I + M?* If the algorithm logic is pure I+M, then ISA extension work (Zbb rotates for SHA-2, etc.) is confined to the hash layer and does not affect the algorithm-level Jasmin code.

2 Methodology

2.1 Analysis target

We analyse `libxmss.a`—the static library containing only XMSS algorithm code: parameter derivation, hash wrappers, WOTS+, L-tree, treehash, BDS state management, XMSS signing/verification, and XMSS-MT. This comprises 13 object files.

Crucially, the library excludes `printf`, `malloc`, stack guards, and other libc/test-harness code that would pollute the profile.

2.2 Toolchain

Compiler	`riscv64-linux-gnu-gcc` 13.3.0 (Ubuntu)
Flags	`-march=rv64gc -mabi=lp64d -O3`
Disassembler	`riscv64-linux-gnu-objdump` (Binutils 2.42)

*With Claude (Anthropic), which co-authored the analysis tooling and report text.

The `rv64gc` march is the standard general-purpose profile: RV64I + M + A + F + D + Zicsr + Zifencei + C. It does *not* include Zba, Zbb, or other bitmanip extensions.

2.3 Classification

Each disassembled instruction is classified on two orthogonal axes:

1. **Semantic extension** (what the instruction does): I, M, A, F, D, Zba, Zbb, etc. Determined by looking up the mnemonic in a table generated from the `riscv-opcodes` database (the same database used by the RISC-V toolchain).
2. **Encoding width** (how it is encoded): 16-bit compressed (C extension) or 32-bit standard. Determined from the raw instruction byte count in the objdump output.

The semantic extension is what matters for Jasmin. C encoding is a secondary observation: the assembler handles it automatically and it does not affect which instructions Jasmin source code must express.

A subtlety: GNU objdump renders compressed instructions using their uncompressed aliases (`sd` not `c.sd`). A naïve classifier that looks for `c.` prefixes would report C = 0%. We detect C encoding from byte width instead.

3 Findings

3.1 Extension summary

Table 1 shows the complete picture: XMSS uses only the I and M extensions.

Table 1: Semantic extension summary across all 13 object files.

Extension	Instructions	Unique mnemonics	Notes
I	9505	46	Base integer: loads, stores, branches, arithmetic, shifts
M	57	3	`mulw` (37), `mul` (17), `divuw` (3)
A, F, D	0	0	Not present despite being enabled by `rv64gc`
Zba, Zbb, Zbs	0	0	Not present (not in `rv64gc`; see §4)

99.4% of instructions are base integer (I). The 0.6% that are M come entirely from compiler-generated address arithmetic: `mulw` for array index calculations, `mul` for 64-bit offset computation, and `divuw` for parameter derivation in `params.c`.

3.2 Per-module breakdown

Table 2 shows instruction counts per object file, grouped into algorithm modules and hash modules.

Two observations:

- The hash layer (SHA-2, SHAKE, hash dispatch) uses *zero* M instructions. It is pure RV64I.
- M instructions appear only in the algorithm layer, and only for index/offset arithmetic the compiler generates from C expressions like `idx * n` or `h / d`.

Table 2: Per-object-file instruction counts. M column shows M-extension instruction count; all remaining instructions are I.

Object file	Total	M insns	C-encoded (%)
Hash layer			
sha2_local.c.o	1953	0	33%
shake_local.c.o	1337	0	44%
xmss_hash.c.o	1082	0	51%
Algorithm layer			
bds.c.o	1130	5	55%
xmss_mt.c.o	1145	18	58%
wots.c.o	760	4	47%
xmss.c.o	637	3	60%
treehash.c.o	484	2	50%
bds_serialize.c.o	473	14	53%
params.c.o	286	6	60%
ltree.c.o	129	5	69%
address.c.o	99	0	38%
utils.c.o	47	0	68%
Total	**9562**	**57**	**48%**

3.3 Compressed encoding

48% of instructions use 16-bit C encoding. This varies from 33% (`sha2_local.c.o`, which has many 32-bit shift-immediate instructions for SHA-2 rotations) to 69% (`ltree.c.o`, which is mostly register moves and branches).

