SIMD at Insomniac Games

(How we do the shuffle)

Andreas Fredriksson
Lead Engine Programmer, Insomniac Games

GAME DEVELOPERS CONFERENCE®
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015

Hi, I'm Andreas!

- One of the Engine leads at Insomniac Games
  - Heading up the fearless "FedEx" Core group
- Our focus: Engine runtime + infrastructure
  - But also involved in gameplay optimization
  - We're all about getting the job done

Talk Outline

- SSE Intro
- Gameplay Example
- Techniques (tricks!)
- Best Practices
- Resources + Q & A

SIMD at Insomniac Games

- Long history of SIMD programming in the studio
  - PS2 VU, PS3 SPU+Altivec, X360 VMX128, SSE(+AVX)

SIMD at Insomniac Games

- Long history of SIMD programming in the studio
  - PS2 VU, PS3 SPU+Altivec, X360 VMX128, SSE(+AVX)
- Focus on SSE programming for this cycle
  - Even bigger incentive when PCs+consoles share ISA
  - PC workstations are ridiculously fast when you use SIMD

SIMD at Insomniac Games

- Long history of SIMD programming in the studio
  - PS2 VU, PS3 SPU+Altivec, X360 VMX128, SSE(+AVX)
- Focus on SSE programming for this cycle
  - Even bigger incentive when PCs+consoles share ISA
  - PC workstations are ridiculously fast when you use SIMD
- Lots of old best practices don’t apply to SSE

Why CPU SIMD?

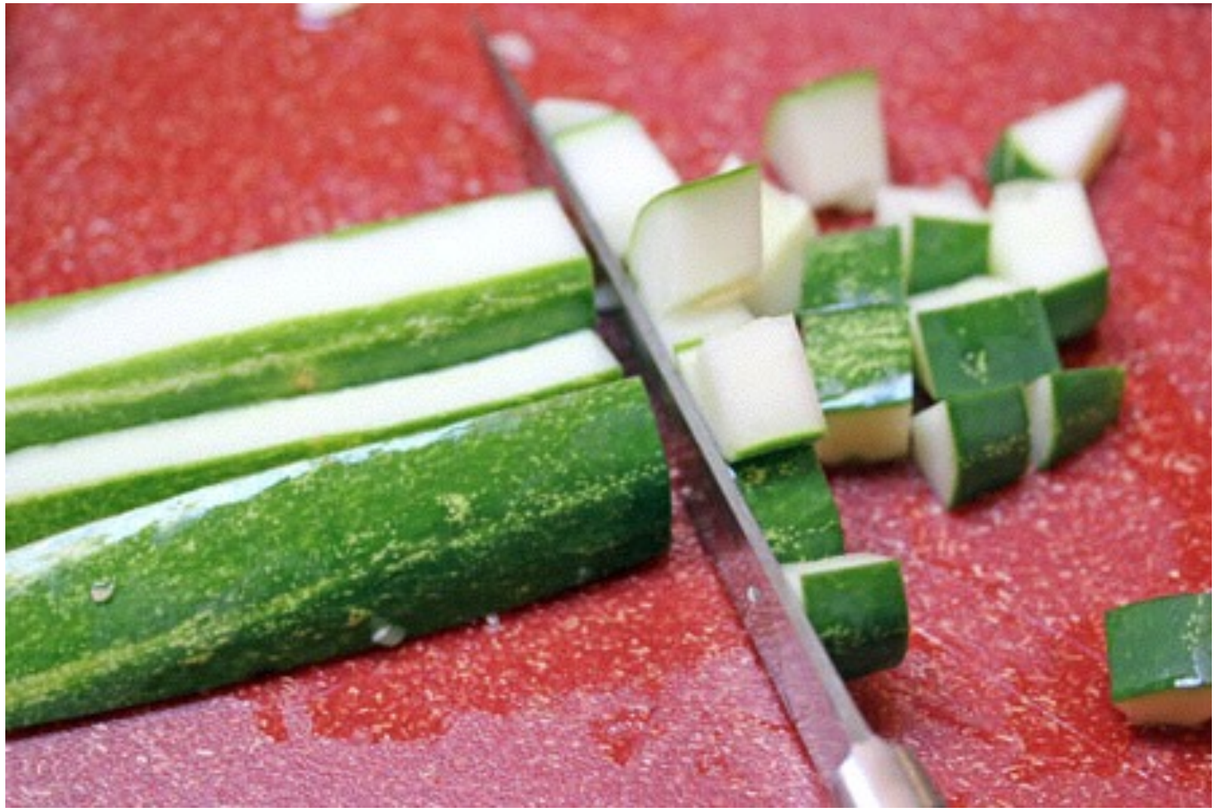
- Isn’t everything GPGPU nowadays?
  - Definitely not!
  - Don’t want to waste the x86 cores on current consoles
  - Many problems are too small to move to GPU

Why CPU SIMD?

