Optimization of multiplication in the RISC-V architecture using bitmanip extension

Alexander Lekomtsev
Software Engineering Chair
St. Petersburg State University
St. Petersburg, Russia
a.lekomtsev@spbu.ru

Kirill Smirnov
Information Analytical Systems Chair
St. Petersburg State University
St. Petersburg, Russia
k.k.smirnov@spbu.ru

***Abstract*—RISC-V is an open architecture that is gaining popularity in academia as well as in industry. The basic instruction set itself is very limited, and additional functionality is implemented in extensions, some of them are included in the standard. The most common bit manipulation instructions are organized in a dedicated bitmanip extension. Bit operations are actively used to optimize arithmetic operations, in particular for multiplication by a constant. For example, on the x86 architecture, one can use the lea and logical shift instead of expensive mul instruction. In this paper, the authors investigate all possible optimizations for the multiplication operation using shXadd instructions from the bitmanip extension.**

***Index Terms*—RISC-V, Bitmanip, Multiplication, Clang, Compilers, Compiler Optimization**

I. Introduction

RISC-V is an open architecture that is actively gaining popularity both in academia and in industry [1]. RISC-V ISA has a modular design: the base instruction set is extremely small, and almost all general-purpose instructions are organized in extensions, both standard and non-standard. This modularity makes the architecture highly extensible, hardware manufacturers need to support only the base set, and the implementation of additional instructions is optional.

A dedicated extension is designed for the most common bit manipulations. The bitmanip extension is a set of bit and byte manipulation instructions aimed at improving performance, reducing assembly code size and power consumption. Since this extension is relatively new [2], compilers do not always use these instructions efficiently while generating an assembly code. For example, a recent study shows that both gcc and clang do not recognize code patterns for most bitmanip instructions [3].

It is well known that a shift and add instructions are faster than an integer multiplication instruction for almost all conventional microarchitectures. For example, core i7 spends 1 cycle for an integer addition and 3 cycles for an integer multiplication. Sometimes the multiplication by a constant can be replaced by a faster sequence of shifts and additions. Back in the IBM/360 and x86 architectures, the LEA (load effective address) instruction was designed to generate a memory address while accessing an array member. Later, it became clear that this instruction could replace multiplication in some cases. For example, multiplication by 5 on x86 architecture can be written as follows: lea eax, [eax*4 + eax]. Modern compilers know this pattern well and apply this transformation for faster code. RISC-V provides similar instructions that combine shift and addition, namely, sh1add, sh2add, sh3add. In this paper, we refer to any of them as shNadd. On some microarchitectures, a sequence of three shNadd is faster than multiplication. Moreover, if a multiplication instruction is split into several shNadd, a compiler can reuse their results in a series of number-crunching operation.

Inverse transformation — folding a sequence of shifts to a single mul instruction — is also used by compilers to generate code optimized for size.

In this paper, we study all possible transformations from multiplication by constant to a series of shXadd for RISC-V ISA.

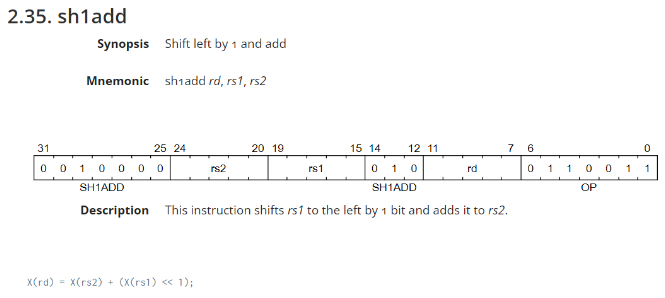
II. Background

Strictly speaking, bitmanip is not a single extension, but a collection of extensions, all related to bit manipulation [4]. Since there are lots of various manipulations, they are split into several groups: for address generation (zba), basic bit manipulation (zbb), carry-less multiplication (zbc) and single-bit instructions (zbs). In this paper, we consider sh1add, sh2add, sh3add instructions only. The semantics of the instruction “shNadd A B C” is as follows: calculate “(B << N) + C” and write the result into register A (Fig. 1). These instructions are most useful for address calculations when accessing arrays of 16-bit, 32-bit, or 64-bit elements. Also, the shNadd instruction can be used to replace multiplication in some cases. For example, to multiply a0 by 243, the straightforward way is: “li a5, 243; mulw a0, a5, a0”. Please note that RISC-V lacks “multiply by immediate” instruction, so the constant must be placed into a register beforehand. It is possible to rewrite it as:

sh1add a_0, a_0, a_0
sh3add a_0, a_0, a_0
sh3add a_0, a_0, a_0

The first instruction can be rewritten to $a_0 = (a_0 << 1) + a_0$; which is equivalent to multiplying a_0 by 3. The second and third instructions are all the same as $a_0 = (a_0 << 3) + a_0$; which is multiplication by 9. The first instruction multiplies by

3, the second and third by 9, and the entire sequence multiplies by 243.



2.35. sh1add

Synopsis Shift left by 1 and add

Mnemonic sh1add *rd*, *rs1*, *rs2*

Description This instruction shifts *rs1* to the left by 1 bit and adds it to *rs2*.

```
X(rd) = X(rs2) + (X(rs1) << 1);
```

Fig. 1. sh1add instruction specification.

III. Related work

Bit manipulation is a technique that aims to optimize operations. For example, using bit manipulations one can optimally check whether a number is a power of two, find the nearest power of two, count the number of leading zeros. In most processors, bitwise operations are performed in one clock cycle, which is often faster than multiplication or division.

One frequent application of bit manipulation is multiplication and division using bit shifts. With a left or right bit shift, you can multiply or divide by a power of two. For example, the x86 architecture has a "shl" instruction for this, ARM has a "slsl" instruction, and RISC-V has a "slli" instruction.

Operations combining bit shifts and addition are also used to optimize multiplication. Ways to use this kind of optimization vary from architecture to architecture. Multiplication by 5 can be written with different instructions on different architectures, but with the same idea: $x * 5 = (x << 2) + x$. In x86 this is done with lea (lea eax, [rdi + 4*rdi]), ARM uses an addition instruction along with a bit shift (add r0, r0, r0, r0, lsl #2), and RISC-V has its own combining instructions (sh2add a0, a0, a0).

The idea of optimization using shNadd instructions has already been used in compilers. For example, the Clang 18.1.0 compiler can generate a sequence of two instructions while processing multiplication by some constants (11, 13, 19, 21, 25, 27, 37, 41, 45, 81), and GCC 13.2.0 could generate sequences of three and four instructions:

$$\begin{array}{l} \text{sh2add } a_5,\ a_0,\ a_0 \\ \text{sh2add } a_5,\ a_5,\ a_5 \\ \text{sh2add } a_5,\ a_5,\ a_0 \\ \text{sh1add } a_0,\ a_5,\ a_0 \end{array}$$

to multiply value in register a_0 by 203 [5].

IV. Method

The main task is to find all values, multiplication by which can be rewritten as a sequence of three shNadd instructions. The proposed solution is quite straightforward: let's enumerate all possible combinations of three shNadd instructions, calculate the result, and if the result is a multiplication by a constant, collect the constant and the sequence of shNadd. To enumerate all combinations efficiently, the following notation and pattern were introduced:

- a_0 — the receiver register, initially it will hold the value we want to multiply by something, the result of multiplication will be added to it;
- a_1 — auxiliary register for storing intermediate calculations;
- $a_?$ — a register that can hold either a_0 or a_1;
- shXadd, shYadd, shZadd — any instruction of the set [sh1add, sh2add, sh3add].

All possible non-degenerate sequences of three instructions can be written using this pattern:

$$\begin{array}{llll} \text{shXadd} & ? & a_0 & a_0 \\ \text{shYadd} & ? & ? & ? \\ \text{shZadd} & a_0 & ? & ? \end{array}$$

We can safely ignore the case of using the register a_2 because it can occur for the first time only in the second line, and a_1 must have been used already (otherwise nothing is stored in a_1 and you can use it instead of a_2). Also, the third instruction must have both a_1 and a_2 registers (otherwise we could skip the line where the value is written to this register), but a_0 is not used, so the value written to a_2 can be written to it.

