Carry-Less Multiplication in Rocket Core

MohamedHusain NoorMohamed
Electrical and Computer Engineering Dept
Virginia Tech
Blacksburg, USA
nkhusain@vt.edu

FNU Jayakumar
Electrical and Computer Engineering Dept
Virginia Tech
Blacksburg, USA
jayck@vt.edu

***Abstract*—In this project, we implemented a RISC-V instruction called Carry-Less Multiplication Instruction (CLMUL) in Rocket Core to improve the performance of cryptography applications. Carry-Less Multiplication has been used in finite field arithmetic which is prevalent in encryption and checksum computations. Scalar Cryptography Extensions are added in RISC-V ISA, to enable accelerated Cryptography applications, including the CLMUL instruction. The Rocket Chip Core implements the RV64GC which does not support CLMUL Instruction. We have implemented CLMUL instruction with micro-architecture changes in Rocket Core and tested the performance gains with existing cryptography applications. In our results, we show that hardware implementation of CLMUL instruction in Rocket Core takes fewer cycles with other improvements for cryptography applications compared to software implementation.**

***Index Terms*—RISC-V, ISA, Cryptography, AES, GCM, Galois Fields(2^m)**

I. Introduction

The rise of small portable devices has been exponential in recent years like smartphones, tablets, and smartwatches. Many of the computations like image processing are computed within the devices themselves. So it is highly important to identify the resource-intensive tasks and make them more efficient by introducing processing units in micro-architecture to avoid software computations.

It is very common for ISA vendors to introduce new instructions to avoid software computations. ARM introduced the SIMD extensions to aid multimedia applications like audio and video encoders [1]. Cryptography code like encryption and decryption is necessary for all devices. RISC-V group introduced Scalar Cryptography Extension to enable the acceleration of cryptography workloads [2].

A. Finite Field Arithmetic

Galois Field, commonly known as finite field, refers to a field in which there exist finitely many elements. Galois field is just a set of integers mod p, where p is a prime number. Finite field arithmetic is different from existing rules for integer addition and multiplication. The finite field variant GF(2)(Binary Galois fields,p=2) allows easier mathematical operations which are used extensively in cryptography [3].

B. Carry-less Multiplication

In finite fields, the addition and multiplication operations have a property called ”carry-less”, which means there is no propagation of carry. The Carry-less Multiplication uses the XOR operation instead of ADD operation for intermediate results. It has been used widely in the cryptography workloads like elliptic curve cryptography, checksum, and error correction codes [**?**]. Before the introduction of CLMUL instruction, it has been computed in the software (like the code below) using existing ADD and XOR instructions which were inefficient. Figure 1 illustrates the computational approach of Ordinary Multiplication and Carry-Less Multiplication.

```
/*CarryLess Mulitplication Implementation in S/W*/
uint32_t ClMultiply(uint32_t op_a,uint32_t op_b)
{
    uint64_t result = 0;
    for( size_t i = 0; i < 32; ++i )
    {
        if( (op_a >> i) & 1 )
        {
            // If the bit is set, then
            //"add" (xor)a left-shifted
            // version of op_b to the result
            result ^= uint64_t(op_b) << i;
        }
    }
    return result;
}
```

Listing 1. Carry-Less Multiplication logic

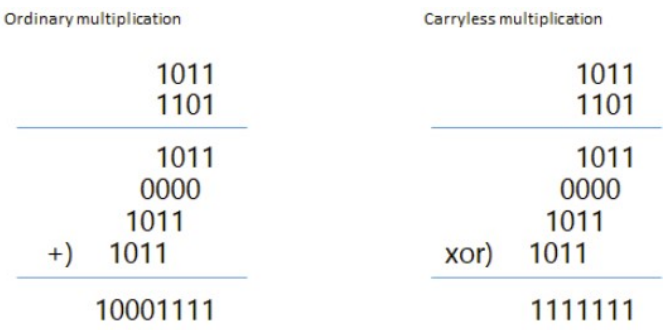

Fig. 1. Carry-Less Multiplication

C. Rocket Core

Rocket is an five-stage integer pipeline, in-order, single-issue scalar processor developed at University of California, Berkeley. Its supports the RV64GC RISC-V instruction set written in Chisel hardware construction language. Since it does not support any cryptography ISA extensions, we will be implementing the CLMUL instruction for this processor.

D. Advanced Encryption Standard

Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S government. AES is implemented in software and hardware throughout many applications to encrypt sensitive data [4]. It is essential for government computer security, cybersecurity, and electronic data protection. We chose an AES encrypt/decrypt example to demonstrate the performance improvements of our design.

