Cipher Implementation for CESEL

Kevin Kiningham
Stanford University

Maurice Shih
Stanford University

Abstract

CESEL is a recently proposed cryptographic architecture that can accelerate a wide variety of ciphers, in contrast to most previous accelerator designs which focus on optimizing for a particular cipher. The architecture features a 32 lane SIMD architecture with 8-bit datapath, specialized execution units, and a control unit designed for cryptographic control flows. In this project we implement several cryptographic ciphers for CESEL, a RISC-V microcontroller, and as a fixed function accelerator. We then evaluate the power efficiency of each design, allowing us to both compare CESEL's performance on different ciphers, and investigate how easy it is to map new ciphers to CESEL.

1 Introduction

Low power embedded devices are increasingly being used to collect and report sensitive data. Storing and transmitting this data encrypted then becomes critically important to ensure the security of the overall system. However, for many of these devices, encrypting data on an embedded microcontroller can strain power budgets, meaning that system designers have to choose between strong encryption and a longer battery life.

To address this issue, many Solutions on Chip (SoCs) include fixed function cryptographic accelerators, which can encrypt data using orders of magnitude less power than a microcontroller. Unfortunately, this solution only works as long as the exact ciphers needed can be predicted over the lifetime of the device; if security requirements change unpredictably (for example, due to changes in regulations, or advances in cryptanalysis) fixed function accelerators no longer provide a significant benefit, requiring hardware to be replaced and upgraded.

A proposed approach to this problem is CESEL, a flexible accelerator for cryptography, designed to accelerate a wide variety of ciphers and cryptographic primitives. In this project, we are implementing several key ciphers and evaluating how easy it is to translate the ciphers into the proposed architecture. Additionally, we are evaluating the overall efficiency of the architecture across multiple different algorithms. We find that CESEL can provide significant acceleration to all ciphers we investigated, providing between a 10x-60x improvement in total energy consumption depending on the cipher. We also find that CESEL uses between 10x-20x more energy than fixed function hardware (Table 1).

2 CESEL Architecture

The design of CESEL is split into two distinct pieces: the frontend, which handles instruction fetch and decode, and the backend, which handles instruction execution.

2.1 Frontend

The CESEL frontend consists of three components, shown in figure 1. The first, the fetch and decode unit, interfaces with the memory bus and decodes fetched instruction into control signals. The second, a loop stack, records loop information and execution context. The third, an instruction queue, holds decoded instructions waiting to be executed.

An important design feature of CESEL is it's lack of data dependent branches. Practical implementations of cryptographic algorithms avoid data dependent branching since it is a common source of timing attacks. Additionally, removing data dependent branches means that control decisions can be made with perfect accuracy after instruction decode, which significantly simplifies CESEL's design.

However, this can also dramatically increase the code size needed for ciphers with many loops. To address this, a special component called a ”loop stack” is used, which

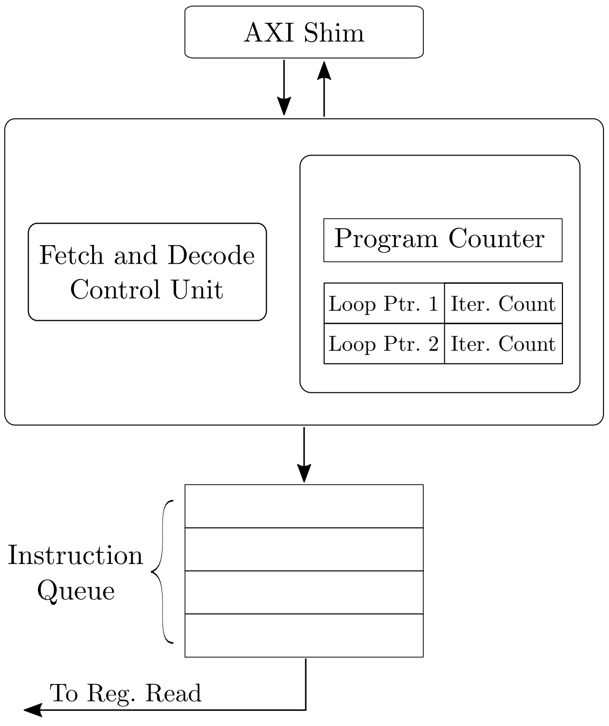

Figure 1: Instruction fetch and decode datapath.