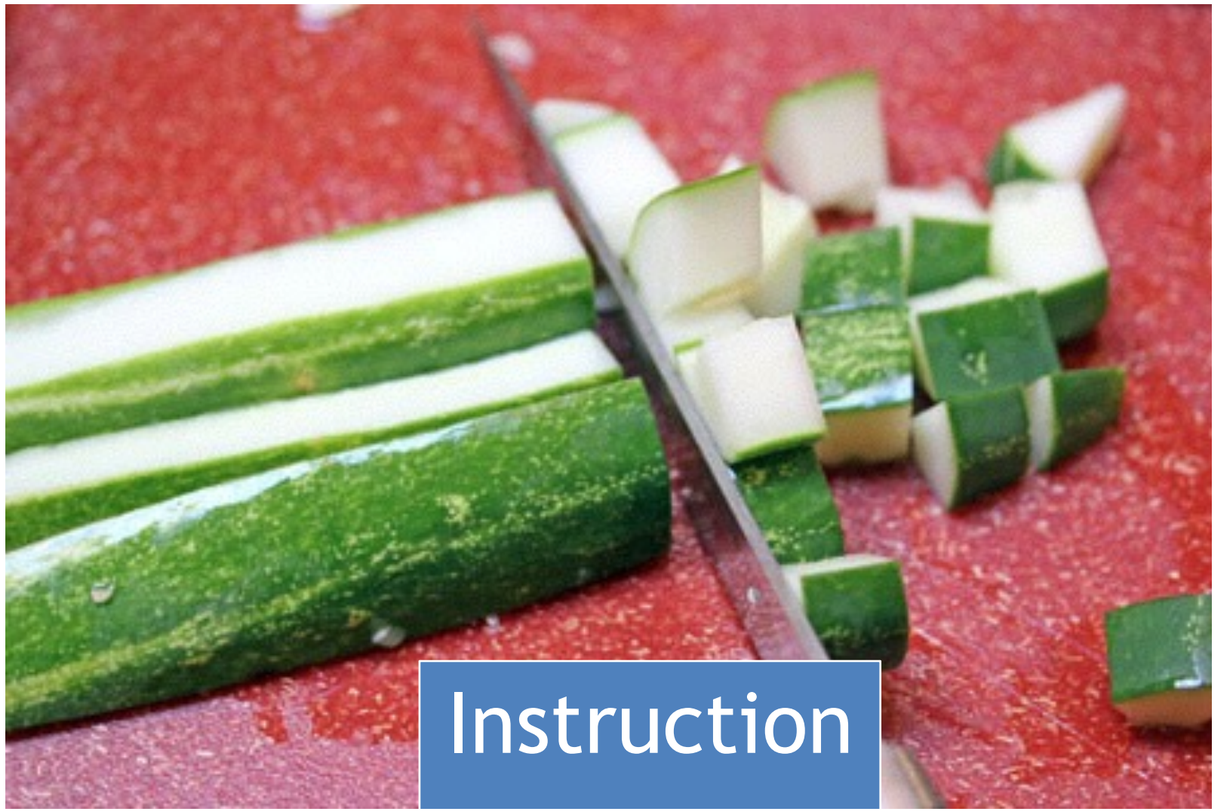
- Isn’t everything GPGPU nowadays?
  - Definitely not!
  - Don’t want to waste the x86 cores on current consoles
  - Many problems are too small to move to GPU
- Never underestimate brute force + linear access
  - CPU SIMD can give you massive performance boosts
  - Don’t want to leave performance on the table

