Implementing an Elliptic Curve

or, How to Write Ed25519 in Go

George Tankersley
@gtank__

Preliminaries

Clone this repo

https://github.com/gtank/defcon25_crypto_village

So what’s in there?

The Ed25519 software is available as the crypto_sign/ed25519 subdirectory of the SUPERCOP benchmarking tool, starting in version 20110629. This software will also be integrated into the next release of the Networking and Cryptography library (NaCl).

The Ed25519 software consists of three separate implementations, all providing the same interface:

- amd64-51-30k. Assembly-language implementation for the amd64 architecture, using radix 2^51 and a 30KB precomputed table.
- amd64-64-24k. Assembly-language implementation for the amd64 architecture, using radix 2^64 and a 24KB precomputed table.
- ref. Slow but relatively simple and portable C implementation.

Both SUPERCOP and NaCl automatically select the fastest implementation on each computer.

https://github.com/gtank/defcon25_crypto_village

So what’s in there?

x/crypto/ed25519 and x/crypto/curve25519

The Go (extended) standard library implementations.

https://github.com/gtank/defcon25_crypto_village

So what's in there?

curve25519-dalek

A low-level cryptographic library for point, group, field, and scalar operations on a curve isomorphic to the twisted Edwards curve defined by $-x^2+y^2 = 1 - (121665/121666)x^2y^2$ over $GF(2^{255} - 19)$.

https://github.com/gtank/defcon25_crypto_village

So what’s in there?

Also, I wrote one! It’s not quite done yet.

https://github.com/gtank/defcon25_crypto_village

But wait, there’s more!

There **are** more:

- nacl
- tweetnacl
- 2 (?) variants in tor
- a Java ref10 port that i2p uses
- curve25519-donna
- ...

All of these codebases look very similar

Steps to get ed25519 in your vaguely C-like language:

1. *Download ref10*
2. *Copy + paste*
3. *Fix linter errors*

Theorem: understand one and you can figure out the rest

Corollary: understand pieces of many, then combine

All of these codebases look very similar

The code generally breaks down along two categories:

1. **Field math**. The implementation of basic arithmetic operations (addition, subtraction, multiplication, squaring, inversion, and reduction) on integers in GF(2^255-19) and routines for manipulating field elements.
2. **Group logic**. The actual elliptic curve part, including point addition, doubling, scalar multiplication, and a variety of coordinate representations and conversion routines.

Field

The basics: GF(2^255 - 19)

“**G**alois **F**ield”, means “integers modulo the prime 2^255 - 19”

You could also say “the finite field of characteristic 2^255 - 19”

We are implementing multi-precision arithmetic over GF(2^255-19).

Some things to keep in mind:

- These are 255-bit integers
- We don’t want to use a generic bignum
- Aiming for both **constant-time** execution and high **performance**

Why 2^255 - 19?

I chose my prime 2^255 − 19 according to the following criteria: primes as close as possible to a power of 2 save time in field operations (as in, e.g, [9]), with no effect on (conjectured) security level; primes slightly below 32k bits, for some k, allow public keys to be easily transmitted in 32-bit words, with no serious concerns regarding wasted space; k = 8 provides a comfortable security level. I considered the primes 2^255 + 95, 2^255 − 19, 2^255 − 31, 2^254 + 79, 2^253 + 51, and 2^253 + 39, and selected 2^255 − 19 because 19 is smaller than 31, 39, 51, 79, 95.

(Bernstein, "Curve25519: new Diffie-Hellman speed records")

Why 2^255 - 19?

We like prime fields these days (as opposed to binary or optimal extension fields)

We like characteristic primes near powers of two.

Specifically, primes of the form ***2^k - c*** are called Crandall primes. When ***c*** is small relative to the size of a machine word, this shape allows you to limit carry propagation during multiplications.

Plus something about cramming public keys into 32-bit words. It was 2006.

IMPORTANT POINT: the choice of prime field and representation are usually driven by clever optimizations. They can be **absurdly** platform-specific.

Representing GF(2^255 - 19)

How do we represent numbers so much larger than native integers? We choose an efficient **radix** and decompose the numbers into multiple **limbs**.

So, how can you pack 255 bits?

- On 32-bit, use radix 2^32: 8 limbs * 32 bits = 256 bits
- On 64-bit, use radix 2^64: 4 limbs * 64 bits = 256 bits

These are “uniform” and “saturated” representations, because each limb is the same size and we’re using all of the bits available in each word.

