Composing Ancient Mathematical Knowledge Into Powerful Bit-fiddling

JAMIE POND

Cppcon
The C++ Conference
2024
September 15 - 20

TLDW;

New insights from Ancient Egyptian Multiplication gives us the freedom of synthesising operations that may not be present in the hardware, and doing so with the best performance. So long as your operation can be expressed by the repeated application of associativity.

In particular we have element-wise parallel multiplication of arbitrary widths.

Thanks!

Special thanks to **Eduardo Madrid** and **Scott Bruce**. Without whom this talk would not have been possible!

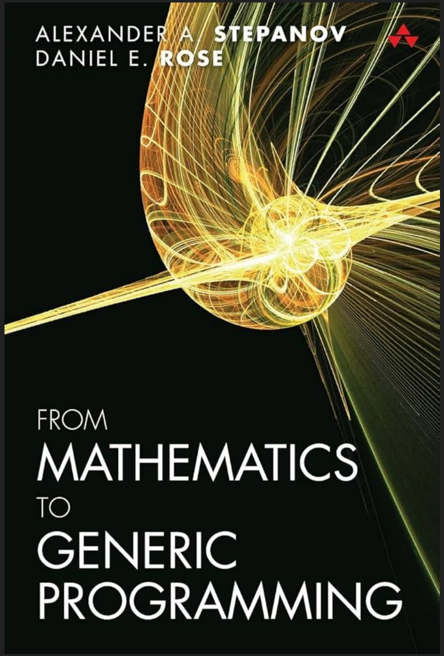
Who am I?

- Lead Developer at mayk.it
- Music tech startup in Los Angeles
- I write a lot of different types of code
    - Lots of C++ in our audio engine
    - C++ in neural inference code
    - Some Swift for our iOS
    - Some Typescript
    - Plus a bunch more…
- MSc Computer Science from Queen Mary, University of London
- BA from the London College of Music.

Inspiration for this work

- Chapter 1 of “From Mathematics to Generic Programming” talks about “Egyptian Multiplication”.

- A technique used by Ancient Egyptians to multiply by **leveraging the power of associativity.**



Associativity is *very* important to the world.

Repeated application of an associative operator is *very* important to the world.

Finite Field Exponentiation (Modular Exponentiation)

- Core operation in public-key cryptography (RSA, Diffie-Hellman)
- Can be implemented by *repeated application of modular multiplication.*
- Enables efficient computation with very large numbers
- **Essential for the functioning of modern digital life.**

Parallel Multiplication using SWAR

- We'll see later how we used the associative property to efficiently implement addition (and exponentiation) in our "software SIMD" library.
- This would not have been possible any other way, and gives *near optimal* performance (if not optimal!)

```
using S = swar::SWAR<8, uint64_t>;
```

Parallel Exponentiation with CUDA cores

- There is not a CUDA primitive for integer exponentiation (there is for doubles).
- By the end of this presentation you should have a good understanding about how you might implement your own CUDA integer exponentiation, which has *at worst* performance as good as the best library available.

Maths Briefing

What is **commutativity**? 🚙

Elements of the equation can “commute”!

If they move (commute) around it won’t affect the result.

:= as some commutative operator.

a = b # c # d # e

a = e # d # c # e

a = d # e # c # b

- **Note here that the semantics are ambiguous, since (#) is a binary operator.**

What is **associativity**? 👫

Elements of the equation can “associate” or “group” together!

If they associate (group) together in arbitrary ways it won’t affect the result.

a = (b # ((c # d) # e))

a = ((b # c) # (d # e))

a = ((b # (c # d)) # e)

But the *ordering* is still important. They cannot (necessarily) commute. ⛔🚙

It is using *this property* that we are able to transform from O(N) to O(logN) complexity.

We can understand these complex transformations through the lens of the wonderfully simple “Ancient Egyptian Multiplication”.