SIMD

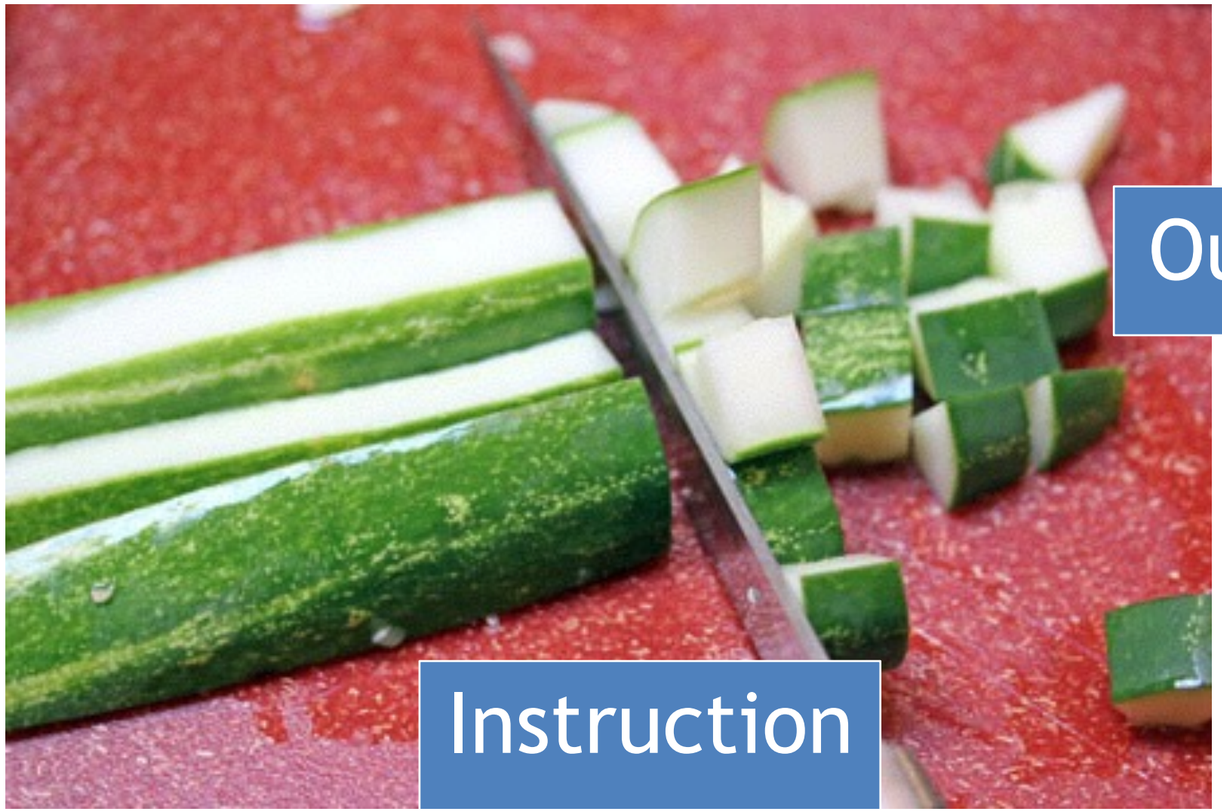
SIMD



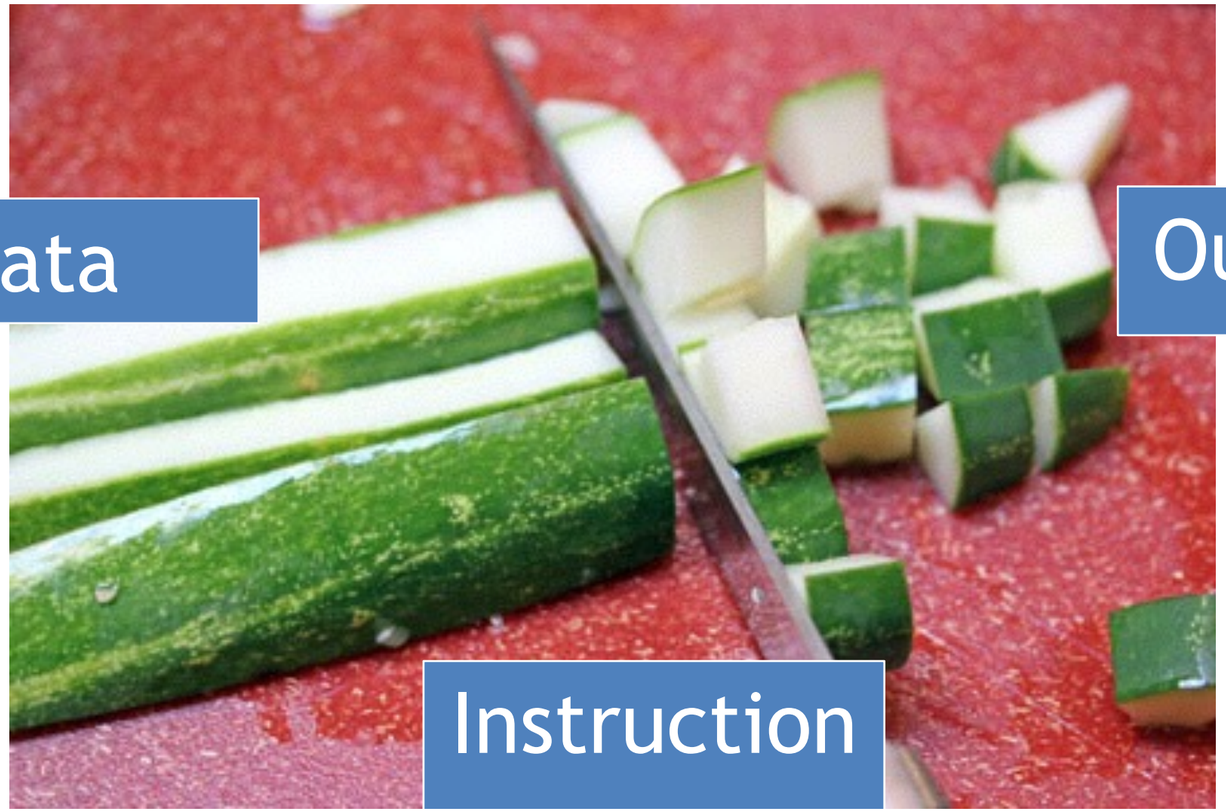
SIMD



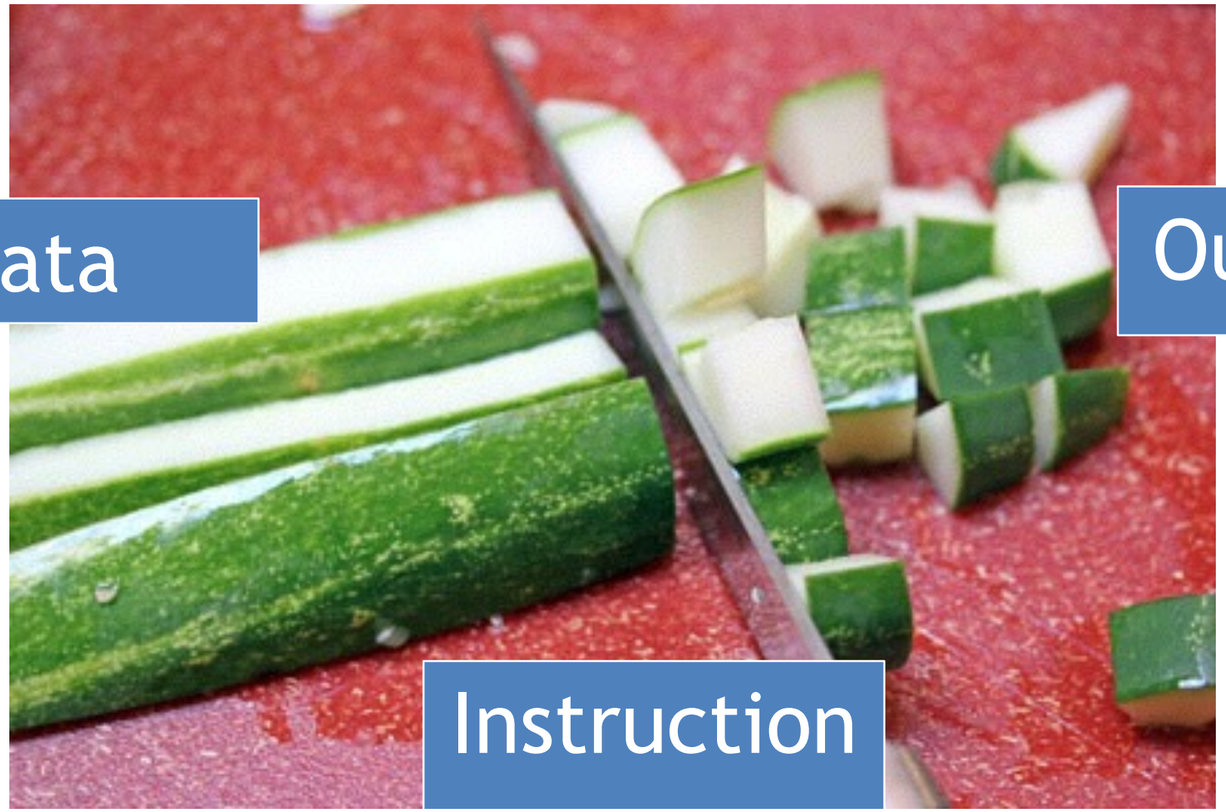
SIMD



SIMD



SIMD



.. It’s just like dicing veggies!

Options for SSE and AVX SIMD

- Compiler auto-vectorization
- Intel ISPC
- Intrinsics
- Assembly

Compiler Auto-vectorization

- Utopian idea, doesn’t work well in practice
  - Compilers are tools, not magic wands

Compiler Auto-vectorization

- Utopian idea, doesn’t work well in practice
  - Compilers are tools, not magic wands
- Often breaks during maintenance
  - Left with a fraction of the performance!