keeps track of state for currently executing loops. To begin a loop, a special instruction `LOOP_BEGIN` pushes the current instruction pointer along with the number of iterations to execute to the top of loop stack. To end a loop, the `LOOP_END` instruction compares the number of iterations at the top of the stack to zero. If it's equal to zero, the top entry is popped off the loop stack and execution continues. Otherwise, the instruction pointer is set the saved instruction pointer and the saved iteration count is decremented.

2.2 Backend

The backend is a 32-lane SIMD architecture, show in figure 2. Each cycle, each lane receives an identical decoded instruction from the frontend and reads the relevant values from it's local register file. Each lane then executes it's instruction and writes the resulting value back to a register. To move values between lanes, a special permute instruction is used. Since the frontend does not allow data dependent branching, there are no hazards except for register read-after-writes (RaW), which are handled using forwarding paths. As a result, an instruction can be executed every cycle as long as the instruction queue is filled.

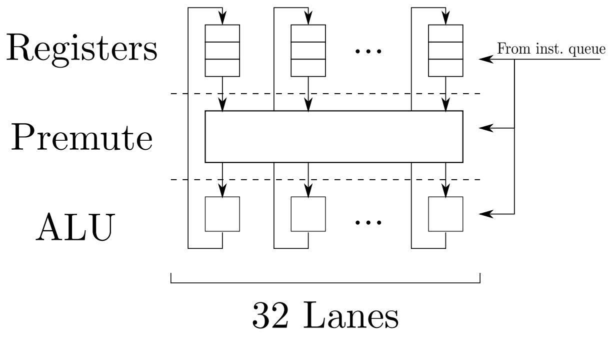

Figure 2: Register read and execution datapath.

3 ChaCha-20 (work by Kevin)

We have implemented ChaCha-20 in CESEL assembly, C, and Verilog. Our CESEL assembly implementation is based on the AVX2 implementation in libsodium[3]. We found that this implementation mapped very closely to CESEL's own instruction set and could be nearly directly used; since ChaCha was designed to execute using simple vectorized instructions, it maps cleanly onto CESEL.

Our C implementation is the reference implementation from libsodium compiled for RISC-V. Our application specific circuit is a fully unrolled implementation based on the implementation by SecWorks.

4 Curve25519 (work by Kevin)

We have implemented Curve25519 key-exchange in C and have a partially working implementation in CESEL assembly. Our C implementation is a direct port of the ref10 implementation by D. J. Bernstein included in libsodium[3].

Our CESEL implementation is based on the Sandy2x[2] by Tung Chou included in libsodium. Unfortunately, this implementation was much more difficult to port to CESEL assembly than ChaCha since it uses several features of x64 that are not present in CESEL. For example, it uses a 32-bit operand/64-bit result vectorized multiply which does not exist on CESEL; instead this operation was implemented using a sequence of 8-bit multiply-accumulate operations. Unfortunately, we did not finish the complete implementation and cannot fairly compare between architectures.

5 SHA-256 (work by Maurice)

We have created a C implementation and CESEL for the SHA-256 hashing [4] of one post-processed 512-bit block of text. Since SHA-256 is constant for one block, we don't worry about branching. In addition, all computation is bit manipulation so we are not dependent on possible branching of operations.

5.1 C Implementation

In the SHA-256 algorithm, 32-bit blocks of data called words are operated on by functions such as shift and rotate. Since unsigned integers in C are 32 bits, many of the operations can implemented through a small number of C bitwise operators.

For example. rotating to the right by n can be done with $(x << n)|(x >> (32-n))$. From bitwise operations like these, the six main functions, ch, maj, $\sum_0$, $\sum_1$, σ_0, and σ_1 of SHA-256 were implemented.

