Notes on overflow calculation in `emulated`

Saravanan Vijayakumaran

April 21, 2024

1 Introduction

In the `emulated` crate, field arithmetic in a prime field $\mathbb{F}_n$ is performed using elements from another prime field $\mathbb{F}_p$. The latter is typically the scalar field of an elliptic curve group. The simulated field $\mathbb{F}_n$ is called the **non-native field**. Each non-native field element is represented using multiple field elements from $\mathbb{F}_p$, which are called **limbs**.

Suppose the non-native field is the prime field of order $n = 2^{255} - 19$. This is the base field of the ed25519 curve. If $p > 2^{64}$, then an element a of $\mathbb{F}_n$ can be represented using four limbs $a_0, a_1, a_2, a_3 \in \mathbb{F}_p$ as $a = \sum_{i=0}^{3} a_i 2^{64i}$ where each a_i is restricted to the range $\{0, 1, 2, \ldots, 2^{64} - 1\}$. In this case, the **limb width** is 64 bits. In case $p > 2^{85}$, we can also represent elements from $\mathbb{F}_n$ using three limbs each having limb width of 85 bits. We will use the ed25519 base field represented using four limbs as a running example in the rest of this document. We will use 64 as the limb width in some expresssions to make them less cumbersome.

Typically, p will be at least 2^{250}. The number $\lfloor \log_2 p \rfloor$ is called the **capacity** of $\mathbb{F}_p$. If c is the capacity of a prime field, then all integers in the range $\{0, 1, 2, \ldots, 2^c - 1\}$ can be represented as field elements. Integers having $c + 1$ or more bits in their binary representation may exceed p, and hence require a modular reduction modulo p before they can be represented in the field $\mathbb{F}_p$.

Arithmetic operations between non-native field elements can be represented as operations between their respective limbs followed by a reduction step. For example, suppose we want to add two field elements from the ed25519 base field using their limbed representations, given by $a = \sum_{i=0}^{3} a_i 2^{64i}$ and $b = \sum_{i=0}^{3} b_i 2^{64i}$.

The sum $a_i + b_i$ of their respective limbs will have a maximum possible bitwidth of 65 bits, as the sum is in the range $\{0, 1, \ldots, 2^{65} - 2\}$. The reduction step involves finding another set of limbs $c_0, c_1, c_2, c_3 \in \{0, 1, 2, \ldots, 2^{64} - 1\}$ such that

$$\sum_{i=0}^{3} (a_i + b_i) 2^{64i} - \sum_{i=0}^{3} c_i 2^{64i} = 0 \bmod n.$$

Checking the above equality involves accounting for carry bits and range checks. This makes the reduction step expensive in terms of R1CS constraints.

The `emulated` crate takes an approach (pioneered by the Gnark library) of deferring the reduction step as much as possible. This approach requires us to keep track of the current **overflow** in the limbs of a non-native field element. Denote the **maximum overflow** as the value

$$\texttt{maximum\_overflow} = \texttt{capacity} - \texttt{limb\_width} - 3,$$

where `capacity` is the capacity of the field $\mathbb{F}_p$. The reasoning behind this value will be explained below.

2 Addition

Consider the sum of the limbed representations of the ed25519 base field elements $a = \sum_{i=0}^{3} a_i 2^{64i}$ and $b = \sum_{i=0}^{3} b_i 2^{64i}$. We could avoid the reduction and simply represent the sum $a + b$ as $\sum_{i=0}^{3} (a_i + b_i) 2^{64i}$.

Since the maximum bitwidth of the sum of the limbs is 65, we say that the overflow in the limbs $a+b$ is 1 bit. For this overflowed representation to be valid, the capacity of the field $\mathbb{F}_p$ should be at least 65. Otherwise, the limb sums $a_i + b_i$ will experience a modulo reduction with respect to p, resulting in an incorrect value for $a+b$.

