

How C++ beats other languages at card games

Using integers as arrays of bitfields, SWAR

Presented by Eduardo Madrid
Tech Lead Camera Platform
Snap, Inc.
2019



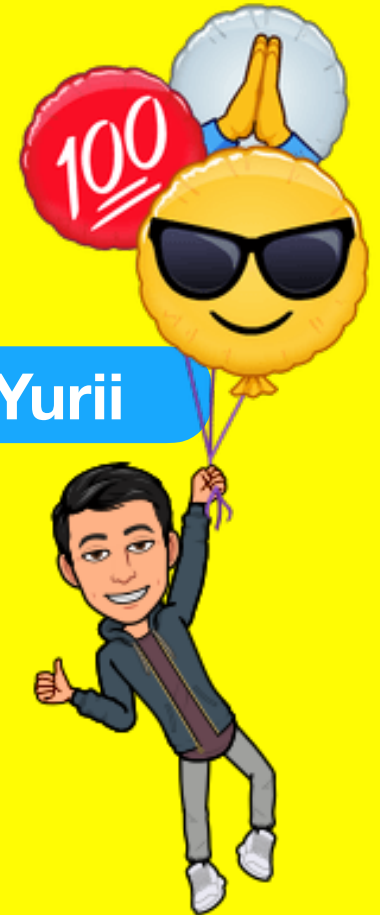
Eduardo:



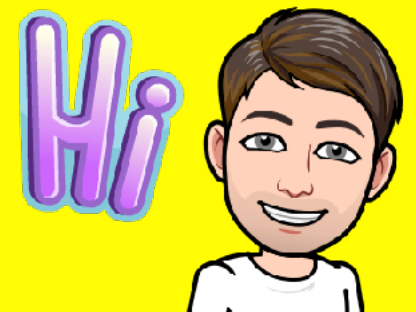
- Presented at CPPCon 2016 on metaprogramming
- Worked on Financial Technologies/Automated Trading
- You might know me from presenting at the **Chicago C++ users group**
- Author of the `zoo::function` and the Pokerbotic library we will talk about today.
- Tech Lead at the Camera Platform team, Snap, Inc:
 - Team mates are presenting a poster on how we use C++ for one of the widest AR deployments! Check it out!



Yurii



Fedir



Evgenii

OUTLINE

- What are SWAR techniques
- SWAR operations implemented for Pokerbotic
 - Useful SWAR algorithms!
- Comparisons with other approaches
- How C++ Generic Programming helps

OBJECTIVE

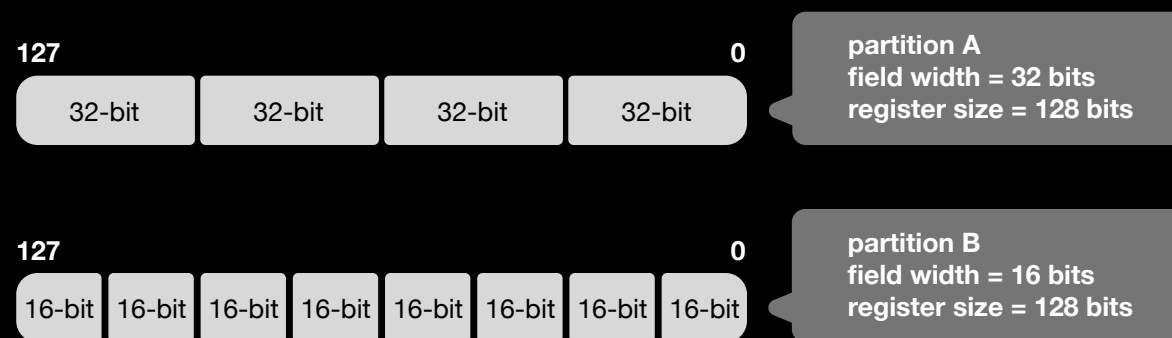
- Allow you to understand SWAR techniques by showing them within a real application domain — Poker
- Help SWAR become a little more systematic

We will see many slides!