5.2 CESEL Implementation

The biggest bottleneck in this framework is the loading and storing of data. In order to combat this, we load in full registers of data each time we load in data. CESEL has 16 registers, with lanes of 32 8-bit values. Since each value is 8 bits in a lane, 4 values make up one 32 bit word.

Due to having the rotate right assembly instruction and not a shift left, rotating the whole 32 bit or 4 8 bit word is less straight forward. There are four categories of shifting, shifting 0-7 places, 8-15, 16-23, and 24-31. As an example, if we wish to shift by say 3 places, then we need 2 registers for calculation. The first is a copy of the data (the permutation is 0,1,2,3) and a permutation of the four 8 bits as 3,0,1,2. We rotate the two registers by 3. We then and the first lane by 0b00011111 and the second by 0b11100000. The final step is to add the two registers.

For a given main function, such as $\sum_0$, 3 registers are needed, two for manipulation of data and a third to retain the result. The other 12 registers are used to hold data, which can hold 96 32-bit words. At the end of the algorithm, the majority holds the 512 bit (or 64) message block that we will hash.

The bulk of SHA-256 is composed of two for loops in which each step of the for loop is dependent on the values calculated in the previous step. This creates a bottleneck as each step must be computed in serial. This wastes the 32 value lanes as functions are only applied to one 32-bit value at a time (we only care about the computation done on the first four lanes in a register).

6 Frodo LWE (work by Maurice)

We have created a C implementation and CESEL for the Frodo LWE key exchange [1]. This protocol utilizes a common matrix A that is public knowledge, as well as secret matrices $B'S$ and V, which is generated by matrix multiplication and addition of A by error matrices for Alice and Bob respectively. These two matrices are secrets that Alice and Bob have and are very simliar but not identical. Using the reconciliation mechanism, with high probability, Alice and Bob get the identical matrix K from these two matrices that can used as a private session key. In this implementation, we use the recommended parameters with $n = 752$, $q = 2^15$, error distribution of D_3 and $\bar{m} = \bar{n} = 8$.

In our implmentation, we assume that the random bits we need is already generated, so we do not take into account the cycles and power comsumption used to do so. We are counting the totaly amount of work that Alice an Bob must do after random bits are generated fot this key exchange.

6.1 C Implementation

Since all calculations are done in modulo $q = 2^15$, we used unsigned 16-bit integers to represent values and apply the bitwaise and function with $2^15 - 1 = 32767$ when the reduced value is needed. In this protocol, there are four main parts, the matrix multiplication, rounding function, cross function and the reconciliation function.

In the C implementation, matrix multiplication was done with three for loops. For the rounding and cross function, bitwise operations were used to ensure constant running time. For reconciliation function, it was necessary to create a constant time function that given inputs a and b, would output 1 if $a > b$. To achieve this, a new function was created to compare the bits of a and b, starting with the most significant bit. There were two working variables, one to denote if the ith bit of a and b were previously not the same (if this was the case, as already know if $a > b$) and the actual output. Based on the bit that was the greater function output, the reconciliation would output one of two options. However, branching was created.

6.2 CESEL Implementation

All four main parts of the frodo LWE algorithm, the matrix multiplication, rounding function, cross function and the reconciliation function, can each be done in parallel as we apply these functons to multiple values. Specifically, we apply functions to each element in a matrix. Since each register has 32 lanes, this allows each register to represent 16, 16-bit unsigned integers. This allows us to operate 16 values for a single instruction.

As all matrices have a multiple of 16 number of elements, this creates a very natural way of optimizing the protocol by doing operations in parallel. For example, to compute a matrix multiplication of a a by 752 and 752 by b, each entry requires $752/16 = 47$ multiplication instructions (all matrix multiplications are in this

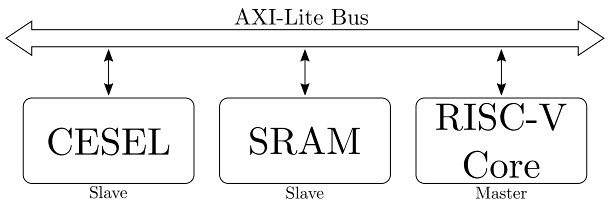

Figure 3: Architectural overview of the test system.

form). However, due to the sheer amount of data the needs to be processed (for example, the matrix A has $752 \cdot 752 = 565504$ values), many loading and storing of values in the registers are needed, which is extremely power costly.