In general, if the current overflows of a and b are `a.overflow` and `b.overflow`, then the overflow of $a+b$ is $\max(\texttt{a.overflow}, \texttt{b.overflow}) + 1$. Overflows are intially set to zero i.e. when the limbed representations of non-native field elements are first created.

Let $\texttt{next\_overflow} = \max(\texttt{a.overflow}, \texttt{b.overflow}) + 1$. Prior to the addition of a and b, the value of `next_overflow` is compared with `maximum_overflow`. If $\texttt{next\_overflow} > \texttt{maximum\_overflow}$, then the field element amongst a and b that has a higher current overflow will be reduced modulo n to a field element with zero overflow.

Suppose $\texttt{a.overflow} > \texttt{b.overflow}$ and $\texttt{next\_overflow} > \texttt{maximum\_overflow}$. Let $a' = \text{reduce}(a)$, i.e. $a' = a \bmod n$. The addition operation between a' and b is now attempted. It is possible that b also needs to be reduced before the sum can be calculated.

3 Multiplication

Consider the product of the limbed representations of the ed25519 base field elements $a = \sum_{i=0}^{3} a_i 2^{64i}$ and $b = \sum_{i=0}^{3} b_i 2^{64i}$. Let $c = ab$. Then we have $c = \sum_{i=0}^{6} c_i 2^{64i}$ where

$$\begin{aligned}
c_0 &= a_0 b_0, \\
c_1 &= a_0 b_1 + a_1 b_0, \\
c_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0, \\
c_3 &= a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0, \\
c_4 &= a_1 b_3 + a_2 b_2 + a_3 b_1, \\
c_5 &= a_2 b_3 + a_3 b_2, \\
c_6 &= a_3 b_3.
\end{aligned}$$

Each product $a_i b_j$ will have a maximum bitwidth of 128 bits. Due to carries, the maximum bitwidths of c_1, c_5 will be 129 bits, of c_2, c_4 will be 130 bits, of c_3 will be 130 bits. So the maximum overflow across all limbs will be 130 bits. Also, note that c has 7 limbs.

In general, the product of $a = \sum_{i=0}^{k_a - 1} a_i 2^{64i}$ and $b = \sum_{i=0}^{k_b - 1} b_i 2^{64i}$ will have $k_a + k_b - 1$ limbs each having a maximum bitwidth of $m_a + m_b + \lceil \log_2 (\min(k_a, k_b)) \rceil$ where the a_i's have bitwidth m_a and the b_i's have bitwidth m_b.

4 Subtraction

Now consider the difference $a - b$ of the limbed representations of the ed25519 base field elements $a = \sum_{i=0}^{3} a_i 2^{64i}$ and $b = \sum_{i=0}^{3} b_i 2^{64i}$. In the subtraction operation, we have to ensure that there is no **underflow** when a limb is deducted from another. For any a_i that is less than b_i, the value of $a_i - b_i$ will wrap around and be equal to $p + a_i - b_i$.

We can prevent the underflow in all the limbs of $a - b$ by computing $a + kn - b$ where kn is a multiple of the prime n that ensures that $kn - b$ has no limbs that underflow. To calculate $kn - b$, we proceed as follows.

- If b has overflow `b.overflow`, then the maximum bitwidth of any limb of b is $\texttt{b.overflow} + 64$. So every limb of b is at most $2^{\texttt{b.overflow}+64} - 1$.
- We compute a big integer u corresponding to each limb being equal to its largest possible value.

$$u = \sum_{i=0}^{3} \left[2^{\texttt{b.overflow}+64} - 1 \right] 2^{64i}$$

Remark. *In the current*[1] `emulated` *implementation (April 2024), the integer u is calculated as*

$$u = \sum_{i=0}^{3} \left[2^{b.overflow+64}\right] 2^{64i}.$$

But $2^{b.overflow+64}$ has an overflow of `b.overflow` *$+1$. Choosing the expression above this remark keeps the overflow in the limbs of u to be* `b.overflow`*. Lower values of overflow allow us to further delay the reduction step.*

