RISC-V Optimization Guide

Table of Contents

About this Document

The intention is to give specific actionable optimization recommendations for software developers writing code for RISC-V application processors.

Intended Audience

This document may be of use to any software developers who need to operate at an instruction set level. For example, this would include:

- Toolchain or managed runtime developers
- Engineers working on architecture-specific software such as operating systems
- Contributors to performance-critical libraries writing SIMD or vectorized implementations
- Educators creating assembly language examples

Implementation-neutral vs implementation-specific

This guide is intended to be vendor-neutral and implementation-neutral. It is meant to present optimizations that are applicable to many different implementations of RISC-V application processors, using standard RISC-V instructions.

However, to reach peak performance, this document will also highlight some implementation-specific optimizations. These will be clearly outlined like such:

IMPLEMENTATION SPECIFIC	This content is implementation specific

The use of non-RISC-V-standard ISA extensions is out of scope of this guide, and shall not be documented here.

Detecting RISC-V Extensions on Linux

Version 6.4 of the Linux kernel introduces a new syscall, riscv_hwprobe that can be used to determine interesting pieces of information about the underlying RISC-V CPUs on which a user space program runs. In particular, it exposes the RISC-V extensions supported by the CPUs in addition to other performance information such as whether and how efficiently the CPUs support unaligned memory accesses.

Where available riscv_hwprobe is preferable to using HWCAP as:

1. It's extensible (there's enough bits for all future extensions).
2. It can be used to query for extensions with multi-letter names, e.g., Zba, Zbb, and Zbs
3. The hwprobe extension bits are versioned and their meaning is concretely defined.
4. It can be used to determine additional information about the CPU. One potentially interesting example is its ability to return information about how well the CPU handles unaligned accesses.

An example usage of riscv_hwprobe is given below.

```c
#include <asm/hwprobe.h>
#include <asm/unistd.h>
#include <errno.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    struct riscv_hwprobe requests[] = {{RISCV_HWPROBE_KEY_MVENDORID},
                       {RISCV_HWPROBE_KEY_MARCHID},
                       {RISCV_HWPROBE_KEY_MIMPID},
                       {RISCV_HWPROBE_KEY_CPUPERF_0},
                       {RISCV_HWPROBE_KEY_IMA_EXT_0}};

    int ret = syscall(__NR_riscv_hwprobe, &requests,
              sizeof(requests) / sizeof(struct riscv_hwprobe), 0,
              NULL, 0);
    if (ret) {
        fprintf(stderr, "Syscall failed with %d: %s\n", ret,
            strerror(errno));
        return 1;
    }

    bool has_misaligned_fast =
        (requests[3].value & RISCV_HWPROBE_MISALIGNED_FAST) ==
        RISCV_HWPROBE_MISALIGNED_FAST;
    printf("Vendor ID: %llx\n", requests[0].value);
    printf("MARCH ID: %llx\n", requests[1].value);
    printf("MIMPL ID: %llx\n", requests[2].value);
    printf("HasMisalignedFast: %s\n", has_misaligned_fast ? "yes" : "no");
    __u64 extensions = requests[4].value;
    printf("Extensions:\n");
    if (extensions & RISCV_HWPROBE_IMA_FD)
        printf("\tFD\n");
    if (extensions & RISCV_HWPROBE_IMA_C)
        printf("\tC\n");
    if (extensions & RISCV_HWPROBE_IMA_V)
        printf("\tV\n");
    if (extensions & RISCV_HWPROBE_EXT_ZBA)
        printf("\tZBA\n");
    if (extensions & RISCV_HWPROBE_EXT_ZBB)
        printf("\tZBB\n");
    if (extensions & RISCV_HWPROBE_EXT_ZBS)
        printf("\tZBS\n");

    return 0;
```

```
}
```

The program uses the riscv_hwprobe syscall to output information about the cores of the current system. Example output generated by the program might look like this

```
Vendor ID: 0
MARCH ID: 0
MIMPL ID: 0
HasMisalignedFast: no
Extensions:
    FD
    C
    V
    ZBA
    ZBB
    ZBS
```

This program requires a distribution with a Linux 6.5 kernel or greater to build. It invokes the syscall directly rather than using a glibc wrapper as glibc does not yet support riscv_hwprobe. The example can be built on a distribution with an older kernel by copying the required constants from the Linux kernel sources. The `__NR_riscv_hwprobe` constant is defined to be 258.