Representing GF(2^255 - 19)

This choice is absurdly platform-specific:

Why split 255-bit integers into ten 26-bit pieces, rather than nine 29-bit pieces or eight 32-bit pieces? ***Answer: The coefficients of a polynomial product do not fit into the Pentium M's fp registers if pieces are too large.*** *The cost of handling larger coefficients outweighs the savings of handling fewer coefficients. The overall time for 29-bit pieces is sufficiently competitive to warrant further investigation, but so far I haven't been able to save time this way. I'm sure that 32-bit pieces, the most common choice in the literature, are a bad idea.* ***Of course, the same question must be revisited for each CPU.*** (Bernstein)

Representing GF(2^255 - 19)

Modern implementations use **unsaturated** representations, where the number of bits we “care” about is less than the size of the word. The difference is called **headspace**.

Some implementations also use **non-uniform** limb schedules.

So, how can you pack 255 bits?

Representing GF(2^255 - 19)

On 32-bit, use radix 2^25.5: 10 limbs * 25.5 bits = 255

“25.5” means a balanced alternating limb schedule of 25/26/25/26/… bits

*Given that there are 10 pieces, why use radix 2^25.5 rather than, e.g., radix 2^25 or radix 2^26 ? Answer: My ring R contains 2^255 * x^10 − 19, which represents 0 in Z/(2^255 − 19). I will reduce polynomial products modulo 2^255 * x^10 − 19 to eliminate the coefficients of x^10 , x^11 , etc. With radix 2^25 , the coefficient of x^10 could not be eliminated. With radix 2^26 , coefficients would have to be multiplied by 2^5 · 19 rather than just 19, and **the results would not fit into an fp register**.* (Bernstein)

Look, it was 2006.

Representing GF(2^255 - 19)

What we actually care about now is 64-bit (and usually amd64)

Use 5 limbs in uniform radix 2^51: 5 limbs * 51 bits = 255 bits

In practice, this bound is loose.

Unsaturated wins here because we can do less carry propagation by letting the limbs grow beyond 51 bits between operations.


YOU ARE LEAVING
THE AMERICAN SECTOR
ВЫ ВЫЕЗЖАЕТЕ ИЗ
АМЕРИКАНСКОГО СЕКТОРА
VOUS SORTEZ
DU SECTEUR AMÉRICAIN
SIE VERLASSEN DEN AMERIKANISCHEN SEKTOR
US ARMY

CHECKPOINT

Where’s the code already?!

[gtank/internal/radix51]

[supercop/amd64-51-30k]

Field Element type

Go:

```go
type FieldElement [5]uint64
```

C:

```c
typedef struct {
    unsigned long long v[5];
} fe25519;
```

```
// FieldElement represents an element of the field
// GF(2^255-19). An element t represents the integer
// t[0] + t[1]*2^51 + t[2]*2^102 + t[3]*2^153 + t[4]*2^204.
```

Field Operations

`Addition`

`Subtraction`

`Multiplication`

`Squaring`

`Inversion`

`Reduction`

Field Addition (fe.go, fe25519_add.c)

```go
func FeAdd(out, a, b *FieldElement) {
    out[0] = a[0] + b[0]
    out[1] = a[1] + b[1]
    out[2] = a[2] + b[2]
    out[3] = a[3] + b[3]
    out[4] = a[4] + b[4]
}
```

```
// FeAdd sets out = a + b. Long sequences of additions without
// reduction that let coefficients grow larger than 54 bits would
// be a problem. “Do not have such sequences of additions”
```