The CESEL architecture only has 16 registers, which means that only 256 16-bit values can be stored at any given time, and that is assuming that we don't need any registers to preform calculations. In reality, many registers are needed for computation so data is loaded and stored often. For example, the size of matrix A that Alice and Bob share is 2209 times the amount of data that the registers can hold. Although the computation time in CESEL is less than C's by a factor of 16 for many of the operations, the load and storing of the data is less efficient and consumes more poewr.

7 Evaluation

We evaluated our cipher implementations by comparing to two baseline systems: an application specific circuit (ASIC) and a C implementation running on a RISC-V microcontroller. In order to accurately measure power differences between systems, we implemented a simple test harness similar to an actual SoC implementation (Fig. 3). CESEL and the RISC-V core are connected over an AXI-Lite bus to a 64K SRAM macro block. All instructions and data are fetched over the AXI bus, although the reported power only includes the execution power, not the power of fetching data.

Power was measured by propagating the activity factors from simulation to a synthesized version of our test harness. Simulation was performed using Synopsys VCS. Synthesis was performed using the Synopsys Design Compiler (with topological mode enabled) targeting a 180nm TSCM process with all power and timing related optimizations enabled. A bottom up flow was used, first synthesizing CESEL and the RISC-V core independently, both with a 20ns clock cycle timing constraint, and then combining the resulting designs in a top level synthesis. For the ASIC designs, the test harness was not used, although a 20ns cycle time was still targeted. For final extraction of power information, we used Synopsys PrimeTime PX.

	ASIC	CESEL	RISC-V
AES-CTR	7.07	147.5	9036
SHA-256	-	3279	12630
ChaCha-20	15.3	302.4	2021
Frodo LWE	-	108224	109502
Curve25519	-	-	40454

Table 1: Energy consumption in nJ for different ciphers.

Results are summarized in table 1.

8 Conclusion and Future Work

In this project we have investigated the efficiency and flexibility of the CESEL architecture by implementing several key ciphers and comparing equivalent implementations in both hardware and software. We find that CESEL can provide significant acceleration to all ciphers we investigated, providing between a 1x-60x improvement in total energy consumption depending on the cipher (Table 1). Additionally, we find that CESEL uses between 10x-20x more energy than fixed function hardware.

For future work, we have three possible improvements we would like to investigate. First, we would like to complete the ciphers we did not finish and implement as many more ciphers as possible to get a better comparison.

Second, we found that the process of translating ciphers into CESEL assembly can be very tedious and time consuming; each cipher took several weeks of work and very little of it could be reused in other ciphers. As a result, a possible improvement would be to write a compiler that could automatically take a high level description of a cipher and convert it into CESEL assembly.

Third, we found several cases where we wish that CESEL had additional support for operations that were not in the underlying architecture. For example, an instruction that could reduce across all 32 lanes would be useful for things like matrix multiplication. As another example, allowing some lanes to be "masked" (i.e. non-active) for a short period would simplify writing algorithms that operate on only a small amount of data at a time. However, it is important that we balance adding specialized hardware for specific algorithms with the need to be generic.

References

[1] J. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from lwe. In *Proceedings of the 2016 ACM SIGSAC Conference on Com-*

puter and Communications Security, pages 1006–1018. ACM, 2016.

[2] T. Chou. Sandy2x: New curve25519 speed records. In *International Conference on Selected Areas in Cryptography*, pages 145–160. Springer, 2015.

[3] F. Denis. Libsodium: A modern and easy-to-use crypto library. `https://github.com/jedisct1/libsodium`, 2014–2017.

[4] D. E. Standard et al. Federal information processing standards publication 46. *National Bureau of Standards, US Department of Commerce*, 1977.