CASCON | EVOKE
NOV 22-25, 2021

Exploring Poly1305 and ChaCha20 Optimization

#CXEtechconf

24 November 2021

Introduction

During summer 2021, Steven Gonder and Pesara Amarasekera were hired as research assistants, translating the Internet Engineering Task Force (IETF) specification for AEAD_CHACHA20_POLY1305 into the Coconut domain-specific language. The project is a step towards eventually generating optimized, scheduled Go assembly code suitable for cryptographic purposes. Special thanks to Akshay Kumar Arumugasamy for helping port IETF test cases and to Dr. Christopher Anand for guiding and supervising the project.

Cryptographic Background

- The ChaCha20 cipher is a high-speed cipher, it is considerably faster than AES (the current standard in encryption) in software only implementations, thus it provides better performance in machines that lack dedicated AES hardware.
- Poly1305 is a high-speed authentication code that can be combined with ChaCha20.
- ChaCha20 is a stream cipher that fulfills the encryption function of an Authenticated Encryption with Associated Data (AEAD) algorithm, while Poly1305 is signing algorithm for generating the MAC (Message Authentication Code).
- We are interested in both the components (ChaCha20 and Poly1305) as well as the combination of them (AEAD) in our research.

Coconut

Anand, Christopher & Kahl, Wolfram. (2009). An Optimized Cell BE Special Function Library Generated by Coconut. Computers, IEEE Transactions on. 58. 1126-1138. 10.1109/TC.2008.223.

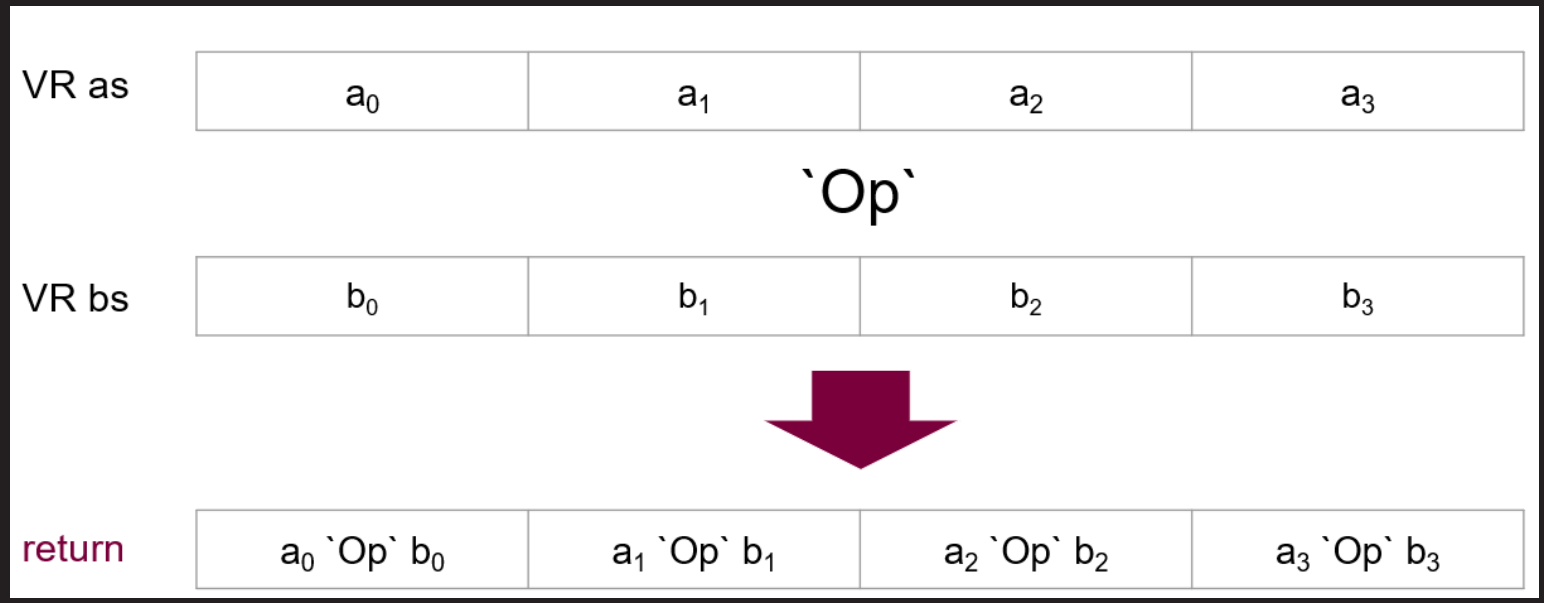
- Coconut(COde CONstructing User Tool) , a tool for developing high-assurance, high-performance kernels for scientific computing, contains an extensible domain-specific language (DSL) embedded in Haskell.
- Coconut is a platform for experimenting with novel ideas in reliable and high-performance code generation.
- Using Coconut, experts in high-performance numerical software can implement mathematical functions in a processor-specific way, meeting the following goals:
  1. The language supports safe development practices and is familiar to mathematicians.
  2. Common code construction tasks are simplified.
  3. The user is insulated from details of the target language not relevant to their task.
  4. The environment supports exploration and rapid prototyping.
  5. Alternatives for instruction selection can be provided by the user.
  6. Impact of program changes on instruction scheduling/pipelining is visible.