- We compute a remainder $r = u \bmod n$ as a big integer.
- We compute the additive inverse of the remainder as $r' = n - r$.
- We then represent the additive inverse r' in the limbed representation $r' = \sum_{i=0}^{3} r'_i 2^{64i}$. Note that there is no overflow in each limb, i.e. $\texttt{r}'.\texttt{overflow} = 0$.
- The field element corresponding to kn is then given by

  $$kn = \sum_{i=0}^{3} \left[2^{\texttt{b.overflow}+64} - 1 + r'_i\right] 2^{64i}.$$

  Note that the maximum possible overflow in kn is `b.overflow` $+1$. This is because adding r'_i can result in a carry in a limb.
- Now the limbs of $kn - b$ can be represented as the following without any chance of underflow.

  $$kn - b = \sum_{i=0}^{3} \left[2^{\texttt{b.overflow}+64} - 1 + r'_i - b_i\right] 2^{64i}.$$

  Note that the overflow of $kn - b$ continues to be `b.overflow` $+ 1$ as $b_i \leq 2^{\texttt{b.overflow}+64} - 1$.

If the current overflows of a and b are `a.overflow` and `b.overflow`, then the overflow of $a + kn - b$ is $\texttt{next\_overflow} = \max(\texttt{a.overflow}, \texttt{b.overflow} + 1) + 1$.

Prior to the calculation of $a - b$, the value of `next_overflow` is compared with `maximum_overflow`. If `next_overflow` $>$ `maximum_overflow`, then the field element amongst a and b that has a higher current overflow will be reduced modulo n to obtain a field element with zero overflow.

5 Reduction

At any point in a sequence of computations, a field element could be reduced modulo n to satisfy overflow constraints. The reduction operation will result in the overflow of the field element being reset to zero.

Recall that $\texttt{maximum\_overflow} = \texttt{capacity} - \texttt{limb\_width} - 3$. Suppose we want to reduce a field element c whose overflow satisfies $\texttt{c.overflow} \leq \texttt{maximum\_overflow}$. We proceed as follows:

- Using the big integer representation of c, we compute the remainder $r = c \bmod n$.
- We compute this remainder's limbed representation $r = \sum_{i=0}^{3} r_i 2^{64i}$. The overflow of r will be zero.
- We check that $c - r = 0 \bmod n$ by showing that the difference of their limbed representations $\sum_i (c_i - r_i) 2^{64i}$ is equal to qn for a quotient q.

[1] https://github.com/lurk-lab/bellpepper-gadgets/tree/adc96d8441141c74402d06499a63484c5458299d

5.1 Maximum overflow in the difference

The difference $d = c - r$ will have a maximum overflow of $\max(\texttt{c.overflow}, \texttt{r.overflow} + 1) + 1 = \max(\texttt{c.overflow}, 0 + 1) + 1 = \texttt{c.overflow} + 1$. We are assuming that `c.overflow` is at least 1 (if it was zero then reduction would not be needed).

Since $\texttt{c.overflow} \leq \texttt{maximum\_overflow}$, $\texttt{c.overflow} + 1 \leq \texttt{maximum\_overflow} + 1 = \texttt{capacity} - \texttt{limb\_width} - 2$. So the limbs in the difference $d = c - r$ will not experience a modular reduction.

Remark. *We will perform a limb-wise equality check between qn and d using the procedure described in Section 5.4. During the equality check, we can assume that $\texttt{qn.overflow} \leq \texttt{maximum\_overflow}$. But we can only assume that $\texttt{d.overflow} \leq \texttt{maximum\_overflow} + 1$. So the overflow in d can exceed the value of `maximum_overflow`. This violation is temporary and will disappear at the end of the reduction operation. We have to ensure that all the steps in the reduction procedure do not overflow the capacity of $\mathbb{F}_p$ in spite of this violation.*

5.2 Number of limbs in the quotient

Using the big integer representation of d, we compute the quotient q when d is divided by n. We represent q in its limbed form as $q = \sum_i q_i 2^{64i}$. The number of limbs required for the quotient depends on the maximum possible value of d and the bitwidth of n.