II. Related Work

Carry-less multiplication has been adopted for its efficiency in cryptography by major ISA's like ARM, x86, and RISC-V. It also has been implemented in some of the RISC-V simulators such as the spike simulator.

A. Intel x86

The CLMUL instruction in x86 performs carry-less multiplication of two 64-bit operands. Galois counter mode performs carry-less multiplication of 128-bit operands, producing a 255-bit product. This is the first step of computing a 'Galois hash', which is part of the GCM mode [5]. The CLMUL instruction computes the 127-bit product of two 64-bit operands. It can be used by software as a building block for generating the 255-bit result required for GCM. CLMUL helps speedup the computation of CRC with polynomials other than the iSCSI polynomial, over the dedicated existing ISA instruction CRC32 [5].

B. Spike RISC-V ISA Simulator

Spike, the RISC-V ISA Simulator, implements a functional model of one or more RISC-V harts and models a RISC-V core and cache system. Spike simulator supports many scalar cryptographic extensions viz, Zbkb, Zbkx, Zknd, Zkne, Zknh, Zksed, Zksh including the Carry-Less Multiplication Instruction Zbkc [6].

III. Proposed Method

Our proposed method can be described in three sections.

A. Instruction Format

The instruction is of the format R-Type. Thus the instruction reads two source registers *rs1* and *rs2* as source operands and writes the result into register *rd*. Our CLMUL RISC-V instruction in the Rocket Core will have the following format.

- Carry-less multiply (low-part)
  clmul rd, rs1, rs2
- Carry-less multiply (high-part)
  clmulh rd, rs1, rs2

rd is the destination register

rs1 and rs2 are the source registers

The instructions described above are referred from existing RISC-V ISA extensions [7].

B. Opcode for CLMUL instruction

The RISC-V Standard Extension offers custom-0 and custom-1 instructions which can be used in custom implementations and it will be not used in any future extensions. From figure Figure 4 , we can derive that the opcode for custom-0 instruction as *0xb*. We utilized this opcode for our CLMUL instructions. Since R-Type instructions have *func3* field, it can be used to differentiate between *clmul* and *clmulh* instruction i.e both the proposed instructions share the same opcode and differ only in *func3* fields.

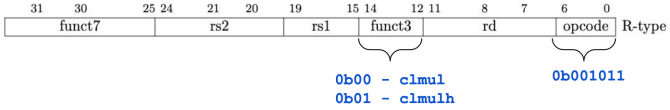

Fig. 2. CLMUL instruction Description

C. Implementation in Rocket Core

For adding new instructions there are two methods available. First, the instruction-specific functional unit can be added to Rocket core. Another method is to use the Rocket Custom Co-processor (RoCC), which holds the functional unit. The second method might degrade the performance nullifying the performance gains we achieve with the new instruction. So we are going with the first method of adding a Functional Unit in the execution stage.

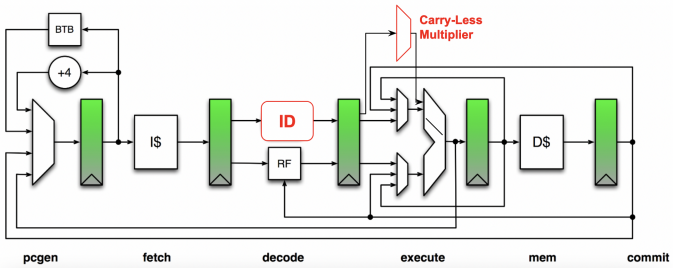

Fig. 3. Rocket Core updated pipeline - Components in red indicates the changes in our design

There are two major changes that we had to do Rocket Core design.

- **Updating the instruction decoder to identify the new CLMUL opcode:** This step is already implemented with SCIE. The Simple Custom Instruction Extension (SCIE) in the Rocket, the core allows the implementation of custom instructions with two sources and one destination register. The SCIE implements the custom-0 instruction in the RISC-V GC extension. So whenever there is a clmul or clmulh instruction in the program the SCIE block will be called.

- **Implementing Carry-Less Multiplier unit in execution stage:** The functionality of SCIE block is implemented in Verilog HDL. We wrote the Carry-Less Multiplication logic in Verilog and added it to the SCIE block.

```
module SCIEUnpipelined #(parameter XLEN = 32) (
          input  [${SCIE.iLen-1}:0] insn,
          input  [XLEN-1:0] rs1,
          input  [XLEN-1:0] rs2,
          output [XLEN-1:0] rd);
```

Listing 2. SCIE Module Definition

Figure 3 describes our changes in the existing Rocket Core pipeline.

