Towards a RISC-V Instruction Set Extension for Multi-word Arithmetic

Youngjin Eum, Naifeng Zhang, Larry Tang, Franz Franchetti
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA
{yeum, naifengz, lawrenct, franzf}@andrew.cmu.edu

I. Introduction

Multi-word arithmetic is a method of doing calculations with data which are bigger than what the machine registers can hold. For example, we could do 128-bit arithmetic with a 64-bit machine by using this technique. One application of multi-word arithmetic is in cryptography, since working with large numbers is a critical security feature [1]. Currently, many CPUs and GPUs support multi-word arithmetic by providing a single carry flag. This is useful for implementing simple 128-bit operations such as addition and subtraction, but is limited with more complicated operations such as multiplication or modulo.

Contributions. Our key contributions are:

- An extension to the RISC-V ISA to accelerate multi-word arithmetic incorporating multiple carry bits to enable modulo and multiplication.
- An implementation of the ISA using the Chipyard framework [2] and the RoCC interface, seeing up to 1.5× speedup in clock cycles.

II. Background

Since multi-word operations will use multiple instructions to compute a single result because the source and destination operands do not fit in a single register, we need to keep track of state, usually a carry flag, through multiple instructions.

An instruction can have a carry in, a carry out, or both. Since an addition of two N-bit numbers will produce a result which is $N + 1$ bits wide, an instruction with a carry out can set a carry bit based on if the $(N + 1)$-th bit was set. A subsequent instruction can use the carry bit as a carry-in to the addition, in effect computing a $2N$-bit addition.

Many ISAs such as x86 or ARM have a carry flag which gets set every time an operation produces one. RISC-V does not support an explicit carry flag, so programmers must add another instruction detecting if the result overflowed to implement multi-word arithmetic.

Prior work [1], [3] has explored the use of multi-word arithmetic on GPUs in a cryptographic context, specifically opting for NVIDIA GPUs. The GPUs support only 32-bit integer operations, so they needed to use carries to implement 64-bit integer operations. Similar to other ISAs, the CUDA architecture uses a carry flag to implement multi-word arithmetic. Another approach to multi-word arithmetic is to have a coprocessor take the input over multiple cycles. Since every clock cycle we can read from two source registers and write to one, by having a coprocessor with an input mode, compute mode, and output mode, we can work with batches of data wider than machine registers [4].

III. Approach

We implemented the custom ISA using Chipyard v1.12, an open-source SoC platform. Specifically, we have used RoCC (Rocket Chip Coprocessor) interface with the Rocket core to quickly prototype the accelerator.

As opposed to the carry flag in commonly used ISAs, there are four distinct carry flags in our extension. This allows not only for strings of additions and subtractions, but more complex operations which need to keep track of multiple carry bits. For example, we can multiply two 128-bit numbers with carry-in and carry-out using multiple distinct carry bits. We could also use Barrett reduction [5] to compute the modulo of two 128-bit numbers using only addition and multiplication operations, keeping track of carry bits in the operations separately.

Figure 1 illustrates the encoding of an add with carry instruction, and an example implementation in Chisel HDL is shown in Listing 1.

The encoding is identical to the standard R-type instruction in RISC-V. The `A`, `CI`, `CO` bits within the funct7 field encode whether the instruction is an addition or a subtraction, consumes a carry in, and produces a carry out respectively. The `cout_idx` and `cin_idx` fields address into the four

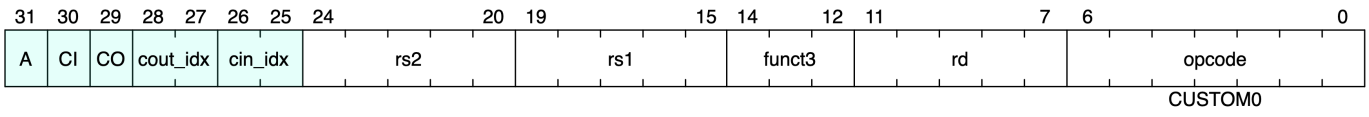

Fig. 1: Encoding of addition/subtraction with carry instruction.

```
1  class RoCCCarryModule(outer: RoCCCarry) extends
   LazyRoCCModuleImp(outer) {
2  val opcode = io.cmd.bits.inst.opcode
3  val funct7 = io.cmd.bits.inst.funct
4  val cout_index = funct7(3, 2)
5  val cin_index = funct7(1, 0)
6  val sum = Wire(UInt(65.W))
7
8  when (funct7(6) === 0.U) {
9    when (funct7(5) === 1.U) {
10     // carry in
11     sum := io.cmd.bits.rs1 +& io.cmd.bits.rs2 +&
       regs(cin_index)
12   } .otherwise {
13     // no carry in
14     sum := io.cmd.bits.rs1 +& io.cmd.bits.rs2
15   }
16
17   when (funct7(4) === 1.U && io.cmd.fire) {
18     // carry out
19     regs(cout_index) := sum(64)
20   }
21 }
22 // ... omitted
23 io.resp.bits.data := sum(63, 0)
```

Listing 1: Implementation of addition with carry instruction.

carry bits to specify which bit is used in the instruction. We instantiate a small register file in order to implement the four carry flags. We index into it to access the carry in bits as in line 11, and write to it if we see a carry out as in line 19. Note the use of the `+&` operator in Chisel to preserve the carry.