Let the number of limbs in $d = c - r$ be l. Since the overflow in $c - r$ is $\texttt{c.overflow} + 1$, we have $\texttt{d.overflow} = \texttt{c.overflow} + 1$. The maximum value of d is bounded by

$$\begin{aligned}
\sum_{i=0}^{l-1} \left[2^{64+\texttt{d.overflow}} - 1\right] 2^{64i} &= \left[2^{64+\texttt{d.overflow}} - 1\right] \sum_{i=0}^{l-1} 2^{64i} \\
&= \left[2^{64+\texttt{d.overflow}} - 1\right] \frac{2^{64l} - 1}{2^{64} - 1} \\
&= \frac{2^{64+\texttt{d.overflow}+64l} - 2^{64+\texttt{d.overflow}} - 2^{64l} + 1}{2^{64} - 1} \\
&\leq \frac{2^{64+\texttt{d.overflow}+64l} - 1}{2^{64} - 1} \qquad (\text{since } 2^{64+\texttt{d.overflow}} + 2^{64l} - 1 \geq 1) \\
&\leq 2^{\texttt{d.overflow}+64l+1} - 1.
\end{aligned}$$

The last inequality follows from the identity

$$\frac{2^{x+k} - 1}{2^k - 1} \leq 2^{x+1} - 1$$

when $k \geq 1$ and $x \geq 0$. In our case, $k = 64$ and $x = \texttt{d.overflow} + 64l$.

The identity can be checked by cross-multiplying and canceling terms.

$$\begin{aligned}
& 2^{x+1} - 1 \geq \frac{2^{x+k} - 1}{2^k - 1} \\
\iff & 2^{x+k+1} - 2^k - 2^{x+1} + 1 \geq 2^{x+k} - 1 \\
\iff & 2^{x+k} - 2^k - 2^{x+1} + 2 \geq 0 \\
\iff & (2^x - 1)(2^k - 2) \geq 0.
\end{aligned}$$

The last inequality holds since $k \geq 1$ and $x \geq 0$.

Thus the maximum value of d will occupy $\texttt{d.overflow} + 64l + 1$ bits where l is the number of limbs in d. If the prime p occupies $|p|$ bits, the quotient k will occupy a maximum of $\texttt{d.overflow} + 64l + 1 - |p|$ bits.

If we want to represent the quotient q in a limbed representation with 64-bit limbs and zero overflow, the number of required limbs is

$$\left\lceil \frac{\texttt{d.overflow} + 64l + 1 - |p|}{64} \right\rceil.$$

5.3 Compacting limbs

To check that $qn = d$, we first compute the product of the limbed representations of q and the prime n. The next step is to compare the limbs of qn and d while accounting for overflows. The number of R1CS constraints to perform the limb-wise comparison increases with the number of limbs.

In some cases, it might be possible to reduce the number of limbs by **compacting** them. Let us illustrate the compaction process with an example. Suppose the capacity of $\mathbb{F}_p$ is 250 bits. Let b have a non-native representation of the form $\sum_{i=0}^{3} b_i 2^{64i}$ where the limbs b_i have a bit width of 64 bits and maximum overflow of 100 bits.

The least significant limb b_0 will fit in 164 bits. The next significant limb b_1 is multiplied by 2^{64}. So $b_1 2^{64}$ will fit in 228 bits. The sum $b_0 + b_1 2^{64}$ will occupy a maximum of 229 bits, where we add one to the bitwidth of $b_1 2^{64}$ to account for the carry. Since 229 is less than the capacity 250 of $\mathbb{F}_p$, this sum can be represented in a single limb from $\mathbb{F}_p$.

Similarly, the third and fourth limbs b_2 and b_3 can be combined into a single limb as $b_2 + 2^{64} b_3$. So we have reduced the number of limbs from four to two. This process of combining limbs is called compaction. The new limb width is 128 bits and the new overflow is 101 bits.