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Fig. 4. RISC-V base opcode map, inst[1:0]=11 [8]

D. Simulation in iVerilog

For the implementation of Carry-Less Multiplication logic in Verilog, we have to check the correctness of our logic. So we did our Verilog simulations in an open-source Verilog simulation and synthesis tool called Icarus Verilog (iVerilog) [9]. Since the build time for Rocket Core took a considerable amount of time, iVerilog helped us to test our changes easily and compare them with the online CLMUL calculator [10]. Figure 5 shows our simulation output in GTKWave .

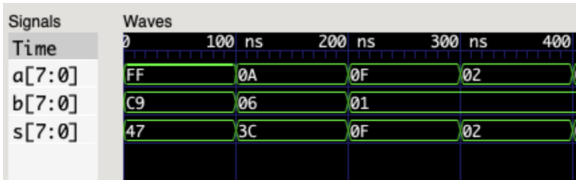

Fig. 5. iVerilog Simulation output in GTKWave

E. Binutils Changes

In order to compile the applications that leverages CLMUL instruction, the riscv-gnu-toolchain need to be modified.

```
// opcodes/riscv-opc.c

struct riscv_opcode riscv_opcodes[] =
{
...
...
{    "clmul",       // name
     64,            // xlen
     {"I", 0},      // ISA subset name
     "d,s,t",       // arguments
     MATCH_CUSTOM0_CLMUL,   // opcode
     MASK_CUSTOM0_CLMUL,    // opcode mask
     match_opcode, }        // match function
...
...
}
```

Listing 3. Adding CLMUL instruction in risc-v-binutils following RISC-V opcode format

Since it is hard to entirely write assembly instructions for cryptography applications, we compiled implementation in C language with an assembler that supports the instruction, since it is easier to evaluate and debug. Whenever there is a use of CLMUL instruction in the C program, we just add the instruction manually with `asm volatile()` macro.

```
$ riscv64-unknown-elf-objdump  -D  clmul.o  | grep -A2 -B2 clmul

clmul.o:     file format elf64-littleriscv


--
 344:   fd843783                ld      a5,-40(s0)
 348:   fd043703                ld      a4,-48(s0)
 34c:   00e7878b                clmul   a5,a5,a4
 350:   fef43423                sd      a5,-24(s0)
 354:   fe843583                ld      a1,-24(s0)
```

Fig. 6. Updated RISC-V binutils that identifies clmul instruction

After making the changes for the instructions, the binutils were able to identify the instructions. Figure 6 shows an objdump output of compiled binary which identifies the new instruction.

IV. Evaluation Method

For evaluation we ran programs in Verilator simulation tool. We chose two set of programs.

- **Simple program that computes only Carry-Less Multiplication(clmul.c):** This program calculates the clmul value of two 32-bit source operands. There will be two variants, hardware implementation which uses *clmul* instruction and software implementation which computes Carry-Less Multiplication in with loops and xor operations. We ran these two variants with 1000 randomly chosen operands to make sure both software and hardware implementations compute the same results.
- **AES Encryption/Decryption with three different key lengths:** We also want to test our implementation with a cryptography application, so we chose the AES encryption/decryption programs. AES has different modes and most commonly used is the AES-GCM mode which provides a high speed of authenticated encryption and data integrity. In total, we tested six AES encryption/decryption programs with varying key sizes and operating modes.
  - AES-GCM-BASE - 128 Bit key (base-128)
  - AES-GCM-BASE - 196 Bit key (base-192)
  - AES-GCM-BASE - 256 Bit key (base-256)
  - AES-GCM-KAR - 128 Bit key (kar-128)
  - AES-GCM-KAR - 196 Bit key (kar-192)
  - AES-GCM-KAR - 256 Bit key (kar-256)

  ACM-GCM-BASE version uses School Book Multiplication algorithm and ACM-GCM-KAR uses an efficient Karatsuba Multiplication. These modes still use Carry-less Multication in their fundamental steps.

V. Evaluation Results

A. clmul.c results

We present the results of running clmul.c for 100 and 1000 different operands and its shows the number of cycles executed for H/W implementation is only 10 percent compared to S/W implementation.