Our current project

- We build off a paper by Gadriwala, Anand, et al. ”Accelerating Poly1305 cryptographic message authentication on the z14”
- The paper can be found (behind a paywall) at https://dl.acm.org/doi/10.5555/3172795.3172802
- The authors note “Coconut has not been used on integer vector code, so work needs to be done to apply it in this new domain”
- Our work as research assistants over the summer supports this goal
- The focus of this presentation will be on applying SIMD to ChaCha20 in Coconut

Coconut, z/Architecture SIMD

- 128-bit vector registers, 32-bit words $a_{0..3}$, $b_{0..3}$



Four ChaCha20 Quarter Rounds at Once using Coconut SIMD

IETF Specification

2.1. The ChaCha Quarter Round

The basic operation of the ChaCha algorithm is the quarter round. It operates on four 32-bit unsigned integers, denoted a, b, c, and d. The operation is as follows (in C-like notation):

```
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= a; d <<<= 8;
c += d; b ^= c; b <<<= 7;
```


Where "+" denotes integer addition modulo 2^32, "^" denotes a bitwise Exclusive OR (XOR), and "<<< n" denotes an n-bit left roll (towards the high bits).

Coconut

```
quarterRoundOP1 (as, bs, cs, ds) =
  let
    as'  = va 2 as bs
    ds'  = vxor ds as'
    ds'' = verll0 2 ds' 16
  in (as', bs, cs, ds'')
```

va 2 = Vector Add (4x 32-bit)
vxor = Vector Xor (128-bit)
verll0 2 = Vector Element Rotate Logical Left (4x 32-bit)

where as, bs, cs, ds are 128-bit VRs

ChaCha20 Block Function as SIMD using 128-bit Rotation

IETF Specification

2.2. A Quarter Round on the ChaCha State

The ChaCha state does not have four integer numbers: it has 16. So the quarter-round operation works on only four of them -- hence the name. Each quarter round operates on four predetermined numbers in the ChaCha state. We will denote by QUARTERROUND(x, y, z, w) a quarter-round operation on the numbers at indices x, y, z, and w of the ChaCha state when viewed as a vector. For example, if we apply QUARTERROUND(1, 5, 9, 13) to a state, this means running the quarter-round operation on the elements marked with an asterisk, while leaving the others alone:

```
 0  *a   2   3
 4  *b   6   7
 8  *c  10  11
12  *d  14  15
```

ChaCha20 runs 20 rounds, alternating between "column rounds" and "diagonal rounds". Each round consists of four quarter-rounds, and they are run as follows. Quarter rounds 1-4 are part of a "column" round, while 5-8 are part of a "diagonal" round:

```
QUARTERROUND(0, 4, 8, 12)
QUARTERROUND(1, 5, 9, 13)
QUARTERROUND(2, 6, 10, 14)
QUARTERROUND(3, 7, 11, 15)
QUARTERROUND(0, 5, 10, 15)
QUARTERROUND(1, 6, 11, 12)
QUARTERROUND(2, 7, 8, 13)
QUARTERROUND(3, 4, 9, 14)
```