If the program is executed on Linux 6.3 or earlier, the riscv_hwprobe syscall will return ENOSYS and the program will exit with an error. However, callers of riscv_hwprobe may wish to handle this specific error and deduce from it that neither V, Zba, Zbb nor Zbs are supported. The riscv_hwprobe syscall predates Linux support for these extensions (riscv_hwprobe was added in 6.4 whereas support for the aforementioned extensions was added in 6.5), so if the syscall is not implemented these extensions are not supported by the kernel.

Multi-versioning

In some cases it can be desirable to create an executable with multiple instances of the same algorithm compiled with different target options. A version of the algorithm compiled with base options, e.g., RV64G, might be supplied so that the binary can be run on a wide range of devices, while a more performant version compiled with some additional extensions, e.g., Zba or Vector, could also be provided. With current toolchains, to support multiple versions of code compiled for different architectural features, separate compilation of translation units in addition to a manual runtime check for extensions (using riscv_hwprobe for example) must be used. In the future, features to support multi-versioning within a single translation unit are planned.

GCC and Clang both support a function attribute called target_clones that can be used to compile multiple versions of a given function with different compiler flags. An ifunc resolver function is automatically created that ensures that the function symbol is resolved to the most suitable version of the function at load time. At the time of writing, the compilers have yet to add support for the target_clones attribute for RISC-V.

Optimizing Scalar Integer

Materializing Constants

Prefer idiomatic LUI/ADDI sequence for 32 bit constants

Signed 12 bit constants can be materialized with a single ADDI instruction. Signed 32 bit constants can be materialized with a pair of LUI and ADDI instructions depending on the constant.

For example, the constant 1024, which fits in 12 bits can be materialized using a single instruction.

```
addi t0, zero, 1024
```

The constant 4097 (0x1001) requires two instructions to load.

```
lui  a0, 1
addiw a0, a0, 1
```

The LUI instruction stores the top 20 bits of the constant (1 << 12 = 4096) into a0 and the ADDIW instruction supplies the bottom 12 bits, adding 1 and yielding the expected result.

There is one complication with this instruction sequence that occurs when the 12th bit of the constant to be materialized is set. This is an issue as the immediate argument of the ADDIW instruction is a 12 bit signed number and so cannot encode a 12 bit unsigned number. In this case the immediate value encoded in the ADDIW instruction is formed by subtracting 4096 from the bottom 12 bits of the constant and by adding 1 to the constant passed to the LUI instruction (which adds 4096). For example, to encode the constant 0x1ffffff, the following sequence can be used.

```
lui a0, (0x1fff + 1)          ; lui a0, 0x2000
addiw a0, a0, (0xfff - 4096)  ; addiw a0, a0, -1
```

GAS provides the assembler modifiers %hi and %lo to simplify the process of loading 32 bit constants. Using these modifiers we can simply write

```
lui a0, %hi(0x1ffffff)
addiw a0, a0, %lo(0x1ffffff)
```

which is more readable. Assemblers also typically provide a pseudo instruction called LI to allow the constant to be loaded in a single assembly language statement. LI hides all the complexities of loading constants from the programmer, and is supported in both LLVM and GNU toolchains. For 32 bit constants, it will generate either one or two RISC-V instructions depending on the size of the constant, e.g.,

```
li a0, 0x1ffffff
```

generates the LUI/ADDIW pair shown above.