Egyptian Multiplication

```
Algorithm Egyptian(a, b):
    result ← 0

    while true do
        if b is odd then
            result ← result + a
            if a = 0 then
                return result
            end if
        end if

        a ← a + a
        b ← b ÷ 2
    end while
end Algorithm
```

Spreadsheet Math!

https://docs.google.com/spreadsheets/d/1oBiLOk_nTOa3nRwOpJmSSYF3wqo972zXkkOLJgehSUU/edit?gid=0#gid=0

Progressive

bin count	result	square
	0	
10110001	a	a
1011000	a	2a
101100	a	4a
10110	a	8a
1011	a+16a=17a	16a
101	49a	32a
10	49a	64a
1	177a	128a

Progressive Egyptian Multiplication

(consume the **LSB** first!)

Progressive Egyptian Multiplication

```
template <typename T>
constexpr T progressive_egpytian_multiply(T a, T multiplier) {
    T result = 0; // neutral of addition!
    for (;;) {
        if (lsb_is_on(multiplier)) { // equal to is_odd (is the lsb set?)
            result += a;
        } else if (multiplier == 0) { break; } // we consumed all the mplier!
        a <<= 1; // double
        multiplier >>= 1; // half (consume the LSB)
    }
    return result;
}
```

Progressive Egyptian Multiplication

```cpp
template <typename T>
constexpr T progressive_egpytian_multiply(T a, T multiplier) {
    T result = 0; // neutral of addition!
    for (;;) {
        if (lsb_is_on(multiplier)) { // equal to is_odd (is the lsb set?)
            result += a;
        } else if (multiplier == 0) { break; } // we consumed all the mplier!
        a <<= 1; // double
        multiplier >>= 1; // half (consume the LSB)
    }
    return result;
}
```

Progressive Egyptian Multiplication

```
template <typename T>
constexpr T progressive_egpytian_multiply(T a, T multiplier) {
    T result = 0; // neutral of addition!
    for (;;) {
        if (lsb_is_on(multiplier)) { // equal to is_odd
            result += a;
        } else if (multiplier == 0) { break; } // we consumed all the mplier!
        a <<= 1; // double
        multiplier >>= 1; // half (consume the LSB)
    }
    return result;
}
```

Progressive Egyptian Multiplication

```
template <typename T>
constexpr T progressive_egpytian_multiply(T a, T multiplier) {
    T result = 0; // neutral of addition!
    for (;;) {
        if (lsb_is_on(multiplier)) { // equal to is_odd (is the lsb set?)
            result += a;
        } else if (multiplier == 0) { break; } // we consumed all the mplier!
        a <<= 1; // double
        multiplier >>= 1; // half (consume the LSB)
    }
    return result;
}
```

Progressive Egyptian Multiplication

```cpp
template <typename T>
constexpr T progressive_egpytian_multiply(T a, T multiplier) {
    T result = 0; // neutral of addition!
    for (;;) {
        if (lsb_is_on(multiplier)) { // equal to is_odd (is the lsb set?)
            result += a;
        } else if (multiplier == 0) { break; } // we consumed all the mplier!
        a <<= 1; // double
        multiplier >>= 1; // half (consume the LSB)
    }
    return result;
}
```

Progressive Egyptian Multiplication

```
template <typename T>
constexpr T progressive_egpytian_multiply(T a, T multiplier) {
    T result = 0; // neutral of addition!
    for (;;) {
        if (lsb_is_on(multiplier)) { // equal to is_odd (is the lsb set?)
            result += a;
        } else if (multiplier == 0) { break; } // we consumed all the mplier!
        a <<= 1; // double
        multiplier >>= 1; // half (consume the LSB)
    }
    return result;
}
```

Progressive Egyptian Multiplication

```cpp
template <typename T>
constexpr T progressive_egpytian_multiply(T a, T multiplier) {
    T result = 0; // neutral of addition!
    for (;;) {
        if (lsb_is_on(multiplier)) { // equal to is_odd
            result += a;
        } else if (multiplier == 0) { break; }
        a <<= 1; // double
        multiplier >>= 1; // half (consume the LSB)
    }
    return result;
}
```

Progressive Egyptian Multiplication

```
template <typename T>
constexpr T progressive_egpytian_multiply(T a, T multiplier) {
    T result = 0; // neutral of addition!
    for (;;) {
        if (lsb_is_on(multiplier)) { // equal to is_odd (is the lsb set?)
            result += a;
        } else if (multiplier == 0) { break; } // we consumed all the mplier!
        a <<= 1; // double
        multiplier >>= 1; // half (consume the LSB)
    }
    return result;
}
```

But wait, we can go the *other way*!