Coconut

```
inner_block state =   roundDiagonal . roundColumn
                    . roundDiagonal . roundColumn
                    . roundDiagonal . roundColumn
                    . roundDiagonal . roundColumn
                    . roundDiagonal . roundColumn
                    . roundDiagonal . roundColumn
                    . roundDiagonal . roundColumn
                    . roundDiagonal . roundColumn
                    . roundDiagonal . roundColumn
                    . roundDiagonal . roundColumn $ state

roundColumn :: forall n. ZType n => (VR n, VR n, VR n, VR n) -> (VR n, VR n, VR n, VR n)
roundColumn = quarterRounds

roundDiagonal :: forall n. ZType n => (VR n, VR n, VR n, VR n) -> (VR n, VR n, VR n, VR n)
roundDiagonal (row0, row1, row2, row3) = let

  (row0', row1', row2', row3') = quarterRounds (
                                 row0,                -- [ i0,  i1,  i2,  i3]
                                 verll128_rot32 row1, -- [ i5,  i6,  i7,  i4]
                                 verll128_rot64 row2, -- [i10, i11,  i8,  i9]
                                 verll128_rot96 row3  -- [i15, i12, i13, i14]
                               )
  in (
    row0',                -- [ i0,  i1,  i2,  i3]
    verll128_rot96 row1', -- [ i4,  i5,  i6,  i7]
    verll128_rot64 row2', -- [ i8,  i9, i10, i11]
    verll128_rot32 row3'  -- [i12, i13, i14, i15]
  )
```

ChaCha20 Block Function - Treat Diagonal Rounds as Column Rounds in SIMD

ChaCha20 runs 20 rounds, alternating between "column rounds" and "diagonal rounds". Each round consists of four quarter-rounds, and they are run as follows. Quarter rounds 1-4 are part of a "column" round, while 5-8 are part of a "diagonal" round:

```
QUARTERROUND(0, 4, 8, 12)
QUARTERROUND(1, 5, 9, 13)
QUARTERROUND(2, 6, 10, 14)
QUARTERROUND(3, 7, 11, 15)
QUARTERROUND(0, 5, 10, 15)
QUARTERROUND(1, 6, 11, 12)
QUARTERROUND(2, 7, 8, 13)
QUARTERROUND(3, 4, 9, 14)
```

CR

i0	i1	i2	i3
i4	i5	i6	i7
i8	i9	i10	i11
i12	i13	i14	i15

Op(s) = 128-bit vector rotate

DR

i0	i1	i2	i3
i5	i6	i7	i4
i10	i11	i8	i9
i15	i12	i13	i14

The indices during a column round (CR) or diagonal round (DR) are independent, allowing for efficient SIMD.

Rotate the second, third, and fourth rows so that a diagonal round can be treated as a column round. When finished, rotate them back.

Testing (IETF / Custom Unit Tests)

The testing consisted of unit and property tests.