Make effective use of the x0 register

To set an integer register to zero use

```
mv x10, x0
```

or

```
li x10, 0
```

Do not use other idioms from other architectures to zero registers, e.g.,

```
xor x10, x10, x10
and x10, x10, x0
andi x10, x10, 0
sub x10, x10, x10
```

Zero can be folded into any instruction with a register operand. There's no need to initialize a temporary register with 0 for the sole purpose of using that register in a subsequent instruction. The following table identifies cases where a temporary register can be eliminated by prudent use of x0.

Do	Don't
`fmv.d.x f0,x0`	`li x5,0` `fmv.d.x f0,x5`
`amoswap.w.aqrl a0,x0,(x10)`	`li x5,0` `amoswap.w.aqrl x6,x5,(x10)`
`sb x0,0(x5)`	`li x6,0` `sb x6,0(x5)`
`bltu x0,x7,1f`	`li x5,0` `bltu x5,x7,1f`

Fold immediates into consuming instructions where possible

Many instructions support signed 12-bit (scalar) or 5-bit (vector) immediates. In particular, scalar loads and stores support reg+imm12 addressing and this should be used aggressively. For example, to load the second element of an array of 64 bit integers whose base pointer is stored in a0, write

```
ld t0, 8(a0)
```

rather than

```
addi a1, a0, 8
ld t0, (a1)
```

Use a constant pool for general 64 bit constants

There are many idioms for specific sub-classes of 64 bit constants (check what your C compiler does!), but in general, using a constant pool and a load is better than using the full 6 or 8 (if no temporary registers are to be used) instruction sequences required to materialize a 64 bit value.

Consider the following code which materializes a 64 bit constant

```
li a0, 0x123456789abcde1
```

This is materialized by binutils 2.40 into an 8 instruction sequence consuming 32 bytes when compiled with `-march=rv64g` or 26 bytes when compiled with `-march=rv64gc`

```
lui a0,0x92
addiw a0,a0,-1493
slli a0,a0,0xc
addi a0,a0,965
slli a0,a0,0xd
addi a0,a0,-1347
slli a0,a0,0xc
addi a0,a0,-543
```

If we use a constant pool the constant can be loaded in 16 bytes, 8 bytes for the constant and 8 for the instructions needed to load it.

```
1:
  auipc a0, %pcrel_hi(large_constant)
  ld a0, %pcrel_lo(1b)(a0)
...

.section .rodata
.p2align 3
```

```
large_constant:
  .dword 0x123456789abcde1
```

Use Canonical Move Idioms

Use the assembler MV mnemonic, which translates to ADDI rd, rs1, 0, to copy values from one register to another. For example use,

```
mv x10, x11
```

in preference to any of the following instructions.

```
or x10, x11, x0
ori x10, x11, 0
xor x10, x11, x0
xori x10, x11, 0
```

Avoid branches using conditional moves

The Zicond extension adds the two conditional operations czero.eqz and czero.nez. Where available, these operations can reduce pressure on the branch predictor, at the cost of a longer critical path. We believe this to generally be a good tradeoff, but examine particular hot instances carefully. Experience with other architectures shows this is frequently a complicated tradeoff.

As an example, assume the following code segment relies on an unpredictable branch to determine which constant to load into a0. If the original value of a0 is non-zero we set a0 to constant1 otherwise it is set to constant2, i.e., (a0 = a0 ? constant1 : constant2;).

```
  beqz a0, 1f
  li a0, constant1
  j 2f
1:
  li a0, constant2
2:
```

The branch can be eliminated using the Zicond instructions CZERO.EQZ and CZERO.NEZ followed by an OR.

```
li t2, constant1
li t3, (constant2 - constant1)
czero.nez t3, t3, a0
add a0, t3, t2
```

After the czero.nez instruction has executed, t3 will contain (constant2 - constant1) if a0 is zero or

zero if a0 is non-zero. Adding t2, which contains constant1, to t3 yields either constant2 if a0 is zero, or constant 1 if it is not. An additional optimization is possible if constant1 fits in 12 bits. In this case the initial li t2, constant1 instruction can be eliminated and the final add instruction can be replaced by an addi, e.g.,

```
li t3, (constant2 - constant1)
czero.nez t3, t3, a0
addi a0, t3, constant1
```