Field Operations

~~`Addition`~~

`Subtraction`

`Multiplication`

`Squaring`

`Inversion`

`Reduction`

Field Subtraction (fe.go, fe25519_sub.c) (signed)

```
// FeSub sets out = a - b
func FeSub(out, a, b *FieldElement) {
	var t FieldElement
	t = *b

	// Reduce each limb below 2^51
	t[1] += t[0] >> 51
	t[0] = t[0] & maskLow51Bits
	t[2] += t[1] >> 51
	t[1] = t[1] & maskLow51Bits
	t[3] += t[2] >> 51
	t[2] = t[2] & maskLow51Bits
	t[4] += t[3] >> 51
	t[3] = t[3] & maskLow51Bits
	t[0] += (t[4] >> 51) * 19
	t[4] = t[4] & maskLow51Bits

// This is slightly more complicated.
// Because we use unsigned coefficients, we
// first add a multiple of p and then
// subtract.
	out[0] = (a[0] + 0xFFFFFFFFFFFDA) - t[0]
	out[1] = (a[1] + 0xFFFFFFFFFFFFE) - t[1]
	out[2] = (a[2] + 0xFFFFFFFFFFFFE) - t[2]
	out[3] = (a[3] + 0xFFFFFFFFFFFFE) - t[3]
	out[4] = (a[4] + 0xFFFFFFFFFFFFE) - t[4]
}
```

At this point, it's going to be hard to fit these on slides:
https://github.com/gtank/defcon25_crypto_village

Field Operations

~~`Addition`~~

~~`Subtraction`~~

`Multiplication`

`Squaring`

`Inversion`

`Reduction`

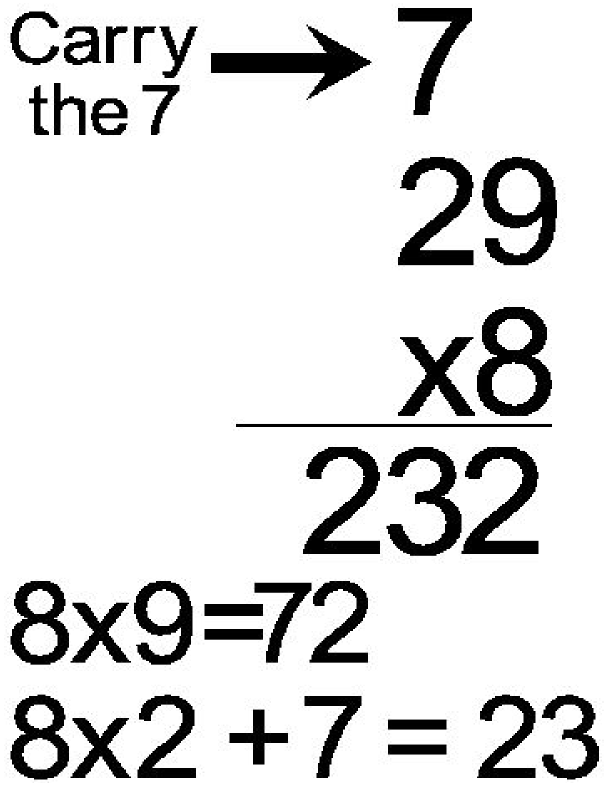
Field Multiplication (fe_mul*, fe25519_mul.s)

“Schoolbook” multiplication

5 limbs takes 25 multiplications

64 bits x 64 bits => 128 bits

“multiply-reduce”



Impossible to fit these on slides. Go code:

gtank/internal/radix51/fe_mul.go

Multiply-reduce?

Theorem: Given a number in base 2, it is easy to reduce it by a number close to a power of 2.

Generally, if $n = 2^k - c$, then $2^k \equiv c \pmod{n}$.

Let $n = 7 = 2^3-1$, then $2^3 \equiv 1 \pmod{n}$
To reduce x mod n, first convert x to base 2^3 by grouping:
If x = (10010), then

```
x’ = (10) * 2^3 + (010) (mod n)
x’ = (10) * 1  + (010) (mod n)
x’ = (10) + (10) (mod n)
Which is the correct answer: 18 ≡ 4 (mod 7)

h/t to hdevalence. Full explain & better example on his blog.
```

Field Operations

~~Addition~~

~~Subtraction~~

~~Multiplication~~

Squaring

Inversion

Reduction

Field Squaring (fe_square.go, fe25519_square.s)

Squaring needs only 15 mul instructions. Some inputs are multiplied by 2; this is combined with multiplication by 19 where possible. The coefficient reduction after squaring is the same as for multiplication. (Bernstein, Duif, Lange, Schwabe, Yang, “High-speed high-security signatures”)

Very similar to multiplication. Not very interesting.

The thing to know is that squaring is noticeably cheaper than multiplication. When implementing higher-level operations, you should use FeSquare(x) instead of FeMul(x, x).