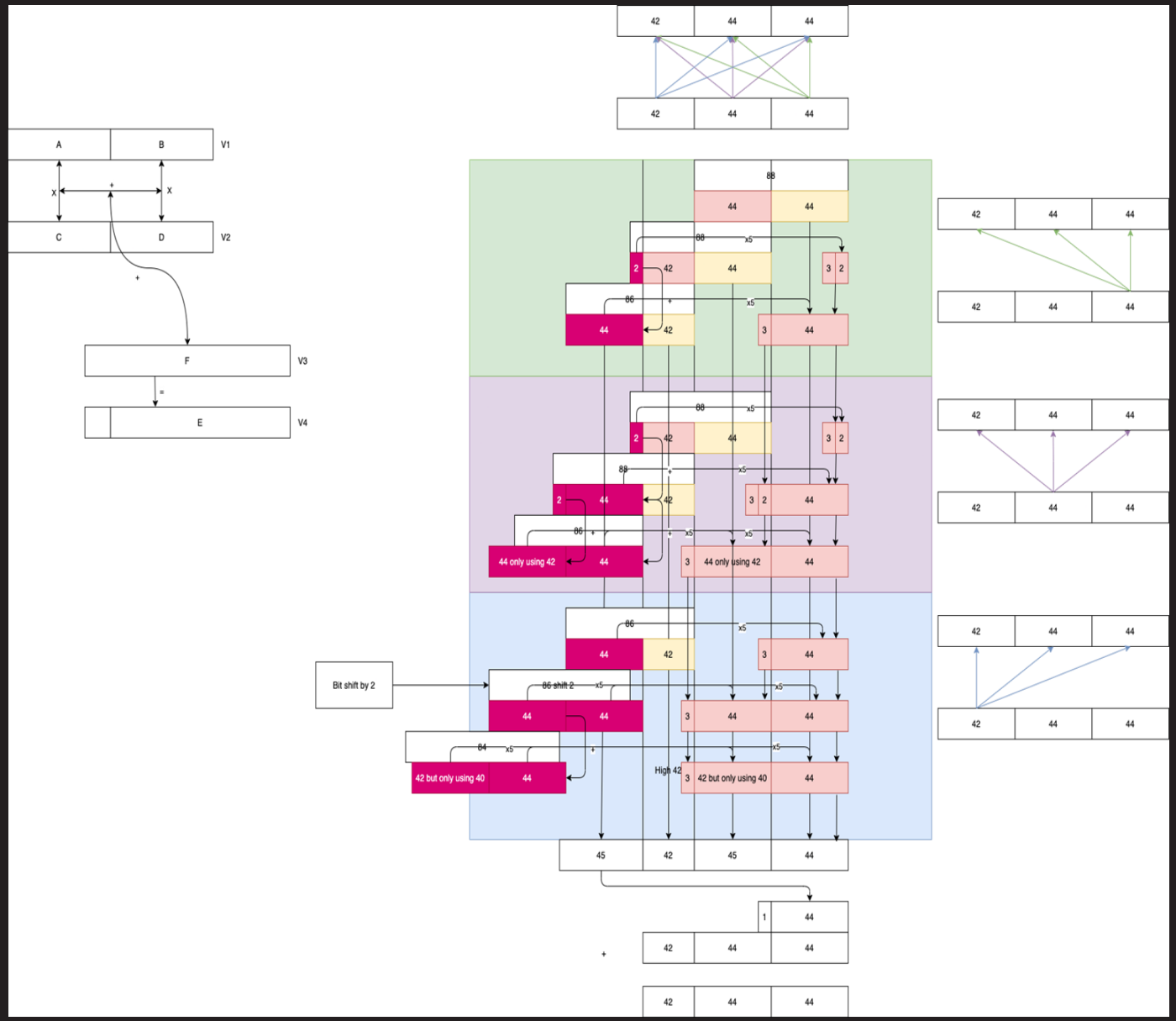
The IETF specification provides certain edge cases and special conditions for the ChaCha20 Block Functions, ChaCha20 Encryption, Poly1305 Message Authentication Code, Poly1305 Key Generation Using ChaCha20 and ChaCha20-Poly1305 AEAD Decryption.

There were unit tests and property tests that were created to test specific parts of our COCONUT implementation that is outside of the specification.

For purposes of testing, debugging tools for specific parts of the implementation were created in order to understand and analyze the error