The above code sequence can also be written using only the base integer ISA albeit with the additional cost of three extra instructions.

```
li t2, constant1
li t3, constant2
seqz t0, a0
addi t0, t0, -1
xor t1, t2, t3
and t1, t1, t0
xor a0, t1, t3
```

The combination of the instructions

```
seqz t0, a0
addi t0, t0, -1
```

results in t0 having no bits set if a0 is zero, or all the bits set if a0 is non-zero. The resulting value in t0 is then a mask which can be ANDed with the constants in its original or inverted form to generate the operands for the OR instruction.

When the 'M' extension is available a shorter sequence using the MUL instruction is possible

```
li t2, constant1
li t3, constant2
xor t1, t2, t3
seqz t0, a0
mul t1, t1, t0
xor a0, t1, t2
```

In the following sequence of instructions

```
seqz t0, a0
mul t1, t1, t0
```

SEQZ sets t0 to either 1 or 0 depending on whether a0 contains 0. The MUL instructions multiplies t0, which is either 1 or 0, by the result of the earlier XOR instruction, storing either 0 or the xor of

the two constants in t1.

When constant1 and constant2 are 0 and 1 respectively, the above code sequence can be written using a single instruction from the base integer ISA.

```
seqz a0, a0
```

Padding

Use canonical NOPs, NOP (ADDI X0, X0, 0) and C.NOP (C.ADDI X0, 0), to add padding within a function. Use the canonical illegal instruction (either 2 or 4 bytes of zeros depending on whether the C extension is supported) to add padding between functions.

Align char array to greater alignment

For a cpu that doesn't support fast misaligned memory accesses, increasing alignment might enable wider load/store usage for memory copy. For example, align to 4 for rv32 and align to 8 for rv64 might enable using LW/SW or LD/SD. For a cpu that supports fast misaligned memory accesses, wider load/store would be more profitable regardless of the data alignment.

Use shifts to clear leading/trailing bits

Use shifts if the immediate can't fit in signed 12-bit of ANDI.

For example

```
slli  x6, x5, 20
srli x7, x6, 20
```

rather than

```
lui x6, 1
addi x7, x6, -1
and x8, x7, x5
```

The above shift example is for a 32-bit system. The shift amount for a 64-bit target will be 52 in this case.

Optimizing Scalar Floating Point

Controlling Rounding Behavior

Prefer instructions with static round modes where possible. For many common scalar floating point

operations, rounding can be controlled by the instruction opcode, and this is strictly better than using FRM. For example, it's better to write

```
fadd.s f10, f10, f11, rtz
```

than

```
csrrwi t0, frm, 1      ; 1 = rtz
fadd.s f10, f10, f11
fsrm t0
```

Use immediates for FRM writes (as shown above) and avoid redundant FRM writes where possible. Save/restore idioms for FRM may be expensive. Try to save/restore over the largest region feasible.

NOTE See the corresponding vector section.

Optimizing Vector

As this document targets RISC-V application processors, the recommendations in this section assume, at a minimum, the presence of the "V" Vector extension for application processors as defined in the 'RISC-V "V" Vector Extension' specification version 1.0. The "V" Vector extension depends on Zvl128b. Consequently, this document assumes a VLEN of >= 128 and does not consider optimizations for smaller VLENs.

Avoid Using v0 for Non-mask Operations

The v0 register defined by the RISC-V vector extension is special in that it can be used both as a general purpose vector register and also as a mask register. As a preference, use registers other than v0 for non-mask values. Otherwise data will have to be moved out of v0 when a mask is required in an operation. v0 may be used when all other registers are in use, and using v0 would avoid spilling register state to memory.