Field Operations

~~Addition~~

~~Subtraction~~

~~Multiplication~~

~~Squaring~~

Inversion

Reduction

Field Inversion (fe.go, fe25519_invert.c)

We implement inversion based on Fermat's little theorem:

$$a^p \equiv a \pmod p$$
$$a^{p-1} \equiv 1 \pmod p$$
$$a * a^{p-2} \equiv 1 \pmod p$$

So inversion mod p is equivalent to raising to the power p - 2.
If p = 2^255 - 19, then p - 2 = 2^255 - 21

Code: FeInvert fe.go#L130

Field Operations

~~Addition~~

~~Subtraction~~

~~Multiplication~~

~~Squaring~~

~~Inversion~~

Reduction

Field Reduction (fe.go, fe25519_freeze.s)

Basic idea is to reduce each limb below 2^51, propagating carries until you reach the top limb carry, which you multiply by 19 and wrap into the bottom limb.

```
// TODO Document why this works.
// It's the elaborate comment about r = h-pq etc etc.
```

Code: FeReduce fe.go#L130

Elaborate comment (about 32-bit repr): supercop/ref10/fe_tobytes.c#L10
(general idea is reasoning about progressively tighter bounds)

Field Operations

~~`Addition`~~

~~`Subtraction`~~

~~`Multiplication`~~

~~`Squaring`~~

~~`Inversion`~~

~~`Reduction`~~


Security Check Point
Ticketed Passengers Only
Beyond This Point
free Wi-Fi

CHECKPOINT

Group

What’s an elliptic curve?

NO

What’s an Ed25519?

$$-x^2+y^2 = 1 - 121665/121666\ x^2y^2 \text{ over } GF(2^{255} - 19)$$

Ed25519 is a **twisted Edwards curve**.

This post gives a great overview of what exactly that means:

https://moderncrypto.org/mail-archive/curves/2016/000806.html (Mike Hamburg, “Climbing the elliptic learning curve”)

Elliptic curves as software interface

```
type Curve interface {
        // IsOnCurve reports whether the given (x,y) lies on the curve.
        IsOnCurve(x, y *big.Int) bool
        // Add returns the sum of (x1,y1) and (x2,y2)
        Add(x1, y1, x2, y2 *big.Int) (x, y *big.Int)
        // Double returns 2*(x,y)
        Double(x1, y1 *big.Int) (x, y *big.Int)
        // ScalarMult returns k*(Bx,By) where k is in big-endian form.
        ScalarMult(x1, y1 *big.Int, k []byte) (x, y *big.Int)
        // ScalarBaseMult returns k*G, where G is the base point of the group
        // and k is an integer in big-endian form.
        ScalarBaseMult(k []byte) (x, y *big.Int)
}
```

https://golang.org/pkg/crypto/elliptic/#Curve

Elliptic curves as math

For the purposes of this talk, we are dealing with **explicit formulas**.

Edwards curves give us **complete** formulas without exceptional failure cases.

This makes implementation easy and “safe”

Explicit Formulas Database:

https://www.hyperelliptic.org/EFD/g1p/auto-twisted.html

Representing curve points

Points are structs made of coordinates.

Coordinates are explicitly-named field elements.

There are multiple coordinate systems in use.

```
type ProjectiveGroupElement struct {
	X, Y, Z field.FieldElement
}

type ExtendedGroupElement struct {
	X, Y, Z, T field.FieldElement
}
```

Affine coordinates

Traditional (x, y) points

The elliptic.Curve interface deals exclusively in affine big.Int coordinates.

```
// We don't actually use this.
type AffineGroupElement struct {
    X, Y field.FieldElement
}
```

Projective coordinates (EFD)

```
(x, y) -> (X:Y:Z)

Satisfying

    x = X/Z
    y = Y/Z
```

Affine to projective: set Z = 1

Projective to affine: multiply by 1/Z

```
// Most implementations use this
// for improved doubling
// efficiency. But...
type ProjectiveGroupElement struct {
    X, Y, Z field.FieldElement
}
```

Extended coordinates (EFD)

```
(x, y) -> (X:Y:Z:T)

Satisfying

    x = X/Z
    y = Y/Z
    x * y = T/Z
```

Affine to extended:

```
    Z=1, T=xy
```

Extended to affine:

```
    drop T, clear Z
```

```
// Used for almost everything else
type ExtendedGroupElement struct {
    X, Y, Z, T field.FieldElement
}
```

Hisil-Wong-Carter-Dawson, "Twisted Edwards Curves Revisited"