For example, the sequence

$$\begin{array}{l} \text{sh3add } a_1\ a_0\ a_0 \\ \text{sh3add } a_2\ a_1\ a_0 \\ \text{sh3add } a_0\ a_1\ a_2 \end{array}$$

can be replaced by

$$\begin{array}{l} \text{sh3add } a_1\ a_0\ a_0 \\ \text{sh3add } a_0\ a_1\ a_0 \\ \text{sh3add } a_0\ a_1\ a_0 \end{array}$$

The results are the same, but the latter uses only two registers.

Moreover, you can get a total of 1728 values (27 value variants (X,Y,Z) and 64 case variants in place of $a_?$), but not all of them are meaningful:

$$\begin{array}{l} \text{shXadd } a_1,\ a_0,\ a_0 \\ \text{shYadd } a_0,\ a_0,\ a_0 \\ \text{shZadd } a_0,\ a_1,\ a_1 \end{array}$$

The expression above can be replaced with:

$$\begin{array}{l} \text{shXadd } a_0,\ a_0,\ a_0 \\ \text{shZadd } a_0,\ a_0,\ a_0 \end{array}$$

The enumeration of all values was realized step by step: first, all possible substitutions a_0 and a_1 are enumerated in place of $a_?$ (partial application of the pattern), then X, Y, Z are substituted.

After the first step of the enumeration, we can obtain an expression with three variables (X, Y, Z) that describe the possible values of the partially-applied pattern. For example, the following sequence of instructions:

$$\text{shXadd } a_1\ a_0\ a_0$$
$$\text{shYadd } a_1\ a_0\ a_1$$
$$\text{shZadd } a_0\ a_0\ a_1$$

corresponds to the expression $t*(2^Z)+t*((2^Y)+(2^X+1))$, where t is the initial value of a_0. Next, values from one to three are substituted in place of Z, Y, X and the possible values for multiplication are obtained.

The last thing to check is an unsigned integer overflow case. If a multiplication instruction overflows, the sequence of shNadd must overflow exactly the same way. Single shift instruction and single add instruction handle this case correctly, the shNadd instruction is also correct. Thus, the sequence of shNadd instructions overflows correctly.

V. Implementation

LLVM uses a dedicated DSL (Domain-Specific Language) and a utility, both called TableGen, to describe platform-dependent optimizations [6]. The code is first written in a declarative style, then processed by the utility. TableGen allows you to describe target platforms and templates to code-generate instructions for a given architecture. Bitmanip Zba extension instructions are described in a separate file; multiplication with two shNadd instructions was described earlier in this file.

With TableGen, you can specify the operation to be optimized and the instructions to generate. For example, the optimization of multiplication by 11 can be written as follows:

```
def : Pat<(mul_const_oneuse GPR:$r,
(XLenVT 11)), (SH1ADD (SH2ADD GPR:$r,
GPR:$r), GPR:$r)>;
```

LLVM optimizer uses this TableGen code to generate an assembler code like this:

$$\text{sh2add } a_1,\ a_0,\ a_0$$
$$\text{sh1add } a_0,\ a_1,\ a_0$$

In this paper, we need to perform the inverse transformation: we have a sequence of shNadd instructions and our goal is to generate a TableGen expression. To solve this problem, we decided to employ a C preprocessor. For each piece of assembler code, we generate a series of C macros and pass them to cpp. For example, for the following assembler code:

$$\text{shXadd } a_1\ a_0\ a_0$$
$$\text{shYadd } a_0\ a_1\ a_0$$
$$\text{shZadd } a_0\ a_1\ a_0$$

we generate C code presented in Listing 1.