Each thing alone helps little,
we aim for the synergy

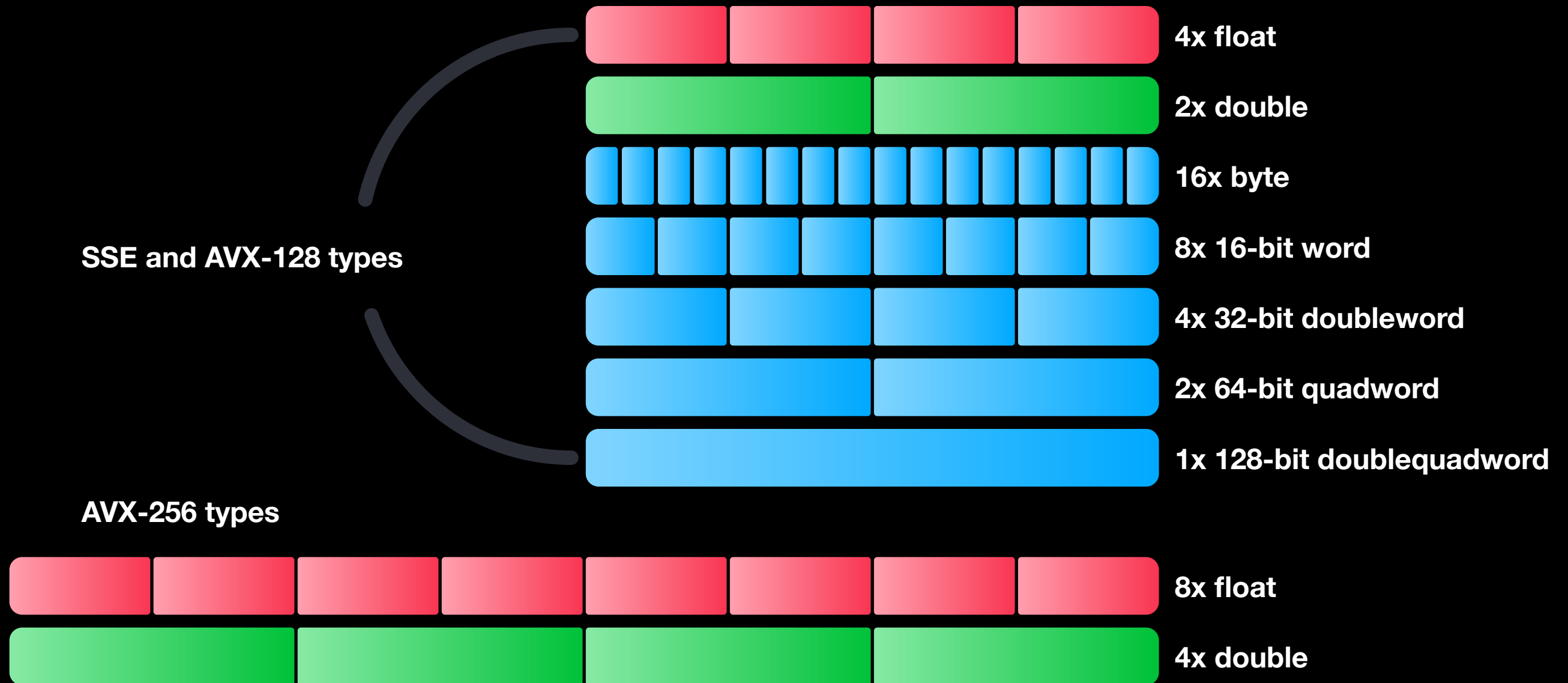
SWAR

- SIMD Within A Register (by Dietz & Fisher):
 - Registers are arrays, “Single Instruction Multiple Data”



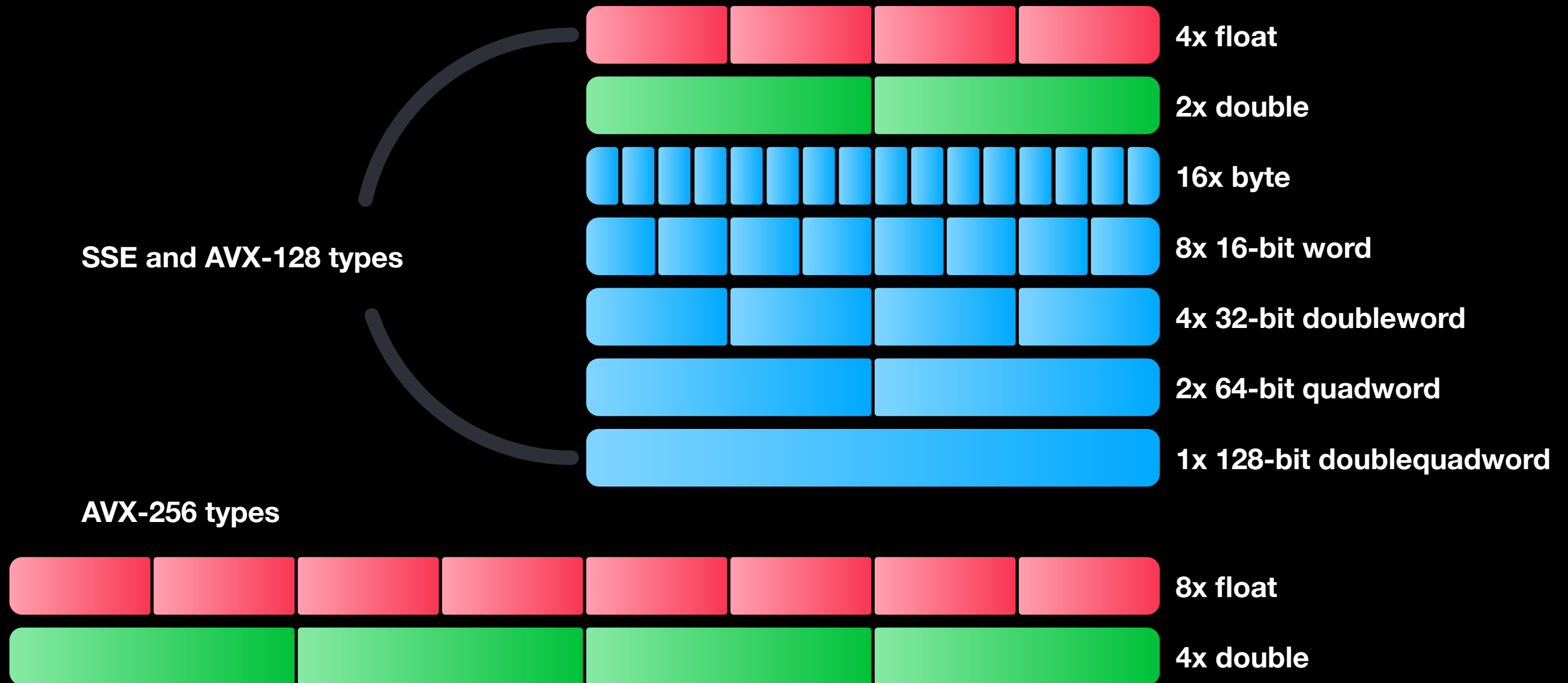
- Partitions of different sizes for the same register
- The operations occur for each element in parallel