“Completed” coordinates (impl)

(x, y) -> (X:Z)(Y:T)

Satisfying

x = X/Z

y = Y/T

Used in **mixed-coordinate** addition and doubling chains.

```
// I got nothing. They work!
type CompletedGroupElement struct {
     X, Y, Z, T FieldElement
}

typedef struct {
 fe25519 x;
 fe25519 z;
 fe25519 y;
 fe25519 t;
} ge25519_p1p1;
```

Curve interface uses affine big.Int pairs

```
type Curve interface {
        // IsOnCurve reports whether the given (x,y) lies on the curve.
        IsOnCurve(x, y *big.Int) bool
        // Add returns the sum of (x1,y1) and (x2,y2)
        Add(x1, y1, x2, y2 *big.Int) (x, y *big.Int)
        // Double returns 2*(x,y)
        Double(x1, y1 *big.Int) (x, y *big.Int)
        // ScalarMult returns k*(Bx,By) where k is in big-endian form.
        ScalarMult(x1, y1 *big.Int, k []byte) (x, y *big.Int)
        // ScalarBaseMult returns k*G, where G is the base point of the group
        // and k is an integer in big-endian form.
        ScalarBaseMult(k []byte) (x, y *big.Int)
}
```

https://golang.org/pkg/crypto/elliptic/#Curve

big.Int <> FieldElement

```go
// Bytes returns the absolute value of x as a big-endian byte slice.
func (x *Int) Bytes() []byte {
	buf := make([]byte, len(x.abs)*_S)
	return buf[x.abs.bytes(buf):]
}
// SetBytes interprets buf as the bytes of a big-endian unsigned
// integer, sets z to that value, and returns z.
func (z *Int) SetBytes(buf []byte) *Int {
	z.abs = z.abs.setBytes(buf)
	z.neg = false
	return z
}
```

big.Int <> FieldElement

```
// Bytes returns the absolute value of x as a big-endian byte slice.

// SetBytes interprets buf as the bytes of a big-endian unsigned
// integer, sets z to that value, and returns z.
```

Problem: field element packing is always little-endian

big.Int <> FieldElement

big.Int has an escape hatch:

```
// Bits provides raw (unchecked but fast) access to x by returning its
// absolute value as a little-endian Word slice. The result and x share
// the same underlying array.
// Bits is intended to support implementation of missing low-level Int
// functionality outside this package; it should be avoided otherwise.
func (x *Int) Bits() []Word {
	return x.abs
}
```

Faster than Bytes(), and already little-endian!

big.Int <> FieldElement

Field element packing in C:

- fe25519_pack.c
- fe25519_unpack.c

Packing in Go is identical. Using Bits() saves us a slice reversal.

New in Go 1.9 (math/bits): we can map to big.Int generically!

- FeFromBig
- FeToBig

Curve interface operations

```
type Curve interface {
        // IsOnCurve reports whether the given (x,y) lies on the curve.
        IsOnCurve(x, y *big.Int) bool
        // Add returns the sum of (x1,y1) and (x2,y2)
        Add(x1, y1, x2, y2 *big.Int) (x, y *big.Int)
        // Double returns 2*(x,y)
        Double(x1, y1 *big.Int) (x, y *big.Int)
        // ScalarMult returns k*(Bx,By) where k is in big-endian form.
        ScalarMult(x1, y1 *big.Int, k []byte) (x, y *big.Int)
        // ScalarBaseMult returns k*G, where G is the base point of the group
        // and k is an integer in big-endian form.
        ScalarBaseMult(k []byte) (x, y *big.Int)
}
```

https://golang.org/pkg/crypto/elliptic/#Curve

Point-on-curve check (impl)

```
// -x^2 + y^2 - 1 - dx^2y^2 = 0 (mod p).
func (curve ed25519Curve) IsOnCurve(x, y *big.Int) bool {
	var feX, feY field.FieldElement
	field.FeFromBig(&feX, x)
	field.FeFromBig(&feY, y)

	var lh, y2, rh field.FieldElement
	field.FeSquare(&lh, &feX)            // x^2
	field.FeSquare(&y2, &feY)            // y^2
	field.FeMul(&rh, &lh, &y2)           // x^2*y^2
	field.FeMul(&rh, &rh, &group.D)      // d*x^2*y^2
	field.FeAdd(&rh, &rh, &field.FieldOne) // 1 + d*x^2*y^2
	field.FeNeg(&lh, &lh)                // -x^2
	field.FeAdd(&lh, &lh, &y2)           // -x^2 + y^2
	field.FeSub(&lh, &lh, &rh)           // -x^2 + y^2 - 1 - dx^2y^2
	field.FeReduce(&lh, &lh)             // mod p
	return field.FeEqual(&lh, &field.FieldZero)
}
```