We’re pretty sure no one else has done this.

“**Associative Iteration**” is a new term we coined. You won’t be able to Google it (yet)!

Regressive

bin	result
	0
10110001	a
0110001	2a
110001	5a
10001	11a
0001	22a
001	44a
01	88a
1	177a

```
Algorithm RegressiveEgyptian(multiplicand, multiplier):
    iterationCount ← number of bits in type T
    result ← 0
    while true do
        if most significant bit of multiplier is 1 then
            result ← result + multiplicand
        end if

        iterationCount ← iterationCount - 1
        if iterationCount = 0 then
            return result
        end if

        result ← result × 2
        multiplier ← multiplier × 2
    end while
end Algorithm
```

Trade-offs?

Regressive Egyptian Multiplication

(consume the **MSB** first!)

Regressive Egyptian Multiplication

```cpp
template <typename T>
constexpr T regressive_egyptian_multiply(T a, T multiplier) {
    T iterationCount = sizeof(T) * 8; // the number of bits in the type
    T result = 0; // neutral element for addition
    for (;;) {
        if (msb_is_on(multiplier)) {
          result += a;
        }
        if (!--iterationCount) { break; }
        result <<= 1; // __always__ double the result
        multiplier <<= 1; // consume the MSB
    }
    return result;
}
```

Regressive Egyptian Multiplication

```
template <typename T>
constexpr T regressive_egyptian_multiply(T a, T multiplier) {
    T iterationCount = sizeof(T) * 8; // the number of bits in the type
    T result = 0; // neutral element for addition
    for (;;) {
        if (msb_is_on(multiplier)) {
          result += a;
        }
        if (!--iterationCount) { break; }
        result <<= 1; // __always__ double the result
        multiplier <<= 1; // consume the MSB
    }
    return result;
}
```

Regressive Egyptian Multiplication

```cpp
template <typename T>
constexpr T regressive_egyptian_multiply(T a, T multiplier) {
    T iterationCount = sizeof(T) * 8; // the number of bits in the type
    T result = 0; // neutral element for addition
    for (;;) {
        if (msb_is_on(multiplier)) {
          result += a;
        }
        if (!--iterationCount) { break; }
        result <<= 1; // __always__ double the result
        multiplier <<= 1; // consume the MSB
    }
    return result;
}
```

Regressive Egyptian Multiplication

```
template <typename T>
constexpr T regressive_egyptian_multiply(T a, T multiplier) {
    T iterationCount = sizeof(T) * 8; // the number of bits in the type
    T result = 0; // neutral element for addition
    for (;;) {
        if (msb_is_on(multiplier)) {
          result += a;
        }
        if (!--iterationCount) { break; }
        result <<= 1; // __always__ double the result
        multiplier <<= 1; // consume the MSB
    }
    return result;
}
```

Regressive Egyptian Multiplication

```
template <typename T>
constexpr T regressive_egyptian_multiply(T a, T multiplier) {
    T iterationCount = sizeof(T) * 8; // the number of bits in the type
    T result = 0; // neutral element for addition
    for (;;) {
        if (msb_is_on(multiplier)) {
            result += a;
        }
        if (!--iterationCount) { break; }
        result <<= 1; // __always__ double the result
        multiplier <<= 1; // consume the MSB
    }
    return result;
}
```

Regressive Egyptian Multiplication

```
template <typename T>
constexpr T regressive_egyptian_multiply(T a, T multiplier) {
    T iterationCount = sizeof(T) * 8; // the number of bits in the type
    T result = 0; // neutral element for addition
    for (;;) {
        if (msb_is_on(multiplier)) {
          result += a;
        }
        if (!--iterationCount) { break; }
        result <<= 1; // __always__ double the result
        multiplier <<= 1; // consume the MSB
    }
    return result;
}
```