- SIMD Examples:
 - x86-64 SSE, AVX; ARM Neon



EXAMPLE SIMD: AVX

x86-64 architecture



EXAMPLE SIMD: AVX

Notice the inconsistency: **no 256 bit integer arrays**

SIMD — X86-64 SSE

- Supports *saturation arithmetic* for 128 vectors and element sizes of 8, 16, 32 and 64 bits: When overflow occurs, the operation does not affect other elements
- Inconsistency: there are `PMOVBMSKB` and `MOVMSKPS` `MOVMSKPD`, they pick the most significant bit of: 16 bytes in the SSE register, or 4 single precision or 2 double precision. What about 8 16-bit? **There isn't the instruction!**
- Then SWAR comes to the rescue

SWAR

- Using normal 64-bit integer registers as *arrays of bitfields or smaller integers*:
 - Example: 8 8-bit bitfields, 4 16-bit
- Polymorphic operations (the element size doesn't matter):
 - AND, OR, XOR, NOT, unsigned addition
- Operations such as signed addition rely on overflow
 - Overflow is not supported!

SWAR

Signed addition example, in two's complement, arrays of elements of 4 bits.

One array contains an element of -1 (all bits set), the other the number 1:

... | ??? | 1111 | ??? | ... operand1

... | ??? | 0001 | ??? | ... operand2

... | ??? | 0000 | ??? | ... operand1 + operand2

- Normal or wide addition will “carry” 1 outward, affecting another element!

SWAR

- Our challenge #1 is to emulate support for element sizes the ISA does not support
 - Sometimes we can do it in a general way
 - Other times we can be “tactical” (will see example)
- Our challenge #2 is to remedy the possibilities for overflow

SWAR, Why bother?

- A potential n-time speedup divided by the costs of emulation.
- $+$, $-$, \sim , $\&$, $|$, \wedge , \gg , \ll are the fastest operations, then *multiplication* and others such as *deposit*, *extraction*, *population count*
- The techniques generalize to actual SIMD architectures
- They should be easy to implement in hardware such as FPGAs or even processors (example `nextSubset`?)



Now we will see code!

Complete Application

- From the user selected 2 hole cards (of 52)
 - Calculate wins, ties or loses
 - For every 5 community cards of the 50 remaining
 - For every 2 cards from 45 for the other player
- Will allow us to get a feeling for the primitives

```
int main(int argc, const char *argv[]) {
    auto
        hole1 = ep::core::convertToCard(argv[1]),
        hole2 = ep::core::convertToCard(argv[2]);
    uint64_t hole = (hole1 | hole2).value();

    constexpr auto
        stopCommunity = uint64_t(1) << 50,
        // from the fifty remaining cards
        stopHole = uint64_t(1) << 45;
        // after community, 45 remaining choices
    auto currentCommunity = uint64_t(0x1F);
        // Communities: sets of five
    auto wins = 0, ties = 0, losses = 0;
```



```
do { // for all communities
    auto community = ep::core::deposit(hole, currentCommunity);
    // converts a 5-element set of 50 to a set of 52
    auto sevenCards = hole | community;
    auto usersRank = ep::handRank(sevenCards);
    auto currentHole = uint64_t(3); // two bits set
    do { // all competitor's two hole cards
        auto competitor =
            ep::core::deposit(sevenCards, currentHole);
        auto competitorRank =
            ep::handRank(community | competitor);
        auto comparison = usersRank.code - competitorRank.code;
        if(comparison < 0) { ++losses; }
        else if(0 < comparison) { ++wins; }
        else { ++ties; }
        currentHole = ep::nextSubset(currentHole);
    } while(currentHole < stopHole);
    currentCommunity = ep::nextSubset(currentCommunity);
} while(currentCommunity < stopCommunity);
```

```
e.madrid$ time ./equity sA cA
```

```
1781502748:84.9316%
```

Matches the theory

```
11402312:0.543596%
```

```
304667340:14.5248%
```

```
2097572400
```

Google for $(50 \text{ choose } 5) * (45 \text{ choose } 2)$

```
real 0m18.360s
```

```
user 0m18.341s
```

**Over 114 million comparisons per second,
Less than 26 cycles per hand classification
at 2.9 GHz**

```
sys 0m0.007s
```

SWAR — does it work?

1. We saw how Pokerbotic compares hands of cards for the game of Texas Hold'em Poker, try any other way to get this performance?
 1. Your budget is $18.36 \times 2.9E9 / (50 \text{ choose } 5) / (45 \text{ choose } 2) < 26$ cycles per hand classification
 2. This code was “natural”
2. The fastest `atoi`, `strlen` are SWAR, potentially boosted by real SIMD instructions:
 1. Not *explicitly* SWAR, but ad-hoc
 2. We gain clarity by doing SWAR systematically

SWAR — does it work?