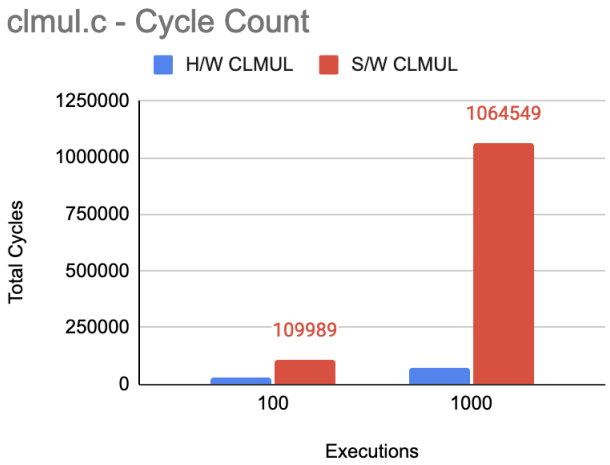

Fig. 7. clmul.c cycles count

B. AES Application results

- **Cycle Count**:
  Figure 8 shows the number of cycles executed for the different modes of the AES programs. The cycle counts of H/W implementation are only half of the S/W implementation.

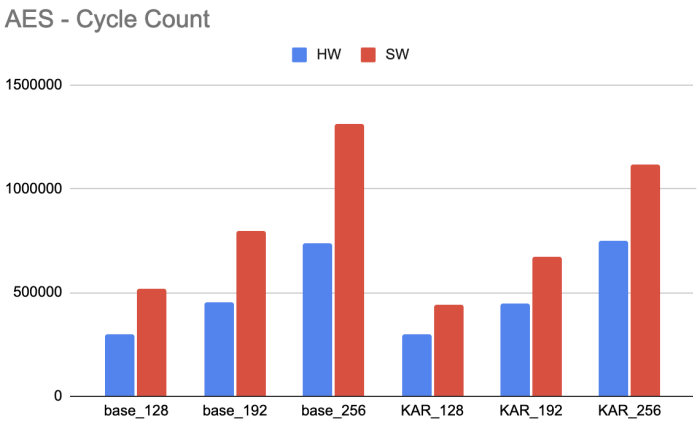

Fig. 8. AES Programs - Cycle Count

- **Instructions Executed**: As expected Instruction count is also reduced for H/W clmul implementation - Figure 9.

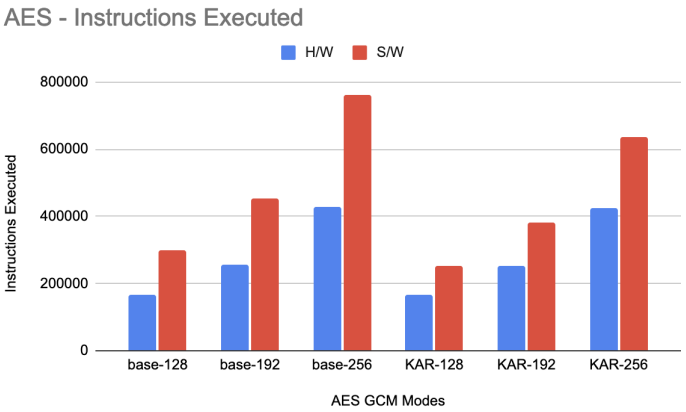

Fig. 9. AES Programs - Instructions Executed

- **Branches and Mispredictions**: Another interesting parameter we noticed is the number of branches and mispredictions reduced for H/W clmul - Figure 10 and Figure 11. Fewer branch and mispredictions means fewer branch hazards and pipeline would be efficiently utilized.

