An Optimized Finite Field Arithmetic Code Generator:
CPSC 490 Project Report

Charles Jin

April 30, 2015

1 Introduction

This project sought to alleviate a particular concern when building cryptosystems - how to generate efficient code for arithmetic operations over a chosen field. Following the footsteps of Dan Bernstein’s work on Curve25519 [1], and more recently Curve41417 [2], I attempted to generalize his methods to arbitrary fields.

The project’s scope was limited to automatically generating optimized code for most common functions (addition, multiplication), given a canonical representation of a number in the field. Since the primes in question are generally too large to fit in any standard datatype, all numbers in the field must be broken up into a virtual array of machine words. The user inputs the prime, as well as how the elements of the field were to be broken up into a series of machine words, and gets as output the required functions. Thus, the primary usage would be to allow the user to test various chosen representations for a particular prime with robust automatically-generated code for testing and prototyping.

For a more complete description of the problem, please see the original proposal. The most recent source code can be found at https://github.com/charlesjin/codegen.

2 Progress

I am happy to announce that I can successful generate highly optimized code for multiplication, squaring, and squaring-and-doubling. In particular, the code matches
up with the hand-tuned optimizations of [1]. This presented, in and of itself, a huge technical challenge, both in terms of implementation and theory.

In particular, great care was taken in identifying efficient common subexpression elimination and pre-computations. A naive implementation of the schoolbook method described in the proposal would be correct, but require many repetitive computations. Lifting common subexpressions substantially improves performance, and in fact, is the entire point of the hand optimized displayed in, for example, [1] and [2].

Apart from generating optimized code, I also kept careful documentation of both necessary and desired qualities of primes and the provided representation. For example, if the multiplication at any step would produce an overflow, the generator exits and displays an error message. Thus, the generator provides a strong guarantee that the code produced is also correct. This allows users to test various representations without being worried about correctness, or even doing too much theoretical optimization - they can merely run their representation through the generator and inspect the output.

Additionally, all of the less technical functions were implemented. A full list of completed functions is as follows:

1. feZero - generates the 0 field element.
2. feOne - generates the 1 field element.
3. feAdd - adds two field elements.
4. feSub - subtracts two field elements.
5. feCopy - copies a field element from a destination to a source.
6. feCMove - conditionally copies a field element from a destination to a source.
7. load3 - converts and saves the first three bytes of a byte array representing some number into an int32.
8. load4 - converts and saves the first four bytes of a byte array representing some number into an int32.
9. feIsNegative - returns true if the number represented by some byte array is negative, and false otherwise.
10. feIsNoneZero - returns true if the number represented by some byte array is nonzero, and false otherwise.
11. feNeg - returns the negation of a field element.

12. feString - prints out the representation of a field element.

13. feFromBytes - converts a byte array representing some number into the canonical representation specified by the user.

14. feMul (partially complete) - multiplies two field elements.

15. feSquare (partially complete) - squares a field element.

16. feSquare2 (partially complete) - squares, then doubles, a field element.

For the partially complete functions, see the Future Work section for an explanation.

3 Implementation

Implementation proceeded in three general steps. First, generate an abstract syntax tree; second, perform some additional optimizations on the abstract syntax; finally, print the code out to proper Go syntax.

3.1 The Abstract Syntax and (Pretty-)Print Modules

I will only briefly talk about the abstract syntax and the printing module here, mainly because they are relatively standard and clear just from inspecting the code. The abstract syntax is a very limited subset of Go, and somewhat littered with hacks, however, because the subset of necessary Go constructs was also extremely limited, it turned out to be cleaner and more expressive to adopt a tailored abstract syntax.

The (pretty-)print module attempted, as much as possible, to output clean and well-formatted code, for the purpose of allowing the user to easily go through and identify potential shortcomings of the code generated from the chosen representation.

3.2 Generating the Abstract Syntax, and Some First-Pass Optimization Techniques

Generating the abstract syntax tree (AST) was the more substantial portion of the project. Apart from implementing the multiplication algorithms correctly as outlined in the proposal, I was also concerned with outputting efficient optimized code.
This was accomplished via identifying two forms of precomputations. First, there was dealing with potentially uneven representations. When multiplying two blocks $a_i$ and $b_j$, their actual representation might be, for example, $2^{26}a_i$ and $2^{51}b_j$. If this product is to fit in a block of prefix $2^{76}$, then what we actually need is $2^{a_i}b_j$. It turns out we can lift this sort of equalization of different block prefixes to a precomputation step in a systematically efficient manner, by scanning over all possible equalizations, and picking some minimum cover of computations necessary.

The second form of precomputation dealt with the special end-around carry. When dealing with a field of size, for example $2^{255}-19$, using a 255-bit representation, any bits past the 255th can be dealt with by multiplying by 19, then adding back into the least-significant bits. Thus, this multiplication by 19 is also a good target for precomputations, again accomplished by scanning over all necessary carries and finding a minimum cover.

3.3 Some Second-Pass Optimization Techniques on the AST

In the final optimization step, the main concern was efficient instruction scheduling. In general, this step attempts to reorder the various precomputations in such a way that doesn’t block the CPU pipeline and minimizes the potential of moving things in and out of arithmetic registers.

In the first case, the reordering seeks to make it such that a variable that has just been assigned to is not needed in any of the next couple operations. That way, the CPU does not have to wait for the first operation to finish before continuing with the rest of the calculations, allowing steps to run in parallel.

In the second case, we would like to order operations that share an operand next to each other. That way, the shared operand can stay in register, and we can potentially save a few mov instructions.

It should be noted that both these optimization techniques are system- and even compiler-dependent. However, in most cases these techniques should, at the very least, not result in slower code, and in the average case, I expect that the optimizations, which are very specific to the structure of the code, should be able to beat the compiler’s optimizations.

3.4 Further Descriptions of Algorithms

More detailed descriptions of algorithms can be found in the commented sections of the source code preceding the corresponding functions. In general, the class of set-cover problems is NP-complete, and optimizing is NP-hard, so the algorithms
employ a single-pass heuristic algorithm for selecting a good cover of small size. The expectation is that, for a good representation, the amount of precomputations should be relatively small, and so this turns out to be effective for our purposes.

4 Future Work

One particular aspect that has yet to be dealt with is the canonicalization step after each of these operations (multiply, square, square-and-double). The solution seems to be to just keep track, after each operation, of the resulting size of each block, then perform a cascading carry to reduce the blocks down to their canonical size as prescribed by the user’s representation. (In any case, the git repository will be updated as soon as this functionality becomes available.)

5 Conclusion

At the time of writing, the core functionality of canonicalization after operations was missing. However, generating the proper multiplication sequences was already a huge undertaking, especially to the point of being able to match the hand-optimized gold standard of [1]. Further extensions may involve using convex optimization heuristics to automatically generate several correct potential representations (the problem itself turns out to be NP-complete linear program, on account of the modular constraints).

6 Acknowledgements

I would like to thank Professor Bryan Ford and the entire Yale DeDis group for their support, both moral and intellectual. No doubt I would not have been able to make such progress without the occasional direction or discussion.

References