If k limbs of b have maximum overflow `b.overflow` are combined in the compaction process, the limb width will increase from `limb_width` to $k \times$ `limb_width`. Furthermore, the overflow will increase from `b.overflow` to `b.overflow` $+ 1$. This is because the compaction process involves a sum of terms of the form

$$\sum_{i=0}^{k-1} b_i 2^{\texttt{limb\_width} \times i}.$$

The most significant term in this sum has a maximum possible bitwidth of $k \times \texttt{limb\_width} + \texttt{b.overflow}$. The sum of the other terms can induce a carry of at most 1 bit when added to this most significant term. So the overall sum has a maximum possible bitwidth of $k \times \texttt{limb\_width} + \texttt{b.overflow} + 1$ which gives us the overflow value of $\texttt{b.overflow} + 1$.

Remark. *In Section 5.4, the variable `limb_width` refers to the post-compaction limb width. If k limbs are combined in the compaction process, then*

$$\textit{post\_compaction\_limb\_width} = k \times \textit{pre\_compaction\_limb\_width}.$$

We have two requirements:

- We don't want the maximum possible bitwidth $k \times \texttt{limb\_width} + \texttt{b.overflow} + 1$ of post-compaction limbs to exceed the capacity of the native field. Hence the number of limbs k which can be combined should satisfy

$$k \leq \left\lfloor \frac{\texttt{capacity} - \texttt{b.overflow} - 1}{\texttt{limb\_width}} \right\rfloor.$$

- In Section 5.1, we remarked that the overflow of the difference $d = c - r$ satisfies $\texttt{d.overflow} \leq \texttt{maximum\_overflow} + 1$. Since the compaction process changes the limb width and the overflow, we want the post-compaction overflow of d to still be bounded above by $\texttt{maximum\_overflow} + 1$.

  We can ensure this by allowing compaction **only** when the pre-compaction overflow is less than or equal to `maximum_overflow`. Then the post-compaction overflow will be at most $\texttt{maximum\_overflow} + 1$. Consider the following formula for the number of limbs that can be combined

$$k = \left\lfloor \frac{\texttt{capacity} - \texttt{b.overflow} - 1}{\texttt{limb\_width}} \right\rfloor.$$

  Recall that $\texttt{maximum\_overflow} = \texttt{capacity} - \texttt{limb\_width} - 3$.

When $\texttt{b.overflow} = \texttt{maximum\_overflow} + 1$, the above formula evaluates to

$$k = \left\lfloor \frac{\texttt{limb\_width} + 1}{\texttt{limb\_width}} \right\rfloor.$$

Then $k = 1$ as long as $\texttt{limb\_width} \geq 2$. So no compaction will take place as long as the pre-compaction limb width is at least 2.

Remark. *A new assumption of* $\texttt{limb\_width} \geq 2$ *was made in the above argument.*

If $\texttt{b.overflow} = \texttt{maximum\_overflow}$, the formula for the number of limbs that can be combined reduces to

$$k = \left\lfloor \frac{\texttt{limb\_width} + 2}{\texttt{limb\_width}} \right\rfloor.$$

Now $k = 2$ if $\texttt{limb\_width} = 2$. So compaction can occur in the extreme case of the pre-compaction limb width being equal to 2 bits.

5.4 Limb-wise equality check

If $qn = d$, it is not necessary that their respective limbs will all be equal to each other. But they will be equal as big integers. The main challenge is to check the equality of the big integer representations of qn and d using only arithmetic in $\mathbb{F}_p$.

For example, suppose the limb width is 4 bits. The integer 100 can be written as $a = 6 \times 2^4 + 4$ and $b = 5 \times 2^4 + 20$. The limbs of a are $a_0 = 4, a_1 = 6$ and the limbs of b are $b_0 = 20, b_1 = 5$. There is no overflow in the limbs of a while the limbs of b have an overflow of 1 bit. The overflow allows the two different sets of limbs to represent the same integer. The difference of the limbs $a_0 - b_0 = -16$. This means that -1 has to be carried and added to the difference $a_1 - b_1 = 1$.