C preprocessor performs all necessary substitutions for us, and the result is a valid TableGen expression:

```
 (SHZADD (SHYADD GPR:$r, GPR:$r), (SHXADD
(SHYADD GPR:$r, GPR:$r), GPR:$r))
```

VI. Experiment

Comparisons of multiplication execution times before adding optimization and after were performed on a VisionFive 2 single-board computer with the following specifications:

- 1GHz RISC-V SiFive U74 processor;
- 4GB RAM.

The executable was compiled with the flags **-static -march=rv64 gc_zba -O2**. Google Benchmark version **1.8.3-7736df03049c** was used to measure code execution time [8].

The benchmark code is presented in Listing 2. To minimize branching instructions interference with our number-crunching, our instructions are repeated 128 times in each cycle. `explicit_mul` function measures execution time of pure multiplication, and `fast_mul` measures bitmanip instructions [9].

The test results are presented in the Table I. We can conclude that the difference between three shNadd instructions and multiplication instruction is statistically insignificant on the current board.

TABLE I
Comparision of explicit multiplication and bitmanip for single multiplication

Benchmark	Time	CPU	Iterations
explicit_mul	2057$\pm$5ns	2053$\pm$5ns	340798
fast_mul	2064$\pm$5ns	2057$\pm$5ns	340416

Another case in which optimization can be used is re-using intermediate results in consecutive multiplications. An example is presented in Listing 3. After each multiplication is transformed into a series of shNadd instructions, compiler can deduce that some of them are the same and optimize them away.

After adding optimization in Clang and compiling this code with the flags **--target=riscv64 -march=rv64gc_zba -S -O1**, we get 20 instructions of the shNadd type instead of the expected 30 (10 multiplications, each of which is represented by three instructions). The results of some calculations are re-used, thus reducing their number.

Two versions of this code were compared, before and after optimization [10]. The benchmark is presented in Listing 4, the results are presented in Table II. To avoid interference with memory load/store operations, they are commented out.

TABLE II
Comparision of explicit multiplication and bitmanip for series of different multiplications

Benchmark	Time	CPU	Iterations
explicit_mul	5129 $\pm$ 2 ns	5127 $\pm$ 2 ns	136512
shx	3591 $\pm$ 2 ns	3590 $\pm$ 2 ns	194974

VII. Conclusion

We enumerated every possible combination of three shNadd instructions and obtained a list of constants that are folded into these combinations. The entire mapping is implemented in our publicly available fork of LLVM project [7].

The benchmark results show that although optimization does not speed up the code at single multiplication, there is a noticeable improvement for consecutive multiplications.

References

[1] RISC-V International home page, URL: https://riscv.org/about/ (accessed: 07.04.2024).

[2] Bitmanip Extension in Public Review, URL: https://lists.riscv.org/g/tech-announce/message/66 (accessed: 07.04.2024).

[3] Lekomtsev A. A., Sukrahev A. D, Tool for evaluating the quality of instruction coverage by compilers for RISC-V architecture. Information Technologies and Systems 2023, BSUIR, pp. 103-104

[4] Latest bitmanip specification, URL: https://github.com/riscv/riscv-bitmanip/releases (accessed: 07.04.2024).

[5] Example of godbolt using a sequence of 4 bitmanip instructions, URL: https://godbolt.org/z/fqzccMhh6 (accessed: 07.04.2024).

[6] LLVM TableGen manual, URL: https://llvm.org/docs/TableGen/ProgRef.html (accessed: 07.04.2024).

[7] GitHub repository, URL: https://github.com/vacmannnn/llvm-project/tree/RISCVMulOptimization (accessed: 07.04.2024).

[8] Google Benchmark, URL: https://github.com/google/benchmark (accessed: 07.04.2024).