Point addition (impl)

```go
// Add returns the sum of (x1, y1) and (x2, y2).
func (curve ed25519Curve) Add(x1, y1, x2, y2 *big.Int) (x, y *big.Int) {
	var p1, p2 group.ExtendedGroupElement
	p1.FromAffine(x1, y1)
	p2.FromAffine(x2, y2)
	return p2.Add(&p1, &p2).ToAffine()
}
```

But what does Add do? gtank/internal/group/ge.go#L74

Point doubling (impl 1) (impl 2)

```
// Double returns 2*(x,y).
func (curve ed25519Curve) Double(x1, y1 *big.Int) (x, y *big.Int) {
	var p group.ProjectiveGroupElement
	p.FromAffine(x1, y1)

	// Use the special-case DoubleZ1 here because we know Z will be 1.
	return p.DoubleZ1().ToAffine()
}
```

Specific to Go:

Two doubling formulas. Affine conversion makes the tradeoff less clear.

Arbitrary-point scalar multiplication (impl 1) (impl 2)

“Square-and-multiply” == “double-and-add”

The **why** is genuinely beyond our scope today.

Concept overview:

Bernstein. “curves, coordinates, and computations”

Deeper:

Joye, Yen. “The Montgomery Powering Ladder”

Even deeper:

Costello, Smith. “Montgomery Curves and their Arithmetic”

Base-point scalar multiplication (impl 1) (impl 2)

For any point known ahead of time, can precompute multiples to speed things up.

Usually, you only do this for the basepoint of the curve (think: key generation).

Adam Langley. “Faster curve25519 with precomputation.”

Bernstein, Duif et al. High-speed high-security signatures. Section 4.

This probably best explained in code by dalek: dalek/src/curve.rs#L917

CHECKPOINT

Moral of the story

Questions? We can stop here.

Bonus: Go performance tweaks

64 x 64 bit multiplications

We don’t have them! Go does not expose uint128.

Two answers:

1. Write assembly; amd64 provides 64-bit widening multipliers
2. Fight the inliner

Option 2 is a whole other talk.

64 x 64 bit multiplications (impl)

```go
import "unsafe"

// mul64x64 multiples two 64-bit numbers and adds them to two accumulators.
func mul64x64(lo, hi, a, b uint64) (ol uint64, oh uint64) {
	t1 := (a>>32)*(b&0xFFFFFFFF) + ((a & 0xFFFFFFFF) * (b & 0xFFFFFFFF) >> 32)
	t2 := (a&0xFFFFFFFF)*(b>>32) + (t1 & 0xFFFFFFFF)
	ol = (a * b) + lo
	cmp := ol < lo
	oh = hi + (a>>32)*(b>>32) + t1>>32 + t2>>32 +
		uint64(*(*byte)(unsafe.Pointer(&cmp)))
	return
}
```

Writing assembly 1

This is mostly what I’ve done. The implementations are in radix51/fe_mul_amd64.s and radix51/fe_square_amd64.s.

Things to note:

1. Go uses Plan9 assembly! Have fun finding docs.
2. The Go inliner won’t touch assembly functions. So you need to implement the entire field multiplication in asm, not just the 64->128 multiplies.
3. The build flag `noasm` exists.

Writing assembly 2

There are some good tools that help with writing and benchmarking Go assembly:

PeachPy is a tool for writing platform-agnostic assembly and generating output for your target platform. It supports goasm as an output mode. Damian Gryski wrote a tutorial on using it for Go: https://blog.gopheracademy.com/advent-2016/peachpy/

pprof has modes aimed at benchmarking and exploring compiler output.

Writing assembly 3

Go assembly can handle platform intrinsics, but doesn’t know about them.

You end up using BYTE literals, e.g.

```
BYTE $0xC5; BYTE $0xFD; BYTE $0xEF; BYTE $0xC0 // VPXOR ymm0, ymm0, ymm0
```

[A full AVX2 example]()

END