- [Link to one GitHub mirror of strlen](#)

SWAR AND C++

The relationship to C improved with Generic Programming

- Our community is **performance maximalist**
- Very fine granularity in the control of processing resources, like being able to topically write assembler or use intrinsic equivalents
- Generic Programming:
 - Relieves the programmer from error prone code without losing performance
 - Provides flexibility to fine tune the code

SWAR - PRIOR ART

- Slow cooking since the beginning of the history of the modern computer
 - There hasn't been a systematic effort to develop these techniques:
 - For example, the 1972 MIT Lab HACKMEM is a “**potpourri**” (says Guy L. Steele)
 - The different processor architectures are a “wild west”: inconsistent primitives and performance
- “Hacker’s Deligth” by Henry S. Warren is a compendium of techniques
- chessprogramming.org shows practical applications
- The work of Wojciech Muła, who continually has had the fastest implementations of population count using x86-64 SIMD

Pokerbotic SWAR

- Template characterized by:
 - the size in bits of the elements,
 - an underlying integer type, typically `uint64_t`
- bitwise `and`, `or`, `xor`, `not` do not care about the size of the elements (they are polymorphic)
- Unsigned addition, scaling, **won't work if overflow!**

```

template<int Size, typename T = uint64_t>
struct SWAR {
    SWAR() = default;
    constexpr explicit SWAR(T v): m_v(v) {}

    constexpr T value() const noexcept { return m_v; }

    constexpr SWAR operator| (SWAR o) { return SWAR(m_v | o.m_v); }
    constexpr SWAR operator& (SWAR o) { return SWAR(m_v & o.m_v); }
    constexpr SWAR operator^ (SWAR o) { return SWAR(m_v ^ o.m_v); }

    // ... other things omitted
protected:
    T m_v;
};

```


Size of elements in bits

Underlying integer type

```
template<int Size, typename T = uint64_t>
struct SWAR {
    SWAR() = default;
    constexpr explicit SWAR(T v): m_v(v) {}

    constexpr T value() const noexcept { return m_v; }

    constexpr SWAR operator| (SWAR o) { return SWAR(m_v | o.m_v); }
    constexpr SWAR operator& (SWAR o) { return SWAR(m_v & o.m_v); }
    constexpr SWAR operator^ (SWAR o) { return SWAR(m_v ^ o.m_v); }

    // ... other things omitted
protected:
    T m_v;
};
```

These operations don't care about the size of elements
They are **polymorphic**

Example 1:

Detecting a three of a kind

We need:

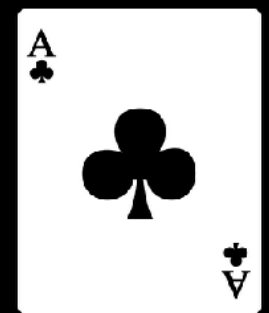
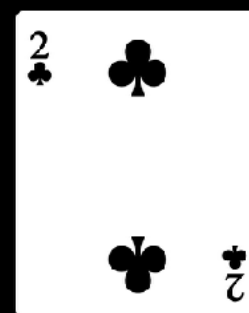
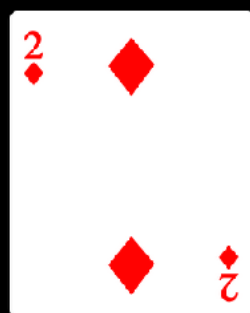
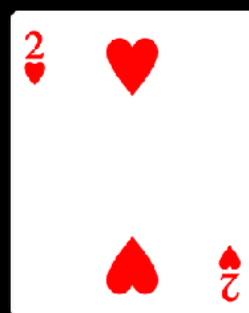
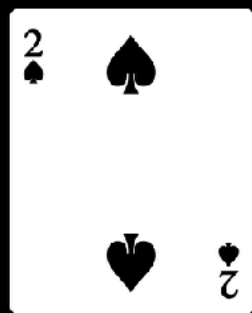
1. How to work with sets represented as bitfields
2. Accessing elements in a SWAR array
3. A “vertical comparison” operation against the constant 3
4. The way to calculate the number of bits set per element, i.e. the population count or “hamming weight”

Example 1:

Detecting a three of a kind

Each card is represented as a bit position:

0 s2 1 h2 2 d2 3 c2 — 37 hJ — 51 cA



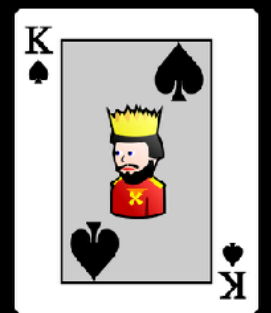
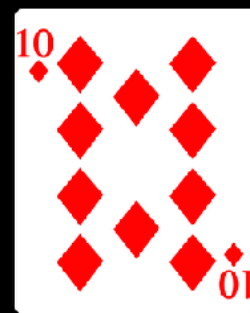
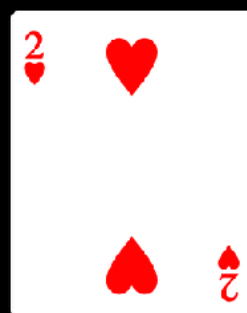
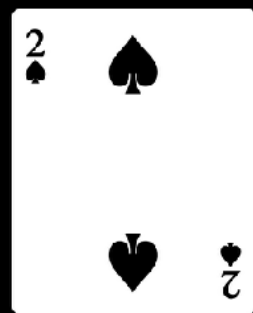
- From the least significant bit:
 - all “2”s
 - “3”s
 - ...
- Alternative views:
 - One bitfield of 52 elements
 - 13 4-bit ranks
 - Four striped arrays of 13 bits (the suits)

Example 1:

Detecting a three-of-a-kind

Community Cards: any player may use them

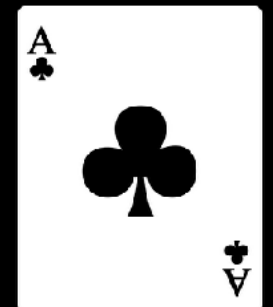
```
auto cc = s2 | h2 | dT | hJ | sK ;
```



Player Hole Cards: exclusive

```
auto phc = d2 | cA ;
```

```
auto playerHand = cc | phc;
```



THREE-OF-A-KIND OR BETTER

- A set of cards is represented by setting to 1 the corresponding bits
- Can be viewed as a SWAR of 13 elements (the ranks) of 4 bits each (the suits)
- Assume we have the count of cards in each rank, the 13 “population counts”
 - The counts are from 0 to 3, in 4 bits, thus the most significant bit in each element is zero

SET OPERATIONS FOR BITFIELDS

- Union of sets: `or`, `|`, a.k.a. disjunction
- Intersection: `and`, `&`, a.k.a. conjunction
- Complement: `not`, `~`
- Count the number of elements: population count or hamming weight
- “any”, “all”, “exists”
- `ep::nextSubset` allows to iterate by subset (original?)