Tradeoffs Between Vector Length Agnostic and Specific

Vector length agnostic ("VLA") code is designed to run on any implementation of the vector specification. Vector length specific ("VLS") code will run on exactly one VLEN. There's also a middle ground where code may run on a family of VLEN values, but not all possible VLENs.

For benchmarking purposes, using vector length specific code is strongly recommended.

One common technique to write fully generic and performant code is to dispatch to multiple implementations based on the dynamic value of the vlenb CSR, at the expense of additional code size.

Controlling VL and VTYPE

Use only immediate VTYPE encodings, vsetvli and vsetivli. The vsetvl instruction should be reserved for context-restoring type operations.

Predicating Efficiently

Avoid undisturbed modes

Undisturbed modes for masks and tails require the hardware to read the destination register before writing back only the active elements. Avoiding tail undisturbed and mask undisturbed states may be desirable. If the tail elements of a vector are undefined, use the tail agnostic state. Only use tail undisturbed when the tail elements contain content which must be preserved in place. In many cases, making a copy of the source and reading from that copy later may perform better than merging data into the same architectural register via tail undisturbed.

Avoid masking where possible

If one only cares about the leading elements, using VL to restrict computation to those leading elements is likely more performant than using masking to restrict computation to those leading elements. Forming a mask to disable a lane which is never read is purely additional overhead without benefit. If an operation must be predicated - because it can fault or might have side effects - prefer VL predication over masking, where possible.

Use v[f]merge instructions rather than unpredictable data-based branches.

Round up VL to the data path width

When performing operations without side effects (e.g., no loads or stores) on vectors of different sizes, issue one call to vsetvli setting VL to the size of the vector data path width, rather than issuing separate vsetvli calls for each vector length. For example, suppose that the data path width is 256 bits and we need to process vectors of 2, 4 and 8 64-bit integers. We'd set the SEW to 64 bits and VL to 4. We'd then process the 2 element vector, the 4 element vector and then the two halves of the 8 element vector. Two elements of the register holding the 2 element vector would be processed unnecessarily but this should not impact performance and their values can be ignored.

Choosing EMUL

Choose the value used for LMUL/EMUL carefully.

- If all operations involved are linear in LMUL choose EMUL to balance register pressure and frontend bandwidth. Depending on the processor implementation, this is going to mean either very high LMUL or very low LMUL, but not both. For example, on Out-of-Order machines, it's generally better to have very low LMUL, resulting in a lower register pressure at the expense of more instructions being issued. While on In-Order machines, it's generally better to have very high LMUL, favoring lower instruction count at the expense of higher register pressure.
- For vrgather.vv, prefer smaller LMUL. Cost is likely proportional to LMUL^2. For portable code,

prefer low LMUL for vrgather.vv. Cost of vrgather.vi will depend on microarchitecture, but will likely be either linear or quadratic in LMUL. Note that vector length specific (but not vector length agnostic) code can split a single high LMUL shuffle into a number of LMUL1 shuffles. Depending on the shuffle mask, doing so may be strongly profitable.

- For normal memory operations, select the largest LMUL known not to exceed the AVL. For example, prefer LMUL1 if VL may be less than VLEN/SEW.
- For indexed and strided memory operations, pick based on the surrounding code.
- Be aware that EMUL may not equal LMUL for all operands of loads, stores, narrowing and widening instructions. For example, limiting EMUL to 2 means that LMUL should be limited to 1 for widening and narrowing instructions.

Preferred Idioms

Fold immediate scalar values into integer vector operations where possible using the .vi instruction variant.

Prefer .vi, .vx and .vf variants over .vv variants when vector register pressure is high.