Regressive Egyptian Multiplication

```
template <typename T>
constexpr T regressive_egyptian_multiply(T a, T multiplier) {
    T iterationCount = sizeof(T) * 8; // the number of bits in the type
    T result = 0; // neutral element for addition
    for (;;) {
        if (msb_is_on(multiplier)) {
          result += a;
        }
        if (!--iterationCount) { break; }
        result <<= 1; // __always__ double the result
        multiplier <<= 1; // consume the MSB
    }
    return result;
}
```

Regressive Egyptian Multiplication

```
template <typename T>
constexpr T regressive_egyptian_multiply(T a, T multiplier) {
    T iterationCount = sizeof(T) * 8; // the number of bits in the type
    T result = 0; // neutral element for addition
    for (;;) {
        if (msb_is_on(multiplier)) {
          result += a;
        }
        if (!--iterationCount) { break; }
        result <<= 1; // __always__ double the result
        multiplier <<= 1; // consume the MSB
    }
    return result;
}
```

Regressive Egyptian Multiplication

```cpp
template <typename T>
constexpr T regressive_egyptian_multiply(T a, T multiplier) {
    T iterationCount = sizeof(T) * 8; // the number of bits in the type
    T result = 0; // neutral element for addition
    for (;;) {
        if (msb_is_on(multiplier)) {
          result += a;
        }
        if (!--iterationCount) { break; }
        result <<= 1; // __always__ double the result
        multiplier <<= 1; // consume the MSB
    }
    return result;
}
```

Thanks for coming to my talk! That’s it.

- Good job. We have a very roundabout way of multiplying integers. 👏

… ok there’s more 😉

- How does this generalize?
  - Let’s look at what we can abstract from the regressive algorithm…
- To summarize
  - We have an operation we’re doing…
  - We have some number of times to do it (the count)

Generalization of Regressive Formulation

```
template <typename T>
constexpr T general_regressive_egyptian_multiply(T a, T multiplier) {
    auto iterationCount = T{sizeof(T) * 8}, result = T{0};
    auto operation = [](auto left, auto right, auto count) {
      return msb_is_on(count) ? left + right : left;
    };
    auto count_halver = [] (auto count) { return count << 1; };
    for (;;) {
        result = operation(result, a, multiplier);
        if (!--iterationCount) { break; }
        result = operation(result, result, msb_on<T>()); // double result!
        multiplier = count_halver(multiplier);
    }
    return result;
}
```

So here's the generalized template for the regressive impl

```cpp
template<typename Base, typename IterationCount,
         typename Operator, typename CountHalver>
constexpr auto associativeOperatorIterated_regressive(
    Base base, Base neutral, IterationCount count, IterationCount forSquaring,
    Operator op, unsigned iterationCount, CountHalver ch
) {
    auto result = neutral;
    if(!iterationCount) { return result; }
    for(;;) {
        result = op(result, base, count);
        if(!--iterationCount) { break; }
        result = op(result, result, forSquaring);
        count = ch(count);
    }
    return result;
}
```

Using the regressive template!

```
template <typename T>
constexpr T template_regressive_egyptian_multiply(T input, T multiplier) {
    auto
        neutral = T{0},
        iterationCount = T{sizeof(T) * 8},
        forSquaring = msb_on<T>();

    auto operation = [](auto left, auto right, auto count) {
      return msb_is_on(count) ? left + right : left;
    };

    auto count_halver = [] (auto count) { return count << 1; };

    return associativeOperatorIterated_regressive(
        input, neutral, multiplier, forSquaring,
        operation, iterationCount, count_halver
    );
}
```