Filling all elements

```
template<int Size, typename T>
constexpr SWAR<Size, T> Fill(T filler) {
    constexpr auto toDo = sizeof(T) * 8; // eight bits per byte
    auto done = Size;
    auto rv = filler;
    while(done < toDo) {
        rv |= (rv << done);
        done <<= 1;
    }
    return SWAR<Size, T>(rv);
}
static_assert(
    0x3030303030303030ull == Fill<8>(0x30ull).value(),
    "");
);
```

Filling all elements

Meant for compilation time only

```
template<int Size, typename T>
constexpr SWAR<Size, T> Fill(T filler) {
    constexpr auto toDo = sizeof(T) * 8; // eight bits per byte
    auto done = Size;
    auto rv = filler;
    while(done < toDo) {
        rv |= (rv << done);
        done <<= 1;
    }
    return SWAR<Size, T>(rv);
}

static_assert(
    0x3030303030303030ull == Fill<8>(0x30ull).value(),
    ""
);
```

Doubling every iteration

No need to manually
write “masks”

SWAR - ELEMENT ACCESS

Require bitblitting operations:

1. Setting to 1, Clearing to 0
2. Copying
3. Flipping
4. Detect differences from a pattern
5. Displacing: << and >>, zeroes get in

Basically, how to handle *sprites* in a bitmap background

BITBLITTING RECAP

We will work with 64 bits at a time in an unsigned integer

The bit positions will be specified with a *mask*:

- Each position of interest is 1 in the mask
- 0 in the mask means the operation does not affect that bit
- The reverse convention is equally good
 - The compiler will refactor and eliminate redundancies

Set to 1 specific bits

... 0 1 0 1 1 0 0 1 1 1 0 1 0 ... mask

... a ? b ? ? c d ? ? ? f ? g ... input

... a 1 b 1 1 c d 1 1 1 f 1 g ... desired result

```
auto result = input | mask;
```

Clear to 0 specific bits

... 0 1 0 1 1 0 0 1 1 1 0 1 0 ... mask

... a ? b ? ? c d ? ? ? f ? g ... input

... a 0 b 0 0 c d 0 0 0 f 0 g ... desired result

```
auto result = input & ~mask;
```

Copy bits to destination

... 0 1 0 1 1 0 0 1 1 1 0 1 0 ... mask

... ? a ? b c ? ? d e f ? g ? ... to_copy

... A ? B ? ? C D ? ? ? E ? F ... destination

... A a B b c C D d e f E g F ... desired result

```
r = (to_copy & mask) | (destination & ~mask);
```

Copy bits to destination

- Cost:
 - two masks
 - Three bitwise operations
- Blitting a copy is like a memory write: a read-modify-write
- This is used to copy a *sprite* into a bitmap background

Flip/negate bits

```
auto result = mask ^ input;
```

Detect differences compared to a pattern

```
auto result = (pattern ^ input) & mask;
```




Now you have the bitblitting power!

Element Access

```
constexpr T at(int position) {  
    constexpr auto filter = (T(1) << Size) - 1;  
    return filter & (m_v >> (Size * position));  
}
```

```
constexpr SWAR clear(int position) {  
    constexpr auto filter = (T(1) << Size) - 1;  
    auto invertedMask = filter << (Size * position);  
    auto mask = ~invertedMask;  
    return SWAR(m_v & mask);  
}
```

Element Access

```
constexpr T at(int position) {  
    constexpr auto filter = (T(1) << Size) - 1;  
    return filter & (m_v >> (Size * position));  
}
```

Masking

Repositioning

```
constexpr SWAR clear(int position) {  
    constexpr auto filter = (T(1) << Size) - 1;  
    auto invertedMask = filter << (Size * position);  
    auto mask = ~invertedMask;  
    return SWAR(m_v & mask);  
}
```

Not cheap!

VERTICAL / HORIZONTAL OPERATIONS

- Vertical operations: The result has the same number of elements
- Horizontal operations: The result is one value, the operation somehow “combines” the elements

VERTICAL / HORIZONTAL

- Vertical operations: The result has the same number of elements:
Examples:
 - bitwise `and`, `or`, `xor`, `not`
 - Vector by scalar multiplication (scaling)
 - Vector comparisons:
 - Which elements are equal? —> Generate a vector of booleans
 - Vector addition

VERTICAL / HORIZONTAL

- Horizontal operations: The result is one value, the operation somehow “combines” the elements, examples:
 - Inner product
 - Set operations of “any”, “exists”, “all”

C++ HELPS **PART 1**

- You can build the “mask” constants at compilation time
 - No errors miscounting the repetitions
 - If your sizes change, everything updates automatically
 - If your new benchmark tells you something different, you can fine-tune
- C++ 14 helps more: I originally programmed `makeBitmask` recursively, less natural

Scaling all elements

- Think of each element as a digit in base 2^{Size} . Normal multiplication scales all elements!

$$S * [E3 \mid E2 \mid E1 \mid E0]$$

—>

$$[S * E3 \mid S * E2 \mid S * E1 \mid S * E0]$$

- However, **changing the size of elements is hard or costly!**

Vertical **unsigned** addition

- Think of each element as a digit in base 2^{Size}

[A3 | A2 | A1 | A0] +

[B3 | B2 | B1 | B0] \rightarrow

[A3+B3 | A2+B2 | A1+B1 | A0+B0]