For example, to add 1.0 to each element of an array of 32 bit floats whose length is stored in a1, we might write

```
  li t0, 1
  fcvt.s.w fa0, t0
1:
  vsetvli t0, a1, e32, m1, ta, ma
  vle32.v v8, (a0)
  vfadd.vf v10, v8, fa0
  vse32.v v10, (a0)
  sh2add a0, t0, a0
  sub a1, a1, t0
  bnez a1, 1b
```

Use vmv.v.x or vmv.v.i to splat a value across all body lanes. For example, to broadcast 3 across all elements of the register group starting at v8, use

```
vmv.v.i v8, 3
```

Use v(f)merge.vxm to perform a masked splat. The following code splats alternating values of 0xaaaaaaaa and 0xbbbbbbbb into v2.

```
vsetvli t0, x0, e32, m1, ta, ma
li t0, 0xaaaaaaaa
li t1, 0xbbbbbbbb
vmv.v.x v0, t0
vmerge.vxm v2, v0, t1, v0
```

If only the first lane is active, use vmv.s.x. Prefer vmv.v.i vd, 0 to zero a vector register. Using these two pieces of advice together in an example, we can set the first element of a vector register to 2 and the remaining elements to 0 as follows.

```
vsetvli t0, x0, e32, m1, tu, ma
vmv.v.i v8, 0
li t1, 2
vmv.s.x v8, t1
```

Use vmv1r.v (and variants) to perform whole register copies, ignoring predication. Use vmv.v.v to perform register moves while respecting inactive lanes.

Controlling Rounding Behavior

See the corresponding scalar section. The discussion of redundant FRM writes applies for vector operations as well. Unfortunately, most vector operations do not have static round mode overrides, so vector code is more sensitive to FRM pressure.

Memory Operations

The RISC-V vector specification provides a wide range of different load and store operations. These operations may not all share the same performance characteristics. Where possible, employ these instructions in the following order of descending preference.

1. Whole register
2. Unit-stride
3. Unit-stride segment
4. Strided
5. Indexed
6. Strided segment
7. Indexed segment

Whole register loads and stores

When moving data of exactly VLEN size (or a size which can be rounded up to VLEN safely), prefer the use of the whole vector register load and store instructions. For example, an array of bytes whose size is a multiple of 64kb could be copied as follows, where a0 and a1 hold the destination and source addresses respectively, and a2 holds the number of bytes to copy.

```
  beqz a2, 1f
  csrr t0, vlenb
  slli t0, t0, 3
2:
  vl8r.v v8, (a1)
```

```
  vs8r.v v8, (a0)
  add a0, a0, t0
  add a1, a1, t0
  sub a2, a2, t0
  bnez a2, 2b
1:
  ret
```

Unit-stride segment loads and stores

Unit-stride segment loads and stores can be used to vectorize loops that process Arrays of Structures (AOS).

For example, consider an array of structures that each contain three 8 bit RGB values. The elements in this AOS are laid out in memory as follows:

```
r1,g1,b1,r2,g2,b2,r3,g3,b3,...
```

The sample code below demonstrates how unit-stride segment loads can be used to unpack the RGB elements into vectors of R, G and B values and further process them to compute a grayscale representation. A single 8 bit grayscale value is computed from each RGB element and is written into an output buffer using a unit-stride store instruction.

```
rgb_to_grayscale:
  vsetvli t0, x0, e8, m2, ta, ma
  li t0, 77
  li t1, -106
  li t2, 29
  vmv.v.x v2, t0
  vmv.v.x v4, t1
  vmv.v.x v6, t2
1:
  vsetvli t0, a2, e8, m2, ta, ma
  vlseg3e8.v v12, (a1)
  vwmulu.vv  v8, v12, v2
  vwmaccu.vv v8, v14, v4
  vwmaccu.vv v8, v16, v6
  vnsrl.wi v8, v8, 8
  vse8.v v8, 0(a0)
  sh1add t1, t0, t0
  sub a2, a2, t0
  add a1, a1, t1
  add a0, a0, t0
  bne a2, x0, 1b
  ret
```

Note that the register pressure is not high in above example so we prefer the .vv rather than the

`.vi`/`.vx` variants of the instructions. Transferring data between the general purpose and vector registers may incur an overhead so we avoid doing so in the loop. If register pressure is high, the `.vi`/`.vx` variants are preferred as stated in Preferred Idioms.