Multiplication is iterated addition

```
template <typename T>
constexpr T template_regressive_egyptian_multiply(T input, T multiplier) {
    auto
        identity = T{0},
        iterationCount = T{sizeof(T) * 8},
        forSquaring = msb_on<T>();

    auto operation = [](auto left, auto right, auto count) {
      return msb_is_on(count) ? left + right : left;
    };

    auto count_halver = [] (auto count) { return count << 1; };

    return associativeOperatorIterated_regressive(
        input, identity, multiplier, forSquaring,
        operation, iterationCount, count_halver
    );
}
```

Exponentiation is iterated multiplication

```
template <typename T>
constexpr T template_regressive_egyptian_multiply(T input, T multiplier) {
    auto
        identity = T{1},
        iterationCount = T{sizeof(T) * 8},
        forSquaring = msb_on<T>();

    auto operation = [](auto left, auto right, auto count) {
      return msb_is_on(count) ? template_regressive_egyptian_multiply(left, right) : left;
    };

    auto count_halver = [] (auto count) { return count << 1; };

    return associativeOperatorIterated_regressive(
        input, identity, multiplier, forSquaring,
        operation, iterationCount, count_halver
    );
}
```

Godbolt!

Does the compiler understand the template…?

https://godbolt.org/z/s5qesv5Mq

So what? This is still just integer multiplication?

Yup!

So, let’s use a different monoid to prove the point that this technique is generalizable.

SWAR Briefing

What is SIMD?

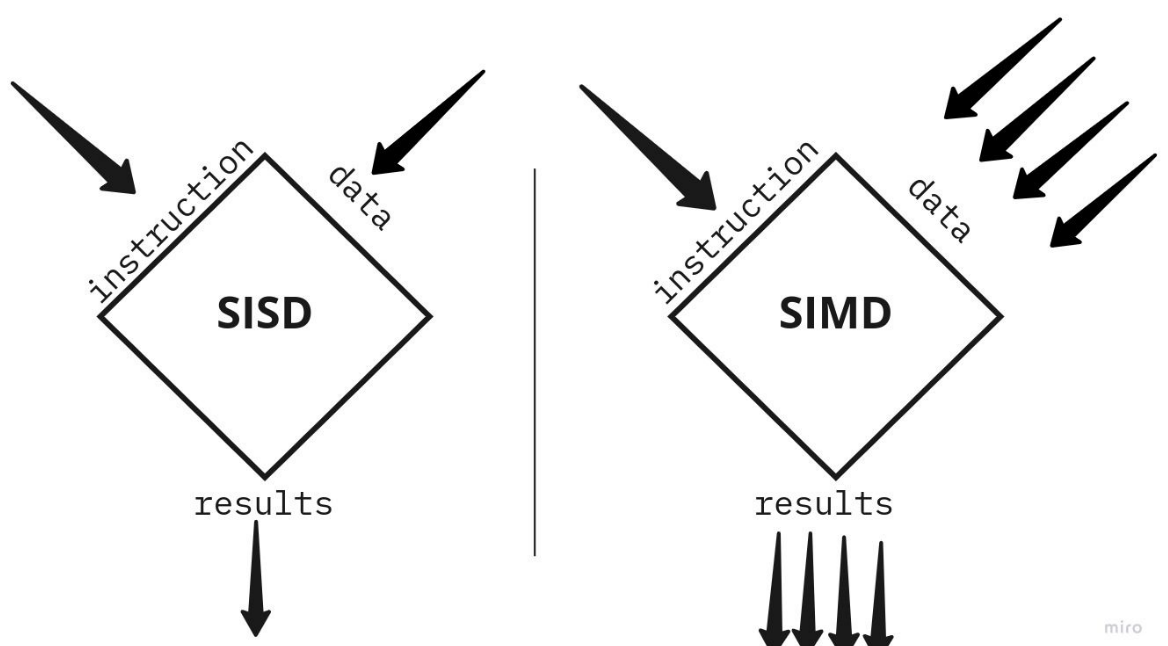
Scalar instructions execute against a single set of operands. Execute “1+5, store result 6 in c”

SIMD operations execute a Single Instruction against Multiple Data.

Data is said to be in multiple ‘lanes’.