C encoding is handled automatically by the assembler and is invisible to Jasmin source code. It reduces code size but does not affect correctness or the set of required ISA extensions.

4 The Zbb question

With `-march=rv64gc`, the hash modules implement SHA-256, SHA-512, and Keccak using only RV64I operations. SHA-2 in particular requires 32-bit rotations, which the compiler synthesises as shift–shift–or sequences:

```
srliw  a5, a5, 17    # high part
slliw  a4, a4, 15    # low part
or     a5, a5, a4    # combine
```

The Zbb extension provides a single `rorw` instruction that replaces this 3-instruction sequence. Similarly, `rev8` (Zbb) replaces multi-instruction byte-swap sequences for SHA-2 endianness conversion. Since `rv64gc` does not include Zbb, the compiler cannot emit these instructions.

4.1 Empirical verification: `rv64gc_zbb`

Recompiling `libxmss.a` with `-march=rv64gc_zbb` confirms the prediction. Table 3 shows the extension breakdown.

Table 3: Extension comparison: `rv64gc` vs `rv64gc_zbb`.

Extension	`rv64gc`	`rv64gc_zbb`	Delta
I	9505	9281	−224
M	57	57	0
Zbb	0	164	+164
Total	9562	9502	−60

The total instruction count drops by 60: Zbb replaces multi-instruction sequences with single instructions, reducing both instruction count and code size.

Table 4 breaks down the 164 Zbb instructions.

Table 4: Zbb instructions emitted with `-march=rv64gc_zbb`.

Count	Mnemonic	Object files	Replaces
41	`rolw`	sha2_local	`srliw+slliw+or`
28	`andn`	sha2_local, shake_local, xmss_mt	`not+and`
27	`rori`	sha2_local	`srli+slli+or`
19	`rev8`	sha2_local	multi-insn byte-swap
17	`roriw`	sha2_local	`srliw+slliw+or`
17	`rol`	sha2_local, shake_local	`srl+sll+or`
12	`maxu`	bds, treehash, xmss_hash	branch-based max
3	`minu`	bds, shake_local	branch-based min

4.2 Where Zbb lands

Of the 164 Zbb instructions, **142 (87%)** are in the hash layer (`sha2_local.c.o` and `shake_local.c.o`). The rotations (`rolw`, `rori`, `roriw`, `rol`) and byte-reversal (`rev8`) appear exclusively in SHA-256/SHA-512 compression and Keccak.

The remaining **22 (13%)** are in the algorithm layer: `maxu`/`minu` for BDS height comparisons and `andn` for bitmask operations in XMSS-MT. These are minor opportunistic optimisations — the algorithm does not structurally depend on Zbb.

This confirms the architecture decision: the hash layer is the only component where ISA-specific optimisation matters. All algorithm-layer Jasmin code can be written in pure RV64I+M.

5 Recommended Jasmin targets

`rv64im` (minimum)
: Base integer + multiply/divide. Sufficient for all XMSS algorithm logic. The hash layer would use software rotations.

`rv64gc` (standard)
: The standard Linux general-purpose profile. Adds A, F, D, Zicsr, Zifencei, and C—none of which XMSS uses, but targeting it ensures binary compatibility with standard RISC-V Linux distributions.

rv64gc_zbb (optimised)

Adds Zbb to the standard profile. The hash layer can exploit `rorw`/`ror` for SHA-2 rotations and `rev8` for endianness conversion. Algorithm-layer code is unchanged.

6 Next steps

1. The Jasmin implementation starts with x86-64, where the compiler backend is mature and the hash layer can use native rotation instructions (`ROR`).
2. Algorithm-layer Jasmin code (WOTS+, BDS, treehash, XMSS sign/verify) should use only portable basic operations—no architecture-specific intrinsics. The ISA analysis confirms this is sufficient.
3. When the Jasmin RISC-V backend matures, the algorithm layer ports directly. The hash layer is the only component requiring architecture-specific work (Zbb rotations).