Compiler Auto-vectorization

- Utopian idea, doesn't work well in practice
  - Compilers are tools, not magic wands
- Often breaks during maintenance
  - Left with a fraction of the performance!
- Compiler support/guarantees = terrible
  - No support in VS2012, some in VS2013
  - Different compilers have different quirks

Auto-vectorization

```
float Foo(const float input[], int n)
{
  float acc = 0.f;

  for (int i = 0; i < n; ++i) {
    acc += input[i];
  }

  return acc;
}
```

GCC 4.8: Vectorized

Clang 3.5: Vectorized

(Only with -ffast-math)

Let’s make some changes..

```
float Foo(const float input[], int n)
{
  float acc = 0.f;

  for (int i = 0; i < n; ++i) {
    float f = input[i];
    if (f < 10.f)
      acc += f;
  }

  return acc;
}
```

GCC 4.8: Vectorized

Clang 3.5: Vectorized

(Both branch free)

Auto-vectorization gone wrong

```
float Foo(const float input[], int n)
{
  float acc = 0.f;

  for (int i = 0; i < n; ++i) {
    float f = input[i];
    if (f < 10.f)
      acc += f;
    else
      acc -= f;
  }

  return acc;
}
```

GCC 4.8: scalar + branchy

Clang 3.5: scalar + branchy

ISPC

- Shader-like compiler for SSE/AVX
  - Write scalar code, ISPC generates SIMD code
  - Requires investment in another level of abstraction
  - Caution—easy to generate inefficient load/store code

ISPC

- Shader-like compiler for SSE/AVX
  - Write scalar code, ISPC generates SIMD code
  - Requires investment in another level of abstraction
  - Caution—easy to generate inefficient load/store code
- Main benefit: automatic SSE/AVX switching
  - Example: Intel's BCT texture compressor
  - Automatically runs faster on AVX workstations

Intrinsics

- Taking control without dropping to assembly
  - Preferred way to write SIMD at Insomniac Games

Intrinsics

- Taking control without dropping to assembly
  - Preferred way to write SIMD at Insomniac Games
- Predictable—no invisible performance regressions

Intrinsics

- Taking control without dropping to assembly
  - Preferred way to write SIMD at Insomniac Games
- Predictable—no invisible performance regressions
- Flexible—exposes all CPU features

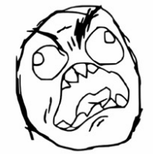
Intrinsics

- Taking control without dropping to assembly
  - Preferred way to write SIMD at Insomniac Games
- Predictable—no invisible performance regressions
- Flexible—exposes all CPU features
- Hard to learn & get going..
  - Not a real argument against
  - All good programming is hard (and bad programming easy)

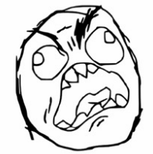
Assembly

- Always an option!

Assembly

- Always an option!
- No inline assembly on 64-bit VS compilers 
  - Need external assembler (e.g. yasm)

Assembly

- Always an option!
- No inline assembly on 64-bit VS compilers 
  - Need external assembler (e.g. yasm)
- *Numerous* pitfalls for the beginner
  - Tricky to maintain ABI portability between OSs
  - Non-volatile registers
  - x64 Windows exception handling
  - Stack alignment, debugging, …

Why isn’t SSE used more?

- Fear of fragmentation in PC space
  - SSE2 supported on *every* x64 CPU, but usually much more

Why isn’t SSE used more?

- Fear of fragmentation in PC space
  - SSE2 supported on *every* x64 CPU, but usually much more
- “It doesn’t fit our data layout”
  - PC engines traditionally OO-heavy
  - Awkward to try to bolt SIMD code on an OO design

Why isn’t SSE used more?

- Fear of fragmentation in PC space
  - SSE2 supported on *every* x64 CPU, but usually much more
- “It doesn’t fit our data layout”
  - PC engines traditionally OO-heavy
  - Awkward to try to bolt SIMD code on an OO design
- “We’ve tried it and it didn’t help”

“We’ve tried it and it didn’t help”

- Common translation: “We wrote class Vec4…”

```
class Vec4 {
  __m128 data;

  operator+ (…)
  operator- (…)
};
```

class Vec4

__m128 has X/Y/Z/W.

So clearly it’s a 4D vector.

class Vec4, later that day

Addition and multiply is going great!

This will be really fast!

class Vec4, that night

Oh no!

Dot products and other common operations are really awkward and slow!

The Awkward Vec4 Dot Product

```
??? Vec4Dot(Vec4 a, Vec4 b)
{
  __m128 a0  = _mm_mul_ps(a.data, b.data);

  // Wait, how are we going to add the products together?


  return ???; // And what do we return?
}
```

The Awkward Vec4 Dot Product

```
Vec4 Vec4Dot(Vec4 a, Vec4 b)
{
  __m128 a0  = _mm_mul_ps(a.data, b.data);
  __m128 a1  = _mm_shuffle_ps(a0, a0, _MM_SHUFFLE(2, 3, 0, 1));
  __m128 a2  = _mm_add_ps(a1, a0);
  __m128 a3  = _mm_shuffle_ps(a2, a2, _MM_SHUFFLE(0, 1, 3, 2));
  __m128 dot = _mm_add_ps(a3, a2);
  return dot; // WAT: the same dot product in all four lanes
}
```

class Vec4, the next morning

Well, at least we've tried it.