- However, **wide (normal) signed addition** does not work because **signed addition** relies on overflow wraparound
- Similar for subtraction

Vertical **signed** addition

- Thus far, very trivial operations
- Signed addition can be done this way (adapted for illustration from chessprogramming.org, **not practical** for Pokerbotic's use cases), this solves the issues of overflow wraparound by “playing” with the most significant bit of elements

```
constexpr SWAR add(SWAR sy) {  
    constexpr auto H = Fill<Size>(T(1) << (Size - 1));  
    auto x = m_v, y = sy.m_v;  
    return SWAR((x & ~H) + (y & ~H)) ^ ((x ^ y) & H);  
}
```

```
constexpr SWAR subtract(SWAR sy) {  
    constexpr auto H = Fill<Size>(T(1) << (Size - 1));  
    auto x = m_v, y = sy.m_v;  
    return SWAR((x | H) - (y & ~H)) ^ ((x ^ ~y) & H);  
}
```

BOOLEAN SWAR

We can choose how to represent that an element is true or false.

Let us use the most significant bit of the element.

Vertical greater-equal than a constant

```
template<int N, int Size, typename T>
constexpr BooleanSWAR<Size, T> greaterEqualSWAR(SWAR<Size, T> v) {
    static_assert(1 < Size, "Degenerated SWAR");
    static_assert(metaLogCeiling(N) < Size, "N is too big");
    constexpr auto msbPos = Size - 1;
    constexpr auto msb = T(1) << msbPos;
    constexpr auto msbMask = makeBitmask<Size, T>(msb);
    constexpr auto subtraend = makeBitmask<Size, T>(N);
    auto adjusted = v.value() | msbMask;
    auto rv = adjusted - subtraend;
    rv &= msbMask;
    return rv;
}
```

Vertical greater-equal than a constant

Greater or equal to N

```
template<int N, int Size, typename T>
constexpr BooleanSWAR<Size, T> greaterEqualSWAR(SWAR<Size, T> v) {
    static_assert(1 < Size, "Degenerated");
    static_assert(metaLogCeiling(N) < Si
    constexpr auto msbPos = Size - 1;
    constexpr auto msb = T(1) << msbPos;
    constexpr auto msbMask = makeBitmask<Size, T>(msb);
    constexpr auto subtraend = makeBitmask<Size, T>(N);
    auto adjusted = v.value() | msbMask;
    auto rv = adjusted - subtraend;
    rv &= msbMask;
    return rv;
}
```

The most significant bit in an element, 8

Fill with 8

Fill with 3

We subtract 3 from elements that have at least 8, it won't "borrow"

We only care for elements that still have the most significant bit set, they were greater or equal to three

Vertical greater-equal than a constant

Takes:

1. loading a constant
2. three of the cheapest operations

For all 13 ranks at once!

EXISTS, ALL, ANY

- Exists: at least one element in the array is true: the boolean SWAR **viewed as an integer is not zero**
- All: all elements are true, **equal to the fill of the most significant bit**
- Any:
 - If “exists”, then we can get the index of a “true” element
 - The first from the **highest index** element true:
 - `__builtin_clz` ← count leading zeroes
 - The first from the **lowest index** element true
 - `__builtin_ctz` ← count trailing zeroes



**PUTTING
THESE
TOGETHER....**


```
inline HandRank handRank(CardSet hand) {
    RankCounts rankCounts{SWARRank(hand.cards())};
    auto toaks = rankCounts.greaterEqual<3>();
    if(toaks) {
        auto foaks = rankCounts.greaterEqual<4>();
        if(foaks) {
            auto foak = foaks.top();
            auto without = rankCounts.clearAt(foak);
            return {
                FOUR_OF_A_KIND,
                1 << foak,
                1 << without.best()
            };
        }
        auto toak = toaks.top();
        auto without = rankCounts.clearAt(toak);
        auto pairs = without.greaterEqual<2>();
        if(pairs) { // a full-house
```

Congrats



Now you have the power
of n-of-a-kind!

Example 2:

Population count

- Counts the number of bits set.
- Also called “hamming weight”, [excellent Wikipedia article](#)
- It is a “combinatorial function”
- Most processors have the instruction, but...
 - Not in all SWAR element sizes
 - Not in parallel (SIMD)
- GCC/Clang give `__builtin_popcount` with the suffixes for sizes
 - Emulates it on x86-64 unless `-march=native` or `-msse4.2`

Combinatorial Functions:

The output only depends on the input


- Lookup tables is a popular choice to implement them
 - Including population count
 - Poker Stove, a good library and application for Poker used lots of lookup tables
 - Pokerbotic assumes lookup tables for population count can't be faster than the processor instruction

Combinatorial Functions: as Lookup Tables

- Implementing SWAR lookups is hard:
 - How do you lookup several entries in parallel?
 - The x86-64 SIMD instruction `PSHUFQ` is pure magic, but off topic
- They put pressure on memory resources:
 - Great for benchmarks, bad for real life performance
- The performance depends on the access pattern!

Pokerbotic uses of population count

- N-of-a-kind
 - 13 4-bit population counts, strong candidate for SWAR
- Flushes
 - 4 13-bit population counts of stripes
- When is calling repeatedly the processor instruction faster than the SWAR logic?