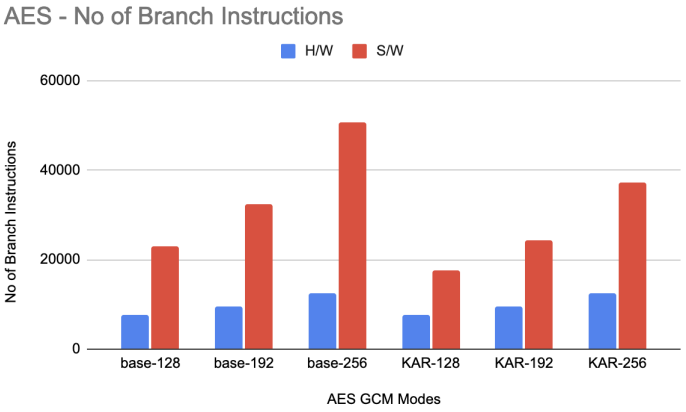

Fig. 10. AES Programs - Number of Branch Instructions

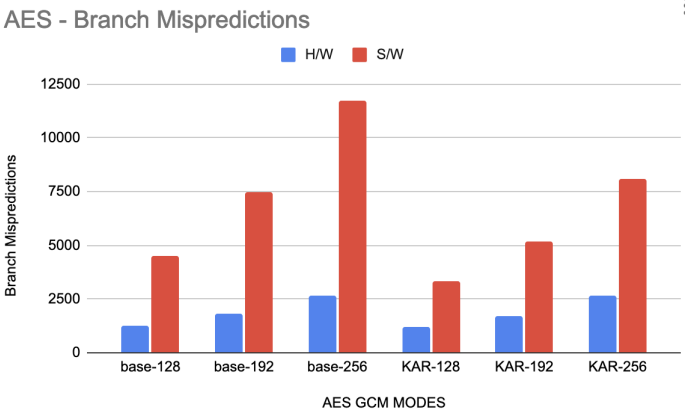

Fig. 11. AES Programs - Branch Mispredictions

VI. Timeline

- **Week 1**
  Proposed: Understand the SCIE in Rocket Chip Generator and refer to previous implementations for adding new instructions.
  Actual: Able to understand SCIE, but could not find any examples for adding new instructions.
- **Week 2**
  Proposed: Implement Carry-less multiplication instruction.
  Actual: Started with Verilog simulation in iVerilog but could not add it to SCIE. We faced some build errors, which took a lot of time to debug since the build took considerable time.
- **Week 3**
  Proposed: Complete implementing new instruction in Rocket Core (SCIE) and test. Start adding support in RISC-V toolchain to build the target programs.
  Actual: Spend the week adding a new instruction in the toolchain, which was pretty straightforward.
- **Week 4**
  Proposed: Complete toolchain changes and add CLMUL instructions to the target programs.
  Actual: Resumed fixing the build errors in SCIE, able to run a clmul instruction in Rocket Core.
- **Week 5**
  Proposed: Compile the target programs with an updated toolchain and run the programs to get the metrics. Prepare for project presentation in parallel.
  Actual: We found an AES example but it used Intel's HW Cryptographic libraries. So started replacing library calls with manual computations but it lot took of time even for small portion, so we dropped that example. We found another repository that did not use any H/W accelerators.
- **Week 6**
  Proposed: Analyze the results and prepare project report.
  Actual: Completed adding clmul instructions to the new AES example we found and ran the benchmarks. We spent a couple of days writing the report.

VII. Contributions of individual

A. MohamedHusain NoorMohamed

- Understanding SCIE and Custom Instructions in Rocket Core.
- Added new instruction support in RISC-V toolchain.
- Implemented clmul logic in SCIE.
- Tested with randomized inputs on clmul. c.

B. FNU Jayakumar

- Prepared Presentation Slides.
- Finding AES programs examples and add clmul instructions in software clmul calculations.
- Ran simulations in iVerilog
- Prepared Final Report.

VIII. Conclusion

In this project, we proposed to implement a Carry-Less Multiplication (CLMUL) logic in Rocket Core to improve the performance of Cryptography applications. We made necessary microarchitecture changes and compiler changes for the new CLMUL instruction. We accelerated an AES encryption algorithm and observed the improved performance results compared to the baseline version without hardware CLMUL implementation. We showed with results that the Hardware implementation of CLMUL logic in Rocket Core took 10 times lesser cycles than the Software implementation. We learnt more about Rocket Core, RISC-V ISA, and Verilog simulations.

References

[1] ARM Developer. Arm architecture reference manual, 2017. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Application-Level-Programmers--Model/Advanced-SIMD-and-Floating-point-Extensions.

[2] RISC-V Group. Scalar cryptography v1.0.0-rc2, Sep 2021. https://github.com/riscv/riscv-crypto/releases/tag/v1.0.0-rc2-scalar.

[3] Francisco Garcia-Herrero Kuo, Yao-Ming and Juan Antonio Maestro. Versatile risc-v isa galois field arithmetic extension for cryptography and error-correction codes. In *Fifth Workshop On Computer Architecture Research With RISC-V (CARRV)*, 2021.

[4] Michael Cobb Corinne Bernstein. Advanced encryption standard (aes), Last Accessed Dec 2022. https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard.

[5] Michael E. Kounavis Shay Gueron. Intel® carry-less multiplication instruction and its usage for computing the gcm mode, May 2010. https://www.intel.cn/content/dam/www/public/us/en/documents/white-papers/carry-less-multiplication-instruction-in-gcm-mode-paper.pdf.

[6] RISV-Software-Src Group. Spike risc-v isa simulator, Dec 2021. https://github.com/riscv-software-src/riscv-isa-sim.

[7] RISC-V group. Risc-v bit-manipulation a, b, c , and s extensions, July 2022. https://five-embeddev.com/riscv-bitmanip/draft/bitmanip.html.

[8] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[9] iVerilato. Verilog simulation and synthesis tool, Oct 2021. https://www.veripool.org/verilator/.

[10] University of New Brunswick. Galois field gf(2) calculator, Last Accessed Dec 2022. http://www.ee.unb.ca/cgi-bin/tervo/calc.pl.