“Add lanes [1,2,3,4] to lanes [5,6,7,8], store in d for result [6,7,10,12]”

Does not *require* special hardware!



What is SWAR…?

```
constexpr u64 a = 0b0001; // 1
constexpr u64 b = 0b0010; // 2
constexpr u64 c = 0b0011; // 3
static_assert(a + b == c);
```

Just pretend that we have “SIMD” lanes!

```
constexpr u64 a = 0b0001'0001; // 1 | 1
constexpr u64 b = 0b0010'0010; // 2 | 2
constexpr u64 c = 0b0011'0011; // 3 | 3
static_assert(a + b == c);
```

Use any base type, to determine the upper limit of lanes!

```
constexpr u64 a = 0b0001'0001'0001'0001; // 1 | 1 | 1 | 1
constexpr u64 b = 0b0010'0010'0010'0010; // 2 | 2 | 2 | 2
constexpr u64 c = 0b0011'0011'0011'0011; // 3 | 3 | 3 | 3
static_assert(a + b == c);
```

This is the same as doing (`4369 + 8738 == 13'107`)

You just interpret the result differently!

Overflow! 😬

```
constexpr u64 a = 0b0000'0001'0001'0001; // 0 | 1  | 1 | 1
constexpr u64 b = 0b0000'1111'0010'0010; // 0 | 15 | 2 | 2
constexpr u64 c = 0b0001'0000'0011'0011; // ! | !  | 3 | 3
static_assert(a + b == c);
```

Note: Library support and programmer diligence avoids over/underflow of lanes.

What does Software SIMD via SWAR cost?

- **Execution time cost is low**: we just use normal general purpose operations.
- **Software SIMD has no lane protection**, the cost comes at software implementation time.
- Ensuring correctness requires lots of bit manipulation, but these operations are very cheap!
- Superscalar CPUs do speculative, out of order, parallel instruction execution via multiple execution units.
- Adding 8 8-bit ints takes 1 add + 1 store and some bitwise ops: wildly faster than the normal 8 adds + 8 stores (16 vs 2 operations!)

Ops	Cost
Bitwise Ops, Shifts	~1
Bitwise Ops, Add, Subtract	~1
Multiply, Branch Hit, Predicted Indirect	1-3
Division, Modulo	~40
Branch Miss, Indirect Jump	10-60
L2 Cache Miss, Bad Indirect Jump	50-150

How do you use the library?

```
template<int NumBitsPerLane, typename UnderlyingType>
struct SWAR { /* impl */ };

static_assert([] {
    using S = swar::SWAR<8, uint64_t>; // 8 bit lanes, 64 bit underlying type
    S a{0x01'02'03'04'05'06'07'08};     // results in 8 lanes of 8 bits each
    S b{0x01'02'03'04'05'06'07'08};
    S c{0x02'04'06'08'0A'0C'0E'10};
    S sum = a + b;
    return horizontalEquality(sum, c);
}());
```

https://github.com/thecppzoo/zoo

SWAR Multiplication via Associative Iteration

```
template<int ActualBits, int NB, typename T>
constexpr auto multiplication_OverflowUnsafe_SpecificBitCount(
    SWAR<NB, T> multiplicand, SWAR<NB, T> multiplier
) {
    auto
        iterationCount = ActualBits,
        neutral = S{0},
        forSquaring = S{S::MostSignificantBit},
        shifted = S{multiplier.value() << (NB - ActualBits)};

    auto operation = [](auto left, auto right, auto counts) {
        return left + (makeLaneMaskFromMSB(counts) & right);
    };

    auto halver = [](auto counts) {
         auto msbCleared = counts & ~S{S::MostSignificantBit};
         return S{msbCleared.value() << 1};
    };

    return associativeOperatorIterated_regressive(
        multiplicand, neutral, shifted, forSquaring, operation,
        iterationCount, halver
    );
}
```