To allow for even more flexibility in the software, there are also instructions to load and store from the carry bit registers to the regular integer registers. This enables computation directly with the carry bits if the software deems it necessary.

IV. Results

The number theoretic transform (NTT) is a generalization of the discrete Fourier transform on finite fields and serves as a core kernel in many encryption schemes such as homomorphic encryption. We benchmarked the RoCC accelerator on NTT code generated by the SPIRAL NTTX package [6], [7] which uses 128-bit arithmetic. We chose the NTT as a benchmark since it is one of the driving applications of multi-word arithmetic. We implemented the core operations in the NTT such as modular addition/subtraction and multiply using the new instructions for preliminary results and used 8-point and 1024-point NTT to study performance across input size. This includes code which only uses the `uint64_t` type and computes the carries manually, and code which uses the `__int128` type, which uses compiler-optimized arithmetic. We measure clock cycle counts by reading the `cycle` CSR immediately before and after the NTT function call. We also ran the NTT kernel 20 times, discarded the first 10 to mitigate the effect of cold caches and averaged the rest of the trials to obtain results.

As illustrated in Figure 2, we see about a 1.5× speedup in comparison to the `__int128` implementation, and a 3× speedup in comparison to the `uint64_t` implementation in the 8-point NTT case. The results are similar in the 1,024-point NTT case as well, with the RoCC code having approximately 1.3× speedup over the best baseline. One reason for the speedup is that the processor executes fewer instructions with the ISA extension, which reduces fetch-decode overhead.

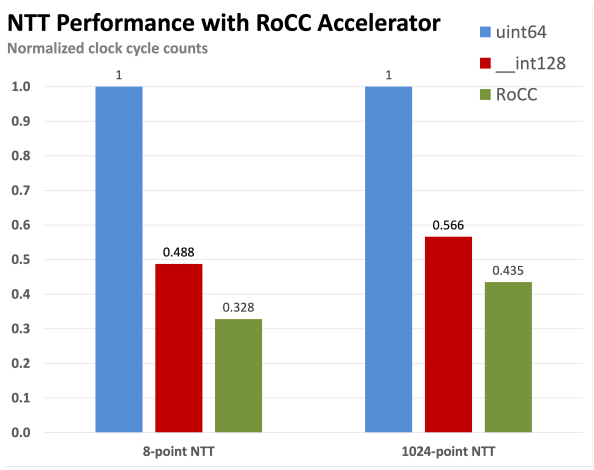

Fig. 2: Clock cycle counts for 8-point and 1024-point NTT.

V. Conclusion and Future Work

In this work, we present an ISA extension which makes use of multiple carry registers to enable efficient multi-word arithmetic. We present an implementation of the ISA extension and benchmarked it against the conventional implementations of multi-word arithmetic, showing performance improvements.

We look to implement a standalone functional unit which can interface with an out-of-order BOOM core in the future. This will allow for us to take advantage of out-of-order issue which will lead to additional performance.

References

[1] N. Zhang and F. Franchetti, "Generating number theoretic transforms for multi-word integer data types," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2023.

[2] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 2–6, 2016.

[3] G. Fan, F. Zheng, L. Wan, L. Gao, Y. Zhao, J. Dong, Y. Song, Y. Wang, and J. Lin, "Towards faster fully homomorphic encryption implementation with integer and floating-point computing power of gpus," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 798–808.

[4] T. Fritzmann, G. Sigl, and J. Sepúlveda, "Extending the risc-v instruction set for hardware acceleration of the post-quantum scheme lac," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 1420–1425.

[5] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Proceedings on Advances in Cryptology—CRYPTO '86*. Berlin, Heidelberg: Springer-Verlag, 1987, p. 311–323.

[6] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, "Spiral: Extreme performance portability," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018.

[7] N. Zhang, A. Ebel, N. Neda, P. Brinich, B. Reynwar, A. G. Schmidt, M. Franusich, J. Johnson, B. Reagen, and F. Franchetti, "Generating high-performance number theoretic transform implementations for vector architectures," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2023, pp. 1–7.