While q and n both have zero overflow, their product qn will have limbs with non-zero overflow as explained in the section on multiplication. If $\texttt{qn.overflow} > \texttt{d.overflow}$, set $a = qn$ and $b = d$. Otherwise, set $a = d$ and $b = qn$.

Recall from Section 5.1 that $\texttt{d.overflow}$ can be equal to $\texttt{maximum\_overflow}+1$. But $\texttt{qn.overflow} \leq \texttt{maximum\_overflow}$ since qn is the result of a multiplication operation (which would have been reduced if its overflow exceeded $\texttt{maximum\_overflow}$). Since we are setting b to be the non-native field element with the possibly lower overflow, we have

$$\begin{aligned}
\texttt{b.overflow} &= \min(\texttt{qn.overflow}, \texttt{d.overflow}) \leq \texttt{maximum\_overflow}, \\
\texttt{a.overflow} &= \max(\texttt{qn.overflow}, \texttt{d.overflow}) \leq \texttt{maximum\_overflow} + 1.
\end{aligned}$$

Let $\texttt{max\_value} = 2^{\texttt{limb\_width}+\texttt{b.overflow}+1}$. Note that the binary representation of $\texttt{max\_value}$ has a one bit followed by $\texttt{limb\_width} + \texttt{b.overflow} + 1$ zero bits.

- We calculate

$$e_0 = \texttt{max\_value} + a_0 - b_0,$$

which is the difference of the least significant limbs of a and b offset by $\texttt{max\_value}$ to prevent underflow. The maximum value of a limb in b is given by $2^{\texttt{limb\_width}+\texttt{b.overflow}} - 1$.

Remark. *We can set* $\texttt{max\_value}$ *to* $2^{\texttt{limb\_width}+\texttt{b.overflow}}$ *and still prevent underflow in the least significant limb. The larger value of* $\texttt{max\_value}$ *is required in the more significant limbs. We stick with the larger value for simplicity.*

The a_0 term occupies a maximum of $\texttt{limb\_width} + \texttt{a.overflow}$ bits and the $\texttt{max\_value} - b_0$ term occupies a maximum of $\texttt{limb\_width} + \texttt{b.overflow} + 2$ bits. Hence the limb e_0 occupies a maximum of $\texttt{limb\_width} + \max(\texttt{a.overflow}, \texttt{b.overflow} + 2) + 1$ bits.

Remark. *To calculate e_0 in $\mathbb{F}_p$, we require*

$$\texttt{limb\_width} + \max(\texttt{a.overflow}, \texttt{b.overflow} + 2) + 1 \leq \texttt{capacity}.$$

This inequality will hold if $\texttt{a.overflow} \leq \texttt{capacity} - \texttt{limb\_width} - 1$ *and* $\texttt{b.overflow} \leq \texttt{capacity} - \texttt{limb\_width} - 3$. *Since* $\texttt{a.overflow} \leq \texttt{maximum\_overflow} + 1$ *and* $\texttt{b.overflow} \leq \texttt{maximum\_overflow}$*, we require* maximum_overflow *to be at most* $\texttt{capacity} - \texttt{limb\_width} - 3$.

First, we check that the limb_width least significant bits in e_0 are all zeros. This will be true if $qn = d$. The addition of max_value does not affect these least significant bits as it contributes only zeros in the limb_width least significant bits.

Next, we will initialize carry_1 with the value of e_0 right shifted limb_width times,

$$\texttt{carry\_1} = e_0 \gg \texttt{limb\_width}.$$

The max_value term in e_0 will become $\texttt{max\_value\_shifted} = 2^{\texttt{b.overflow}+1}$ in carry_1.