[9] Benchmarks listing, URL: https://t.ly/Fir86 (accessed: 07.04.2024)

[10] Benchmarks listing, URL: https://t.ly/wbLek (accessed: 07.04.2024)

```
#define ex GPR:$r
#define gamma (SHYADD ex, ex)
#define beta (SHXADD gamma, ex)
#define alpha (SHZADD gamma, beta)
alpha
```

Listing 1. Code for preprocessor to generate TableGen code

```
#define X2(X)   X       X
#define X4(X)   X2(X)   X2(X)
#define X8(X)   X4(X)   X4(X)
#define X16(X)  X8(X)   X8(X)
#define X32(X)  X16(X)  X16(X)
#define X64(X)  X32(X)  X32(X)
#define X128(X) X64(X)  X64(X)

#define XZ(X) { 128(X) }

static void explicit_mul(benchmark::State& state) {
    int64_t a = 2;
    for (auto _ : state)
        XZ(asm volatile ("li␣%1,39\nmul␣%1,%1,%0" :
                                    "=r"(a) : "r"(a));)
}

static void fast_mul(benchmark::State& state) {
    int a = 2;
    int b;
    for (auto _ : state) {
        XZ(asm volatile ("sh1add␣t0,%0,%0\n"
                         "sh2add␣%0,t0,t0\n"
                         "sh3add␣%1,t0,%0\n" : "=r" (b) :
                                    "r" (a) : "t0");)
    }
}
```

Listing 2. Testing the performance of two multiplication methods

```
int* mul(int *res, int *n) {
    int x = *n;
    res[0] = x * 23;
    res[1] = x * 29;
    res[2] = x * 35;
    res[3] = x * 39;
    res[4] = x * 43;
    res[5] = x * 47;
    res[6] = x * 49;
    res[7] = x * 51;
    res[8] = x * 53;
    res[9] = x * 55;
    return res;
}

int main() {
    int test = 10;
    int arr[10] = {};
    mul(arr, &test);
}
```