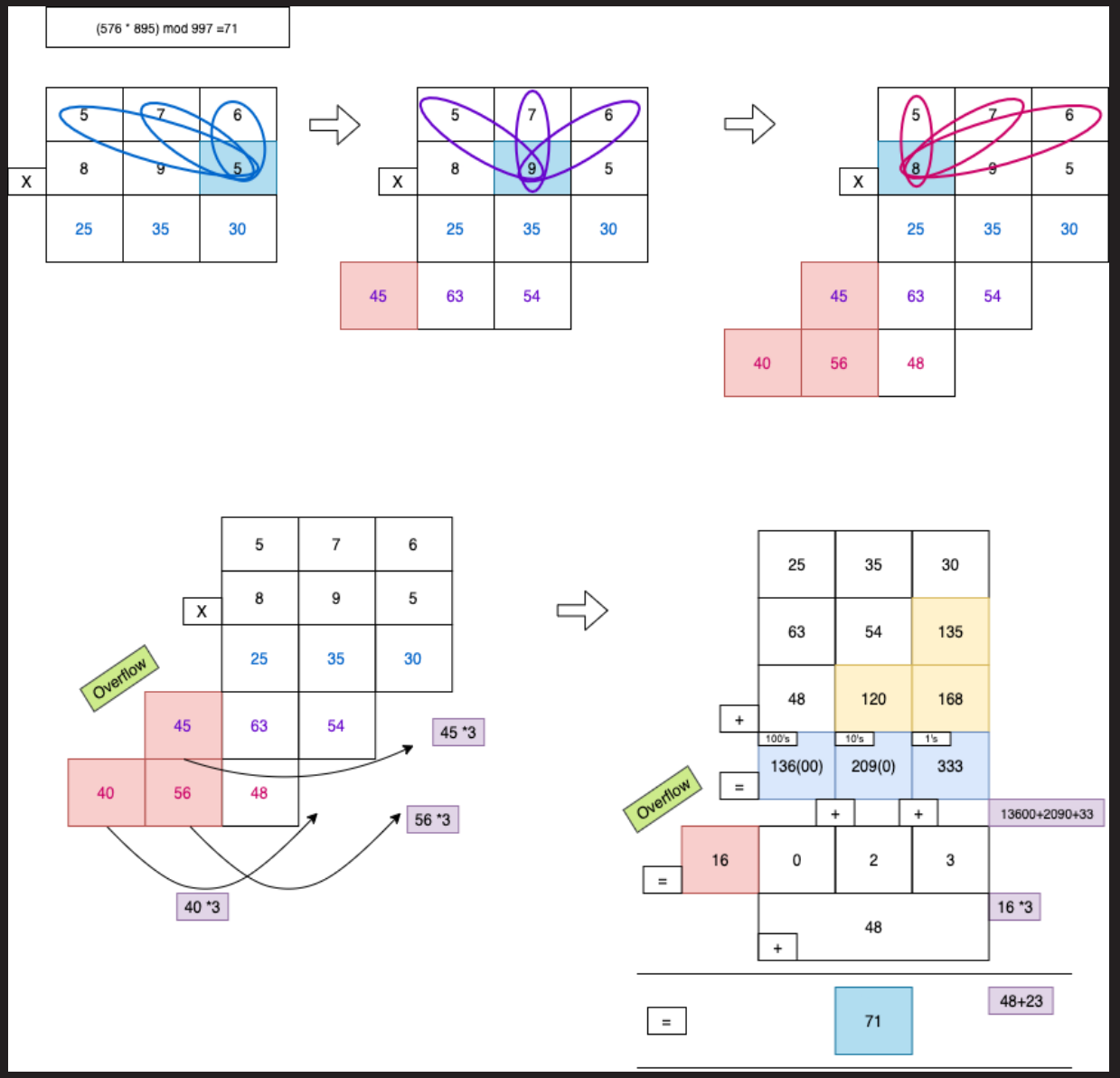
Schoolbook Multiplication

As can be seen regular schoolbook multiplication, though easier to understand requires more operations to compute and is inefficient.



Comba Multiplication

Comba multiplication provides an alternative more effective way of multiplying with less computations on overflows.



Property Testing

The property testing was done with the use of the Haskell QuickCheck framework.

The property tests are mostly extensions of the unit tests or algorithms (we test a certain algorithm with a reference implementation written in haskell against a pure COCONUT implementation that can be used to generate code).

The generators used in the property tests generate values with-in a specified set of values between edge cases which are covered by unit tests.

The property tests were used to test integrating two or more key Procedures that can be sequenced in the algorithm such as the ones used in generating the cipher text and deciphering it.

Next Steps

Find different limb sizes that can be used.

Find and prove of the maximum number of overflows that need to be calculated for a given set of limb sizes.

Write Property tests for AEAD Encryption with longer strings (currently we rely on the IETF test vectors).

Measure Performance of the current 42,44,44 bit limb sizes and compare performance with other limb sizes. Use this to find an optimal way(s) to limbify a 130 bit multiplication.