The carry_1 value needs to be incorporated in the next limb difference. The value of carry_1 cannot be negative. But as we saw in the example involving the two representations of 100, we need to incorporate the carry resulting from $a_0 - b_0$ in the next limb difference. So we need to subtract max_value_shifted from carry_1 before using it in the next limb difference.

- We calculate

$$e_1 = \texttt{max\_value} + a_1 - b_1 + \texttt{carry\_1} - \texttt{max\_value\_shifted},$$

where

  - $\texttt{carry\_1} - \texttt{max\_value\_shifted}$ is the actual carry,
  - $a_1 - b_1$ is the difference of the next most significant limbs of a and b, and
  - max_value is added to prevent underflow.

The b_1 term occupies a maximum of $\texttt{limb\_width} + \texttt{b.overflow}$ bits and the max_value_shifted term occupies $\texttt{b.overflow}+2$ bits. If we assume $\texttt{limb\_width} \geq 2$, the sum $b_1 + \texttt{max\_value\_shifted}$ occupies a maximum of $\texttt{limb\_width}+\texttt{b.overflow}+1$ bits. Since $\texttt{max\_value} = 2^{\texttt{limb\_width}+\texttt{b.overflow}+1}$, the difference $\texttt{max\_value} - b_1 - \texttt{max\_value\_shifted}$ will not underflow. This difference occupies a maximum of $\texttt{limb\_width} + \texttt{b.overflow} + 2$ bits.

Remark. *We need the assumption that* $\texttt{limb\_width} \geq 2$ *in the above argument.*

The a_1 term occupies a maximum of $\texttt{limb\_width}+\texttt{a.overflow}$ bits and the carry_1 term occupies $\max(\texttt{a.overflow}, \texttt{b.overflow} + 2) + 1$ bits. So the sum $a_1 + \texttt{carry\_1}$ occupies a maximum of

$$\max(\texttt{limb\_width} + \texttt{a.overflow}, \max(\texttt{a.overflow}, \texttt{b.overflow} + 2) + 1) + 1$$

bits. This maximum bitwidth can be simplified to

$$\max(\texttt{limb\_width} + \texttt{a.overflow}, \texttt{b.overflow} + 3) + 1$$

as we have already assumed that $\texttt{limb\_width} \geq 2$.

Hence the limb e_1, being the sum of $\texttt{max\_value} - b_1 - \texttt{max\_value\_shifted}$ and $a_1 + \texttt{carry\_1}$ occupies a maximum of

$$\max(\texttt{limb\_width} + \texttt{b.overflow} + 2, \max(\texttt{limb\_width} + \texttt{a.overflow}, \texttt{b.overflow} + 3) + 1) + 1$$

bits. Using the $\texttt{limb\_width} \geq 2$ assumption, we can simplify this maximum bitwidth to

$$\begin{aligned}
&\max(\texttt{limbWidth} + \texttt{b.overflow} + 2, \texttt{limbWidth} + \texttt{a.overflow} + 1) + 1 \\
&= \max(\texttt{b.overflow} + 1, \texttt{a.overflow}) + \texttt{limbWidth} + 2 \\
&= \max(\texttt{a.overflow}, \texttt{b.overflow} + 1) + \texttt{limbWidth} + 2.
\end{aligned}$$

Remark. *To calculate e_1 in $\mathbb{F}_p$, we require the above maximum bitwidth to be at most $\texttt{capacity}$. This implies that $\texttt{a.overflow} \leq \texttt{capacity} - \texttt{limb\_width} - 2$ and $\texttt{b.overflow} \leq \texttt{capacity} - \texttt{limb\_width} - 3$. Since $\texttt{a.overflow} \leq \texttt{maximum\_overflow} + 1$ and $\texttt{b.overflow} \leq \texttt{maximum\_overflow}$, we require $\texttt{maximum\_overflow}$ to be at most $\texttt{capacity} - \texttt{limb\_width} - 3$. This agrees with the bound on $\texttt{maximum\_overflow}$ required to calculate e_0 in $\mathbb{F}_p$.*

Now, we check that the `limb_width` least significant bits in e_1 are all zeros. And initialize `carry_2` with the value of e_1 right shifted `limb_width` times,

$$\texttt{carry\_2} = e_1 \gg \texttt{limb\_width}.$$

The maximum bitwidth of `carry_2` is then $\max(\texttt{a.overflow}, \texttt{b.overflow}+1)+2$. In comparison, the maximum bitwidth of `carry_1` was $\max(\texttt{a.overflow}, \texttt{b.overflow}+2)+1$.