I guess SSE sucks.

class Vec4

- Wrong conclusion: SSE sucks because Vec4 is slow

class Vec4

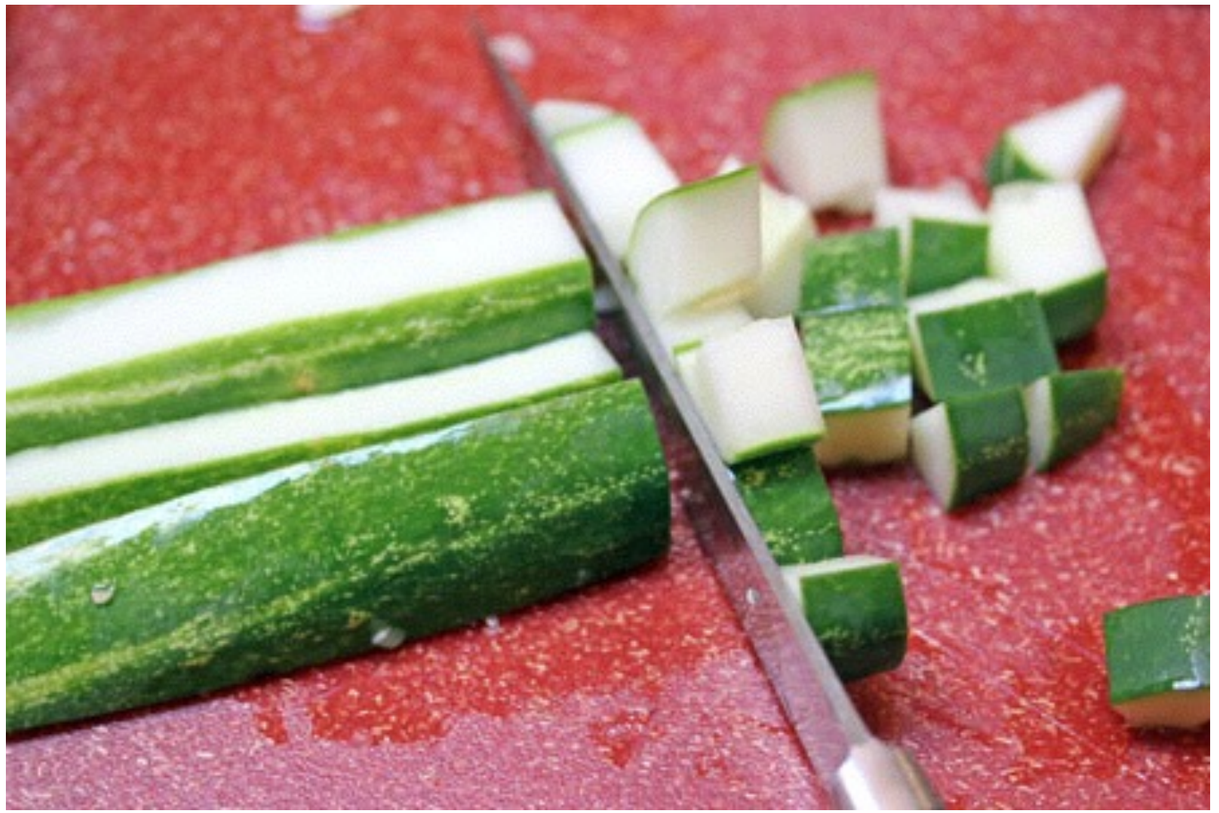
- Wrong conclusion: SSE sucks because Vec4 is slow
- Correct conclusion: The whole Vec4 idea sucks

Non-sucky SSE Dot Products (plural)

```
__m128 dx   = _mm_mul_ps(ax, bx);   // dx = ax * bx
__m128 dy   = _mm_mul_ps(ay, by);   // dy = ay * by
__m128 dz   = _mm_mul_ps(az, bz);   // dz = az * bz
__m128 dw   = _mm_mul_ps(aw, bw);   // dw = aw * bw

__m128 a0   = _mm_add_ps(dx, dy);   // a0 = dx + dy
__m128 a1   = _mm_add_ps(dz, dw);   // a1 = dz + dw
__m128 dots = _mm_add_ps(a0, a1);   // dots = a0 + a1
```

Non-sucky SSE Dot Products (plural)

```
__m128 dx
__m128 dy
__m128 dz
__m128 dw

__m128 a0
__m128 a1
__m128 dots
```



```
dx = ax * bx
dy = ay * by
dz = az * bz
dw = aw * bw

a0 = dx + dy
a1 = dz + dw
dots = a0 + a1
```

Don’t waste time on SSE classes

- Trying to abstract SOA hardware with AOS data
  - Doomed to be awkward & slow

Don’t waste time on SSE classes

- Trying to abstract SOA hardware with AOS data
  - Doomed to be awkward & slow
- SSE code wants to be free!
  - Best performance without wrappers or frameworks

Don’t waste time on SSE classes

- Trying to abstract SOA hardware with AOS data
  - Doomed to be awkward & slow
- SSE code wants to be free!
  - Best performance without wrappers or frameworks
- Just write small helper routines as needed

“It doesn’t fit our data layout”

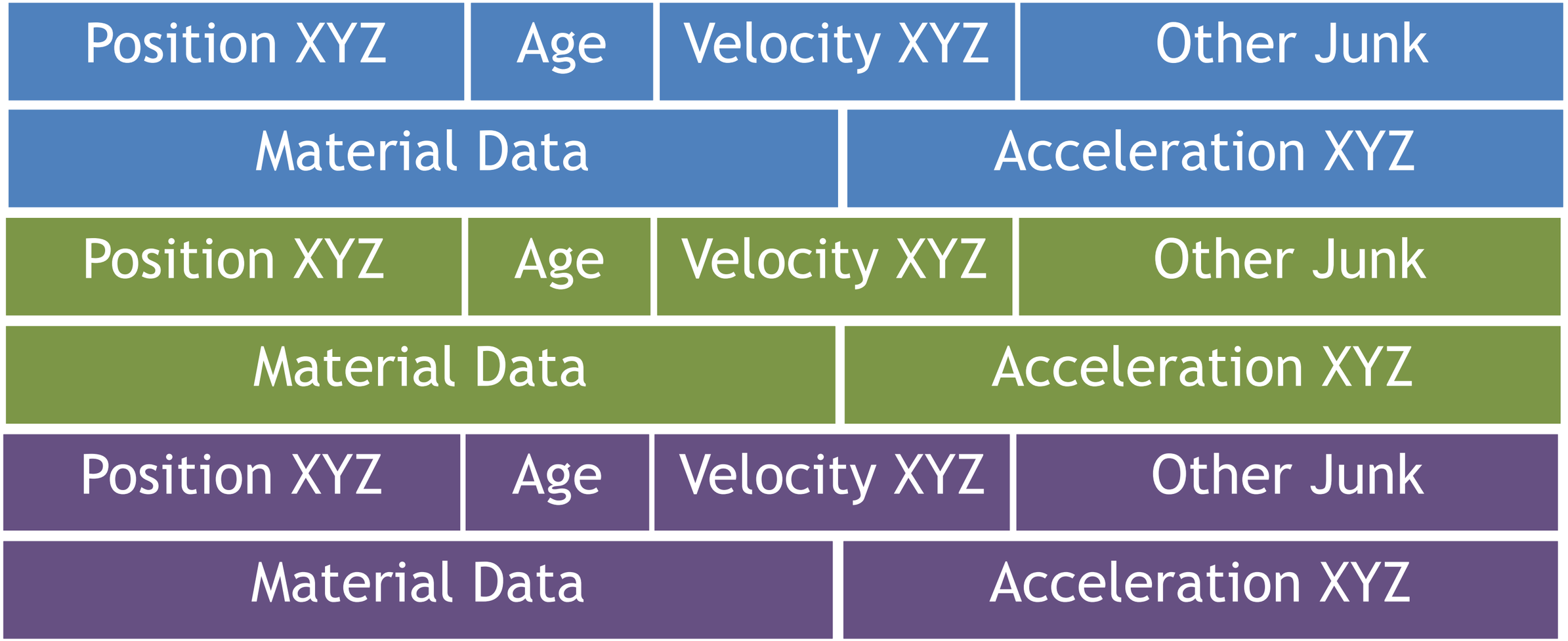
- void SpawnParticle(float pos[3], …);
  - Stored in struct Particle { float pos[3]; … }
  - Awkward to work with array of Particle in SSE