**POPULATION
COUNT ALLOWS
FLUSH
DETECTION!**

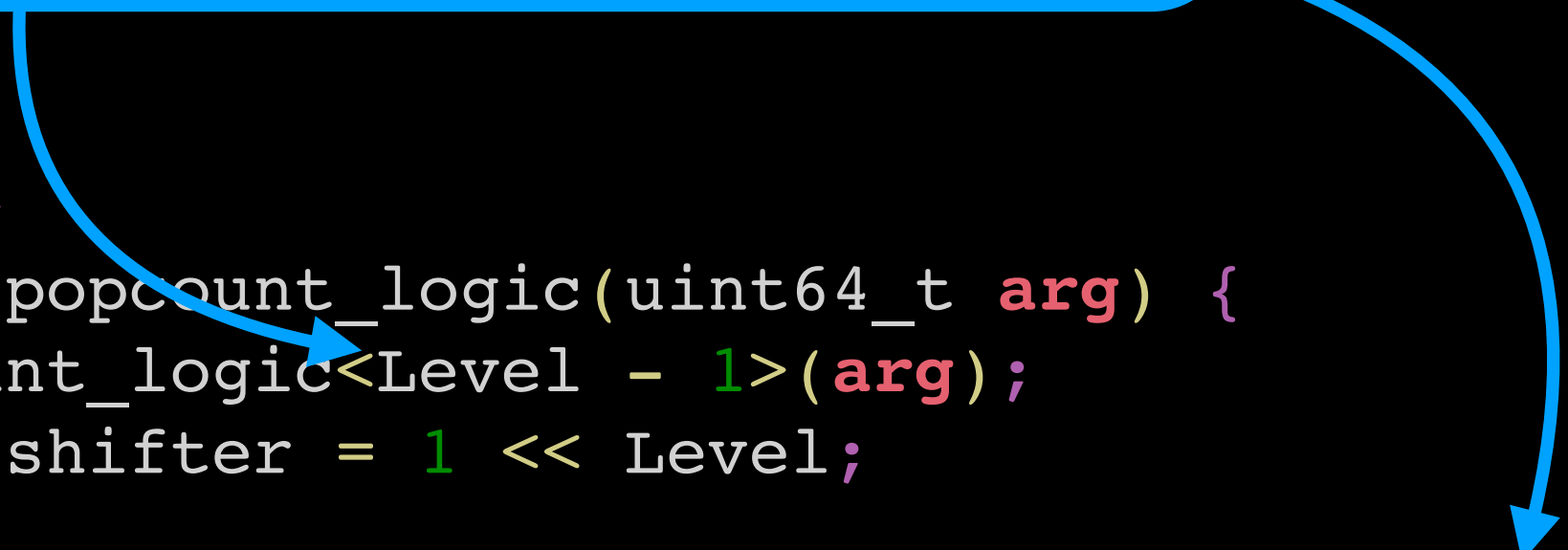
SWAR population count

```
template<int Level>
constexpr uint64_t popcount_logic(uint64_t arg) {
    auto v = popcount_logic<Level - 1>(arg);
    constexpr auto shifter = 1 << Level;
    return
        ((v >> shifter) & detail::popcountMask<Level>) +
        (v & detail::popcountMask<Level>);
}
```


SWAR population count

Key idea: combine population counts of half the bits

```
template<int Level>
constexpr uint64_t popcount_logic(uint64_t arg) {
    auto v = popcount_logic<Level - 1>(arg);
    constexpr auto shifter = 1 << Level;
    return
        ((v >> shifter) & detail::popcountMask<Level>) +
        (v & detail::popcountMask<Level>);
}
```



Do we get to see a benchmark
comparing SWAR logic and
repeating the POPCNT instruction?

Do we get to see a benchmark
comparing SWAR logic and
repeating the POPCNT instruction?



Catch2 Benchmark

```
template<int Level>
auto benchmarkBuiltin(std::vector<uint64_t> &inputs) {
    SWAR
    uint64_t rv = 0;
    for(auto input: inputs) {
        rv ^= ep::core::popcount_builtin<Level>(input);
    }
    return rv;
}

TEST_CASE("Population Count", "[pokerbotic][SWAR][popcount][sse4.2]") {
    std::mt19937_64 gen;
    std::vector<uint64_t> inputs;
    for(auto count = 1021; count-->0; ) {
        inputs.push_back(gen());
    }
    BENCHMARK("2 bits, SWAR") { return benchmarkSWAR<0>(inputs); };
    // ...
}
```

benchmark name	samples	iterations	estimated
	mean	low mean	high mean
	std dev	low std dev	high std dev

2 bits, SWAR	100	397	3.5333 ms
	86 ns	84 ns	90 ns
	10 ns	0 ns	22 ns
4 bits, SWAR	100	201	3.5175 ms
	173 ns	172 ns	176 ns
	10 ns	4 ns	17 ns
8 bits, SWAR	100	133	3.5378 ms
	263 ns	259 ns	278 ns
	35 ns	12 ns	78 ns
8 bits, builtin	100	20	3.542 ms
	1.786 us	1.739 us	1.887 us
	343 ns	200 ns	546 ns

16 bits, SWAR	100	98	3.5378 ms
	342 ns	341 ns	348 ns
	11 ns	0 ns	28 ns
16 bits, builtin	100	39	3.5334 ms
	1.061 us	994 ns	1.151 us
	392 ns	314 ns	511 ns
32 bits, SWAR	100	82	3.567 ms
	421 ns	420 ns	428 ns
	16 ns	0 ns	39 ns
32 bits, builtin	100	90	3.555 ms
	433 ns	405 ns	475 ns
	170 ns	127 ns	234 ns
64 bits, builtin	100	187	3.5343 ms
	175 ns	174 ns	178 ns
	6 ns	0 ns	16 ns

Over 5.9 SWAR 4-bit population
counts (13 ranks) per nanosecond!