If the maximum number of limbs in a and b is 2, then we check that `carry_2` equals `max_value_shifted`. Otherwise, we continue calculating limb differences.

- We calculate

$$e_2 = \texttt{max\_value} + a_2 - b_2 + \texttt{carry\_2} - \texttt{max\_value\_shifted},$$

where

  - $\texttt{carry\_2} - \texttt{max\_value\_shifted}$ is the actual carry,
  - $a_2 - b_2$ is the difference of the next most significant limbs of a and b, and
  - `max_value` is added to prevent underflow.

As before, $\texttt{max\_value} - b_1 - \texttt{max\_value\_shifted}$ will not underflow and will occupy a maximum of $\texttt{limb\_width} + \texttt{b.overflow} + 2$ bits.

The a_2 term occupies a maximum of $\texttt{limb\_width}+\texttt{a.overflow}$ bits and the `carry_2` term occupies $\max(\texttt{a.overflow}, \texttt{b.overflow}+1)+2$ bits. So the sum $a_2 + \texttt{carry\_2}$ occupies a maximum of

$$\max(\texttt{limb\_width} + \texttt{a.overflow}, \max(\texttt{a.overflow}, \texttt{b.overflow} + 1) + 2) + 1$$

bits. This maximum bitwidth can be simplified to

$$\max(\texttt{limb\_width} + \texttt{a.overflow}, \texttt{b.overflow} + 3) + 1$$

as we have already assumed that $\texttt{limb\_width} \geq 2$. This bitwidth is **equal** to the maximum bitwidth of $a_1 + \texttt{carry\_1}$.

Thus, the maximum possible bitwidth of e_2 is $\max(\texttt{a.overflow}, \texttt{b.overflow}+1)+\texttt{limb\_width}+2$, which is the same as the maximum bitwidth of e_1. So the calculation of e_2 in $\mathbb{F}_p$ can be performed as long as $\texttt{maximum\_overflow} \leq \texttt{capacity} - \texttt{limb\_width} - 3$.

As before, we check that the `limb_width` least significant bits in e_2 are all zeros. And initialize `carry_3` with the value of e_2 right shifted `limb_width` times,

$$\texttt{carry\_3} = e_2 \gg \texttt{limb\_width}.$$

If the maximum number of limbs in a and b is 3, then we check that `carry_3` equals `max_value_shifted`. Otherwise, we continue calculating limb differences.

Remark. *At this step, the maximum bitwidth of the carry has stabilized, in the sense that the incoming and outgoing carries have the same bitwidth. So all future limb differences can be calculated in $\mathbb{F}_p$ without worrying about the calculation overflowing its capacity.*

5.5 Pseudocode

To summarize the limb-wise equality check of a and b, consider the following pseudocode:

```
max_limbs = max(a.num_limbs, b.num_limbs)
max_value = 1 << (limb_width + b.overflow + 1)
max_value_shifted = 1 << (b.overflow + 1)
carry = 0

for i in 0..max_limbs:
    e[i] = max_value + carry

    if i < a.num_limbs:
        e[i] = e[i] + a[i]
    if i < b.num_limbs:
        e[i] = e[i] - b[i]

    if i > 0:
        e[i] = e[i] - max_value_shifted

    if least_significant_limb_width_bits(e[i]) != 0:
        exit with error

    carry = e[i] >> limb_width

assert_is_equal(carry, max_value_shifted)
```