“It doesn’t fit our data layout”

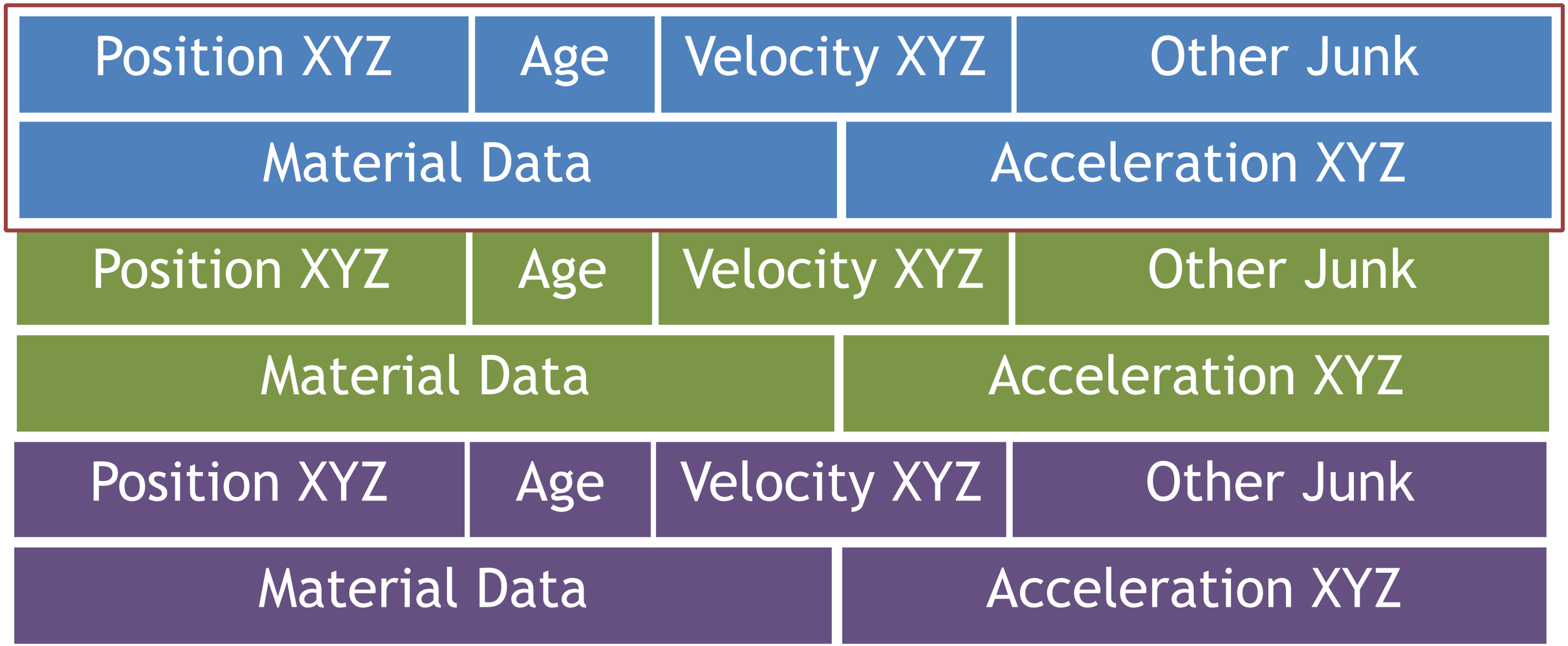
- void SpawnParticle(float pos[3], …);
  - Stored in struct Particle { float pos[3]; … }
  - Awkward to work with array of Particle in SSE

- So don’t do that
  - Keep the spawn function, change the memory layout
  - The problem is with struct Particle, not SSE

Struct Particle in memory (AOS)



Struct Particle in memory (AOS)