Listing 3. Sequence of multiplications

```
#include <benchmark/benchmark.h>

#define xM1 \
asm volatile("li␣␣␣␣␣␣a1,␣10" ::: "a1");\
asm volatile("li␣␣␣␣␣␣a2,␣23" ::: "a2");\
asm volatile("mul␣␣␣␣␣a2,␣a1,␣a2" ::: "a1", "a2");\
asm volatile("#sw␣␣␣␣␣␣a2,␣0(a0)");\
asm volatile("li␣␣␣␣␣␣a2,␣29" ::: "a2");\
asm volatile("mul␣␣␣␣␣a2,␣a1,␣a2" ::: "a2");\
asm volatile("#sw␣␣␣␣␣␣a2,␣4(a0)");\
asm volatile("li␣␣␣␣␣␣a2,␣35" ::: "a2");\
asm volatile("mul␣␣␣␣␣a2,␣a1,␣a2" ::: "a2");\
asm volatile("#sw␣␣␣␣␣␣a2,␣8(a0)");\
asm volatile("li␣␣␣␣␣␣a2,␣39" ::: "a2");\
asm volatile("mul␣␣␣␣␣a2,␣a1,␣a2" ::: "a2");\
asm volatile("#sw␣␣␣␣␣␣a2,␣12(a0)");\
asm volatile("li␣␣␣␣␣␣a2,␣43" ::: "a2");\
asm volatile("mul␣␣␣␣␣a2,␣a1,␣a2" ::: "a2");\
asm volatile("#sw␣␣␣␣␣␣a2,␣16(a0)");\
asm volatile("li␣␣␣␣␣␣a2,␣47" ::: "a2");\
asm volatile("mul␣␣␣␣␣a2,␣a1,␣a2" ::: "a2");\
asm volatile("#sw␣␣␣␣␣␣a2,␣20(a0)");\
asm volatile("li␣␣␣␣␣␣a2,␣49" ::: "a2");\
asm volatile("mul␣␣␣␣␣a2,␣a1,␣a2" ::: "a2");\
asm volatile("#sw␣␣␣␣␣␣a2,␣24(a0)");\
asm volatile("li␣␣␣␣␣␣a2,␣51" ::: "a2");\
asm volatile("mul␣␣␣␣␣a2,␣a1,␣a2" ::: "a2");\
asm volatile("#sw␣␣␣␣␣␣a2,␣28(a0)");\
asm volatile("li␣␣␣␣␣␣a2,␣53" ::: "a2");\
asm volatile("mul␣␣␣␣␣a2,␣a1,␣a2" ::: "a2");\
asm volatile("#sw␣␣␣␣␣␣a2,␣32(a0)");\
asm volatile("li␣␣␣␣␣␣a2,␣55" ::: "a2");\
asm volatile("mul␣␣␣␣␣a1,␣a1,␣a2" ::: "a2");\
asm volatile("#sw␣␣␣␣␣␣a1,␣36(a0)");

#define xM2 xM1 xM1
#define xM4 xM2 xM2
#define xM8 xM4 xM4
#define xM16 xM8 xM8
#define xM32 xM16 xM16
#define xM64 xM32 xM32
#define xM128 xM64 xM64
#define xM256 xM128 xM128

#define x1 \
asm volatile("li␣␣␣␣␣␣a1,␣10" ::: "a1");\
asm volatile("sh2add␣␣a2,␣a1,␣a1" ::: "a2"); \
asm volatile("sh1add␣␣a3,␣a2,␣a1" ::: "a3");\
asm volatile("sh1add␣␣a3,␣a3,␣a1" ::: "a3");\
asm volatile("#sw␣␣␣␣␣␣a3,␣0(a0)");\
asm volatile("sh1add␣␣a3,␣a1,␣a1" ::: "a3");\
asm volatile("sh1add␣␣a4,␣a3,␣a1" ::: "a4");\
asm volatile("sh2add␣␣a5,␣a4,␣a1" ::: "a5");\
asm volatile("#sw␣␣␣␣␣␣a5,␣4(a0)");\
asm volatile("sh2add␣␣a4,␣a4,␣a4" ::: "a4");\
asm volatile("#sw␣␣␣␣␣␣a4,␣8(a0)");\
asm volatile("sh3add␣␣a4,␣a1,␣a1" ::: "a4");\
asm volatile("sh1add␣␣a4,␣a4,␣a1" ::: "a4");\
asm volatile("sh1add␣␣a4,␣a4,␣a1" ::: "a4");\
asm volatile("#sw␣␣␣␣␣␣a4,␣12(a0)");\
asm volatile("sh2add␣␣a4,␣a2,␣a1" ::: "a4");\
asm volatile("sh1add␣␣a5,␣a4,␣a1" ::: "a5");\
asm volatile("#sw␣␣␣␣␣␣a5,␣16(a0)");\
asm volatile("sh1add␣␣a4,␣a4,␣a2" ::: "a4");\
asm volatile("#sw␣␣␣␣␣␣a4,␣20(a0)");\
asm volatile("sh2add␣␣a4,␣a3,␣a1" ::: "a4");\
asm volatile("sh2add␣␣a5,␣a4,␣a3" ::: "a5");\
asm volatile("#sw␣␣␣␣␣␣a5,␣24(a0)");\
asm volatile("sh2add␣␣a2,␣a2,␣a2" ::: "a2");\
asm volatile("sh1add␣␣a2,␣a2,␣a1" ::: "a2");\
asm volatile("#sw␣␣␣␣␣␣a2,␣28(a0)");\
asm volatile("sh2add␣␣a2,␣a4,␣a1" ::: "a2");\
asm volatile("#sw␣␣␣␣␣␣a2,␣32(a0)");\
asm volatile("sh3add␣␣a2,␣a3,␣a3" ::: "a2");\
asm volatile("sh1add␣␣a1,␣a2,␣a1" ::: "a1");\
asm volatile("#sw␣␣␣␣␣␣a1,␣36(a0)");
#define x2 x1 x1
#define x4 x2 x2
#define x8 x4 x4
#define x16 x8 x8
#define x32 x16 x16
#define x64 x32 x32
#define x128 x64 x64
#define x256 x128 x128

static void shx(benchmark::State& state) {

    for(auto _ : state) {
        x256
    }
}

static void explicit_mul(benchmark::State& state) {
    int64_t a;

    for (auto _ : state) {
        xM256
    }
}
```

Listing 4. Testing the performance of two sequence of multiplications