SWAR Multiplication via Associative Iteration

```
template<int ActualBits, int NB, typename T>
constexpr auto multiplication_OverflowUnsafe_SpecificBitCount(
    SWAR<NB, T> multiplicand, SWAR<NB, T> multiplier
) {
    auto
        iterationCount = ActualBits,
        neutral = S{0},
        forSquaring = S{S::MostSignificantBit},
        shifted = S{multiplier.value() << (NB - ActualBits)};

    auto operation = [](auto left, auto right, auto counts) {
        return left + (makeLaneMaskFromMSB(counts) & right);
    };

    auto halver = [](auto counts) {
        return counts.shiftIntraLaneLeft(1, ~S{S::MostSignificantBit});
    };

    return associativeOperatorIterated_regressive(
        multiplicand, neutral, shifted, forSquaring, operation,
        iterationCount, halver
    );
}
```

SWAR Exponentiation!

```
template<int ActualBits, int NB, typename T>
constexpr auto exponentiation_OverflowUnsafe_SpecificBitCount(
    SWAR<NB, T> base, SWAR<NB, T> exponent
) {
    auto iterationCount = ActualBits;
    auto
        neutral = S{S::LeastSignificantBit}, // all ones, in each lane
        forSquaring = S{S::MostSignificantBit},
        shifted = S{exponent.value() << (NB - ActualBits)};

    auto operation = [](auto left, auto right, auto counts) {
      auto mask = makeLaneMaskFromMSB(counts);
      auto product = multiplication_OverflowUnsafe_SpecificBitCount<ActualBits>(left, right);
      return (product & mask) | (left & ~mask);
    };

    auto halver = [](auto counts) {
        return counts.shiftIntraLaneLeft(1, ~S{S::MostSignificantBit});
    };

    return associativeOperatorIterated_regressive(
        base, neutral, shifted, forSquaring, operation,
        iterationCount, halver
    );
}
```

Getting your chain (path to the count) via Binary Expansion is not always optimal.

Progressive (parse 15)

bin count	result	square
	0	
1111	a	a
111	3a	2a
11	7a	4a
1	15a	8a

```c
// seven additions!
int mul15_naieve(int a) {
    int res = 0;
    int square = a;
    res = res + square;
    square = square + square;
    res = res + square;
    square = square + square;
    res = res + square;
    square = square + square;
    res = res + square;
    return res;
}
```

```
// five additions!
// from Stepanov & Rose
int mul15_optimal(int a) {
    int b = (a + a) + a;
    int c = b + b;
    return (c + c + b);
}
```

buuuuuttttt…

https://godbolt.org/z/n18v573oW

What if we had chains which were not just addition chains?

SWAR-lane wise chains using load effective address.

Basic demo (don’t worry we’ll break this down on the next slide!)

https://godbolt.org/z/G76z9aWc3

Extended demo (if we get time at the end!)

https://godbolt.org/z/f53b3j1bb

The compiler synergistically generating lane-wise chains.

```
m7x14(zoo::swar::SWAR<8, unsigned int>):
        movzx   eax, dil   ⟵ zero extend the low lane to the whole register
        movzx   ecx, di    ⟵ zero extend the two lanes to the whole register
        lea     eax, [rcx + 2*rax] == { 1*high, 3*low }
        lea     eax, [rcx + 2*rax] == { 3*high, 7*low }
        and     edi, 65280         == { 1*high, 0 }
        lea     eax, [rdi + 2*rax] == { 7*high, 14*low }
        ret
```

Future Work - Evaluation of Polynomials

- When evaluating polynomials, you need to have a count that makes “pit stops” at the various exponents that make up your polynomial.
- So your chain needs to go on some detour.
- This is how we can leverage associative iteration for the evaluation of polynomials.

Associative Iteration is more general than SWAR. **As general as monoids.**

- You can use Associative Iteration to implement primitives that are not present in your hardware (e.g. for SWAR or CUDA).
- You can use this technique for *anything that forms a monoid!*

You can do soooo muuuchhhhhhhh with associative iteration!

- Some more interesting monoids
    - 2x2 matrix mul (you could impl with SWAR; linearize and write ops over those lanes)
    - Many operations on quaternions form monoids

Thank you for your time 🤘