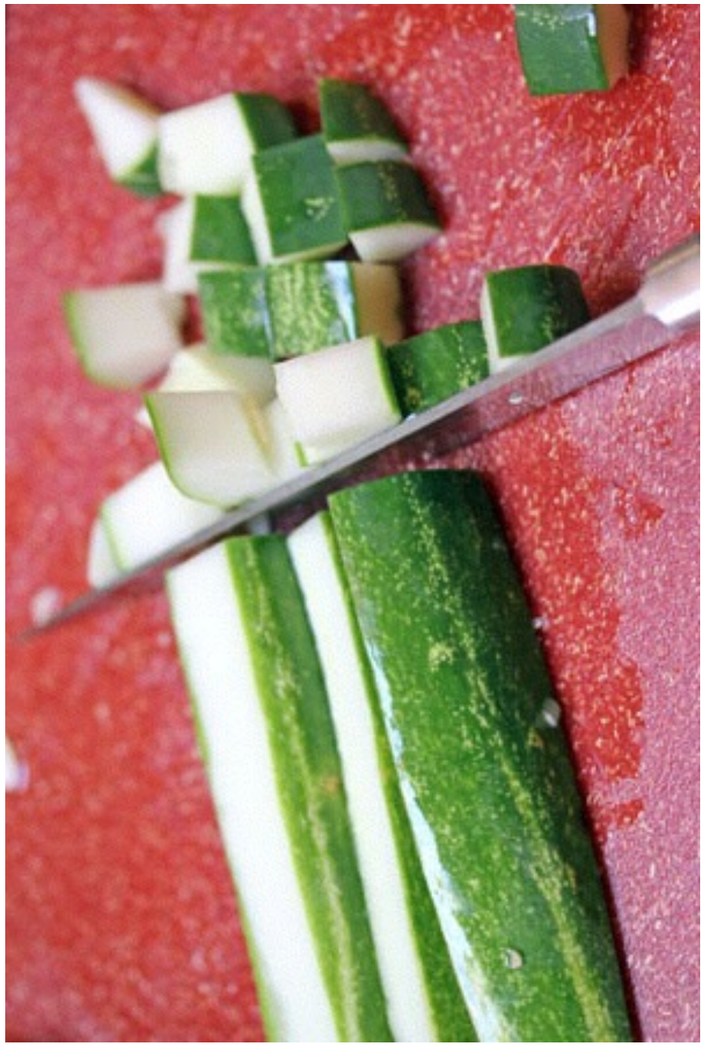
Particles in memory (SOA)

Particles in memory (SOA)

Particles in memory (SOA)

Particles in memory (SOA)

Pos X	Pos X	Pos X	Pos X	…
Pos Y	Pos Y	Pos Y	Pos Y	…
Pos Z	Pos Z	Pos Z	Pos Z	…
Age	Age	Age	Age	…
Vel X	Vel X	Vel X	Vel X	…
…	…	…	…	…



Data Layout Choices

- SOA form usually *much* better for SSE code
  - Maps naturally to instruction set
  - SOA SIMD code maps closely to scalar reference code

Data Layout Choices

- SOA form usually *much* better for SSE code
  - Maps naturally to instruction set
  - SOA SIMD code maps closely to scalar reference code
- AOS form usually better for scalar problems
  - Especially for lookup or indexing algorithms
  - Single cache miss to get at group of values

Data Layout Choices

- SOA form usually *much* better for SSE code
  - Maps naturally to instruction set
  - SOA SIMD code maps closely to scalar reference code
- AOS form usually better for scalar problems
  - Especially for lookup or indexing algorithms
  - Single cache miss to get at group of values
- Generate SOA data locally in transform if needed
  - Trade off SIMD efficiency by shuffling in/out

Case Study: Doors

- Doors to open themselves automatically
  - When actor of right “allegiance” is within some radius
  - Think Star Trek doors
  - Typical game problem

Case Study: Doors

- Doors to open themselves automatically
  - When actor of right “allegiance” is within some radius
  - Think Star Trek doors
  - Typical game problem
- Initially implemented as an OO solution
  - Started to pop on the performance radar
  - ~100 doors x ~30 characters to test against = 3,000 tests!

Original Door Update

```
void Door::Update(float dt)
{
  ActorList all_characters = GetAllCharacters();

  bool should_open = false;

  for (Actor* actor : all_characters) {
    if (AllegianceComponent* c = actor->FindComponent<AllegianceComponent>()) {
      if (c->GetAllegiance() == m_Allegiance) {
        if (VecDistanceSquared(a->GetPosition(), this->GetPosition()) < m_OpenDistanceSq) {
          should_open = true;
          break;
        }
      }
    }
  }
  ...
}
```

Original Door Update

Scalar by definition

```
void Door::Updat
{
  ActorList all_characters = GetAllCharacters();

  bool should_open = false;

  for (Actor* actor : all_characters) {
    if (AllegianceComponent* c = actor->FindComponent<AllegianceComponent>()) {
      if (c->GetAllegiance() == m_Allegiance) {
        if (VecDistanceSquared(a->GetPosition(), this->GetPosition()) < m_OpenDistanceSq) {
          should_open = true;
          break;
        }
      }
    }
  }
  ...
}
```

Original Door Update


```
void Door::Updat
{
  ActorList all_characters = GetAllCharacters();

  bool should_open = false;

  for (Actor* actor : all_characters) {
    if (AllegianceComponent* c = actor->FindComponent<AllegianceComponent>()) {
      if (c->GetAllegiance() == m_Allegiance) {
        if (VecDistanceSquared(a->GetPosition(), this->GetPosition()) < m_OpenDistanceSq) {
          should_open = true;
          break;
        }
      }
    }
  }
  ...
}
```

Original Door Update

Scalar by definition

Repeated work

Multiple L2 misses

```
void Door::Updat
{
  ActorList all_characters = GetAllCharacters();

  bool should_open = false;

  for (Actor* actor : all_characters) {
    if (AllegianceComponent* c = actor->FindComponent<AllegianceComponent>()) {
      if (c->GetAllegiance() == m_Allegiance) {
        if (VecDistanceSquared(a->GetPosition(), this->GetPosition()) < m_OpenDistanceSq) {
          should_open = true;
          break;
        }
      }
    }
  }
  ...
}
```

Original Door Update

```
void Door::Updat
{
  ActorList all_characters = GetAllCharacters();

  bool                                    e;

  for (Actor* actor : all_characters) {
    if (AllegianceComponent* c = actor->FindComponent<AllegianceComponent>()) {
      if (c->GetAllegiance() == m_Allegiance) {
        if (VecDistanceSquared(a->GetPosition(), this->GetPosition()) < m_OpenDistanceSq) {
          should_open = true;
          break;
        }
      }
    }
  }
  ...
}
```