Faster than the builtin!

Freshly
Baked



I am not cheating:

<https://godbolt.org/z/oqpDB1>

```
auto nibbles(uint64_t a) {  
    return popcount_logic<1>(a);  
}  
  
auto longs(uint64_t a) {  
    return popcount_builtin<3>(a);  
}  
  
nibbles(unsigned long):  
    mov     rax, rdi  
    shr     rax  
    movabs  rcx, 6148914691236517205  
    and     rcx, rax  
    sub     rdi, rcx  
    mov     rax, rdi  
    shr     rax, 2  
    movabs  rcx, 3689348814741910323  
    and     rax, rcx  
    and     rdi, rcx  
    lea     rax, [rdi + rax]  
    ret  
  
longs(unsigned long):  
    popcnt  rax, rdi  
    ret
```

Clang vectorizes the *ridiculously hell out*:
The simplistic benchmark folly strikes again!

4 4-bit SWARs per vector — 52 ranks!

Also: I like it when the compiler likes my
code so much it auto-vectorizes...

C++ HELPS PART 2

- It allows expressing these “divide and conquer” strategies *at compilation time*.
- The template recursion has no effect on the compiler’s ability to optimize
- The code tells the compiler so much we just saw how it may surprise you with ridiculous performance...

Example 3:

Straights

Simple way (just ranks, not taking into account aces):

```
unsigned straights(unsigned cards) {  
    auto shifted1 = cards << 1;  
    auto shifted2 = cards << 2;  
    auto shifted3 = cards << 3;  
    auto shifted4 = cards << 4;  
    return cards & shifted1 &  
        shifted2 & shifted3 & shifted4;  
}
```

Straights - simple

Example: cards: sK dT s9 h8 c7 c6 d5

A **K** Q J **T** **9** **8** **7** **6** **5** 4 3 2 A &

K Q J **T** **9** **8** **7** **6** **5** 4 3 2 A - &

Q J **T** **9** **8** **7** **6** **5** 4 3 2 A - - &

J **T** **9** **8** **7** **6** **5** 4 3 2 A - - - &

T **9** **8** **7** **6** **5** 4 3 2 A - - - -

=====

- - - - 1 1 - - - - - - -

Straights

- SWAR ranks:
 - Use `greaterEqualSWAR<1>` to make it “color blind”
 - Not very different
- Aces: **blit them to the bottom, do not use conditionals**
- Can it be done better?
 - Checking for the presence of 5 or 10 does NOT pay for itself: high-entropy conditional branch that the branch predictor hates
 - Yes: read “From Mathematics To Generic Programming” by Stepanov et al. and you’ll learn **addition chains**

Straights - addition chain

```
unsigned straights(unsigned cards) {  
    // assume the point of view from the bit position for tens.  
  
    auto shift1 = cards >> 1;  
    // in shift1 the bit for the rank ten contains jacks  
  
    auto tj = cards & shift1;  
    auto shift2 = tj >> 2;  
    // in shift2 the bit for the rank ten contains queen&king  
  
    auto tjqk = tj & shift2;  
    return tjqk & (cards >> 4);  
}
```

Straights - addition chain

Example: cards: sK dT s9 h8 c7 c6 d5

A	K	Q	J	T	9	8	7	6	5	4	3	2	A == cards
K	Q	J	T	9	8	7	6	5	4	3	2	A	- == shift1
AK	KQ	QJ	JT	T9	98	87	76	65	54	43	32	2A	- == t9
QJ	JT	T9	98	87	76	65	54	43	32	2A	-	-	- == shift2
A-J	K-T	Q-9	J-8	T-7	9-6	8-5	7-4	6-3	5-2	4-A	-	-	- == t987
T	9	8	7	6	5	4	3	2	A	-	-	-	- == cards << 4

=====

- - - - **T-6 9-5** - - - - - - - -

Straights - addition chain

- Three pairs of conjunction/shift operations versus four:
33% less work



**You can code Poker in C++ with great
performance:
GRADUATION!**

**AND
ANY OTHER
CARD GAME**

Postgraduate Programs

- Iterating over the power set
- The deposit instruction:
 - Minor degree in the Floyd Sampling algorithm with preselections
- `strlen`, with explicit SWAR code
- using integer multiplication as SWAR vector inner product
 - Population count of flush, horizontal summation, `atoi`

Example 4:

All 3 element subsets

Condition: as an integer, **less than 64** (sets of 5 elements)

...0 0**0**111

...0 01**0**11

...0 011**0**1

...0 **0**1110

...0 10**0**11

...0 101**0**1

...0 1**0**110

...0 110**0**1

...0 11**0**10

...**0** 11100

...**1** 00011 == 67 > 64

Subsets:

- Identify the first 0 *after seeing a 1*
- Turn on that bit
- Put the “remaining ones” to the least significant position

Subsets:

`ep :: nextSubset`

- In nextSubset.h
- Requires 13 operations
- Implement it in FPGAs? In the processor?
- Original idea?
- Benchmarks almost as fast as nested loops

Example 5:

Horizontal Summation

[A | B | C | D] *

[1 | 1 | 1 | 1]

A	B	C	D
B	C	D	0
C	D	0	0
D	0	0	0

=====

[**A+B+C+D** | B+C+D | C+D | D]

- Use normal multiplication as the inner product
- The highest index element has the summation

Example 6:

Numeric base conversion:

8-byte `atoi`

- Daniel Lemire's 8-byte `atoi`,
`parse eight digits swar`



**CONCLUSION:
MATHEMATICS!**