Specification

- We are using IETF's most recent Request For Comment for AEAD_CHACHA20_POLY1305, that being RFC 8439
- RFC 8439 can be found at https://www.rfc-editor.org/rfc/rfc8439.html
- RFC 8439 obsoletes RFC 7539 and is the latest version
- Errata have been checked for potentially breaking errors, none of which were found
- We additionally reference the preceding paper by Gadriwala, Anand, et al. for comba multiplication and z/Architecture constraints
- Some changes have been made, as seen on the Divergence from Specification slides

Coconut Implementation

- Code takes advantage of SIMD parallelism
- 130-bit messages are limbified into 44/44/42 bits (currently), same as the acceleration paper
- The IETF document primarily operates on little-endian ordered bytestreams, whereas our implementation only uses little-endian ordering within the memory region
- All values operated on in Coconut are big-endian

z/Architecture Instructions Used

- Vector Multiply Sum Logical (vmsl)
- Vector Permute (vperm or shufb)
- Vector Add (va)
- Vector Shift Right Logical / Byte (vsrl/vsrlb)
- Vector Shift Left Byte (vslb)
- Vector Element Rotate Logical Left (verll0)
- Vector Xor (vxor)
- Vector Or (vor)
- Vector And (vand)
- Vector Subtract (vs)
- Vector Subtract Compute Borrow Indication (vscbi)
- Vector Merge Low (vmrl)
- Vector Merge High (vmhl)
- Vector Compare Equal (vceq)
- Vector Select (vsel or selb)

These instructions are intended to be run in constant time

Scheduling

At the moment scheduling is postponed as most members are unavailable to work on this at the moment.

At the moment we are able to encrypt one block (512 bits or 64 bytes) at a time sequentially by running the AEAD algorithm.

Code Snippets/QuarterRound

- The following are a few examples of the code that were refactored (significantly)

```
permutation =     selb constZero byte1P    (vrep 0 15 (vceq 0 mod16 constOne))
                  `vor` selb constZero byte2P    (vrep 0 15 (vceq 0 mod16 constTwo))
                  `vor` selb constZero byte3P    (vrep 0 15 (vceq 0 mod16 constThree))
                  `vor` selb constZero byte4P    (vrep 0 15 (vceq 0 mod16 constFour))
                  `vor` selb constZero byte5P    (vrep 0 15 (vceq 0 mod16 constFive))
                  `vor` selb constZero byte6P    (vrep 0 15 (vceq 0 mod16 constSix))
                  `vor` selb constZero byte7P    (vrep 0 15 (vceq 0 mod16 constSeven))
                  `vor` selb constZero byte8P    (vrep 0 15 (vceq 0 mod16 constEight))
                  `vor` selb constZero byte9P    (vrep 0 15 (vceq 0 mod16 constNine))
                  `vor` selb constZero byte10P   (vrep 0 15 (vceq 0 mod16 constTen))
                  `vor` selb constZero byte11P   (vrep 0 15 (vceq 0 mod16 constEleven))
                  `vor` selb constZero byte12P   (vrep 0 15 (vceq 0 mod16 constTwelve))
                  `vor` selb constZero byte13P   (vrep 0 15 (vceq 0 mod16 constThirteen))
                  `vor` selb constZero byte14P   (vrep 0 15 (vceq 0 mod16 constFourteen))
                  `vor` selb constZero byte15P   (vrep 0 15 (vceq 0 mod16 constFifteen))
                  `vor` selb constZero byte16P   (vrep 0 15 (vceq 0 mod16 constZero))
```

```
+       mod16ToPlace = shufb (verll0 0 (vand constFifteen len) 3) constZero
constVSLBVSRLBPerm
+       vector16ToPlace = shufb (verll0 0 constSixteen 3) constZero constVSLBVSRLBPerm

+       constantRes = vs 0 vector16ToPlace mod16ToPlace

+

+       zeroVR vLast =

+         let

+           res = vslb (vsrlb constantRes vLast) constantRes

+         in

+           res
```

CASCON | EVOKE

NOV 22-25, 2021

Thank you!

Steven Gonder
Computer Science IV
gondes1@mcmaster.ca

Akshay Kumar Arumugasamy
MSc Computer Science I
arumua3@mcmaster.ca

Pesara Amerasekera
Software Engineering IV
amarasep@mcmaster.ca

Supervised by,
Dr. Christopher Anand
Associate Professor,
McMaster University
anandc@mcmaster.ca

Bill O'Farrell
IBM Canada Lab,
Markham
billo@ca.ibm.com