Scalar by definition

Repeated work

Not using all data

Multiple L2 misses

Original Door Update

```
void Door::Updat
{
  ActorList all_characters = GetAllCharacters();

  bool                                    e;

  for (Actor* actor : all_characters) {
    if (AllegianceComponent* c = actor->FindComponent<AllegianceComponent>()) {
      if (c->GetAllegiance() == m_Allegiance) {
        if (VecDistanceSquared(a->GetPosition(), this->GetPosition()) < m_OpenDistanceSq) {
          should_open = true;
          break;
        }
      }
    }
  }
  ...
}
```

Scalar by definition

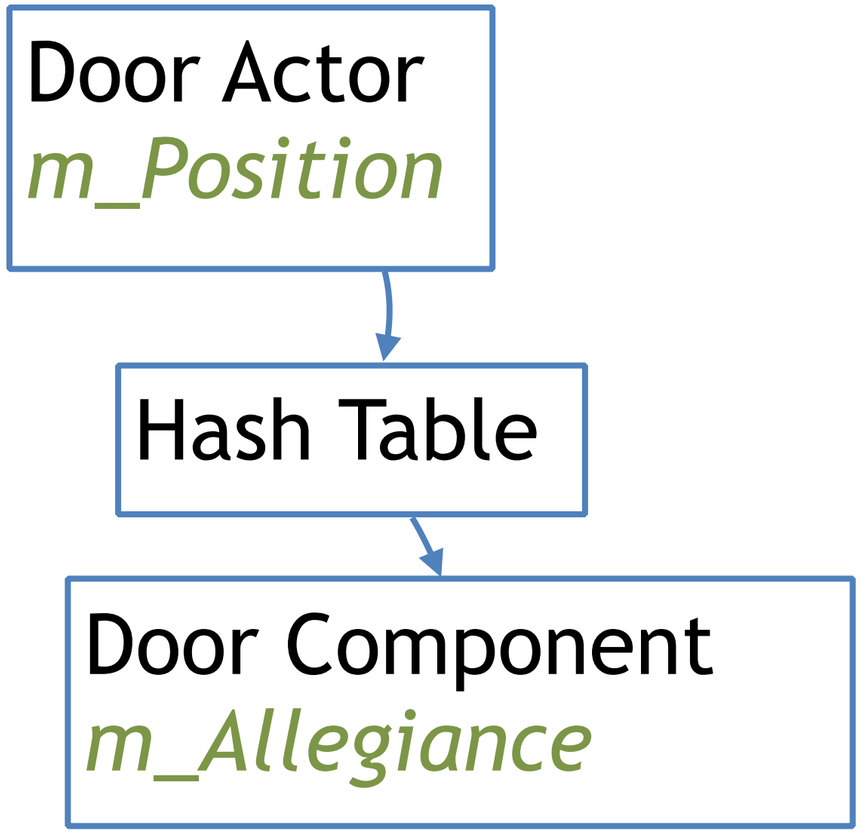
Repeated work

Not using all data

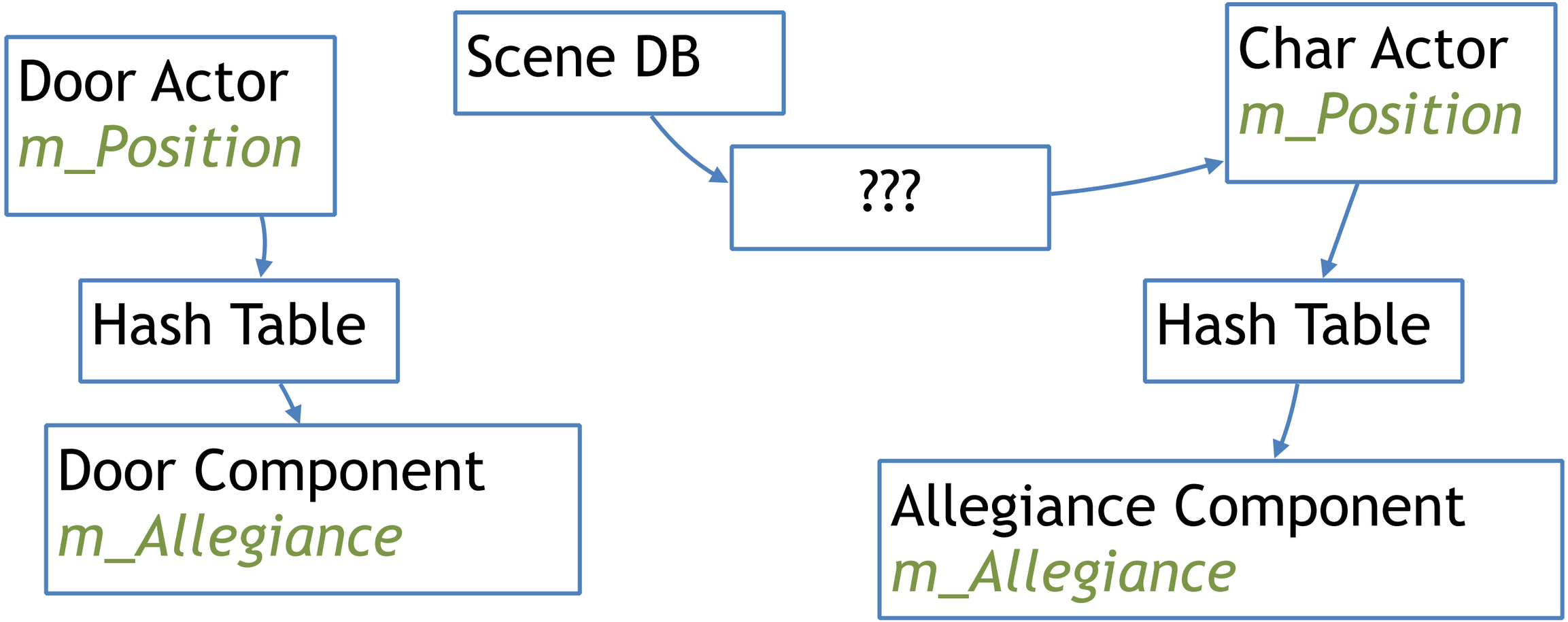
Multiple L2 misses

Scalar compute

Input Data in Original Update



Input Data in Original Update



Input Data in Original Update

Door Actor
m_Positio

Hash Ta

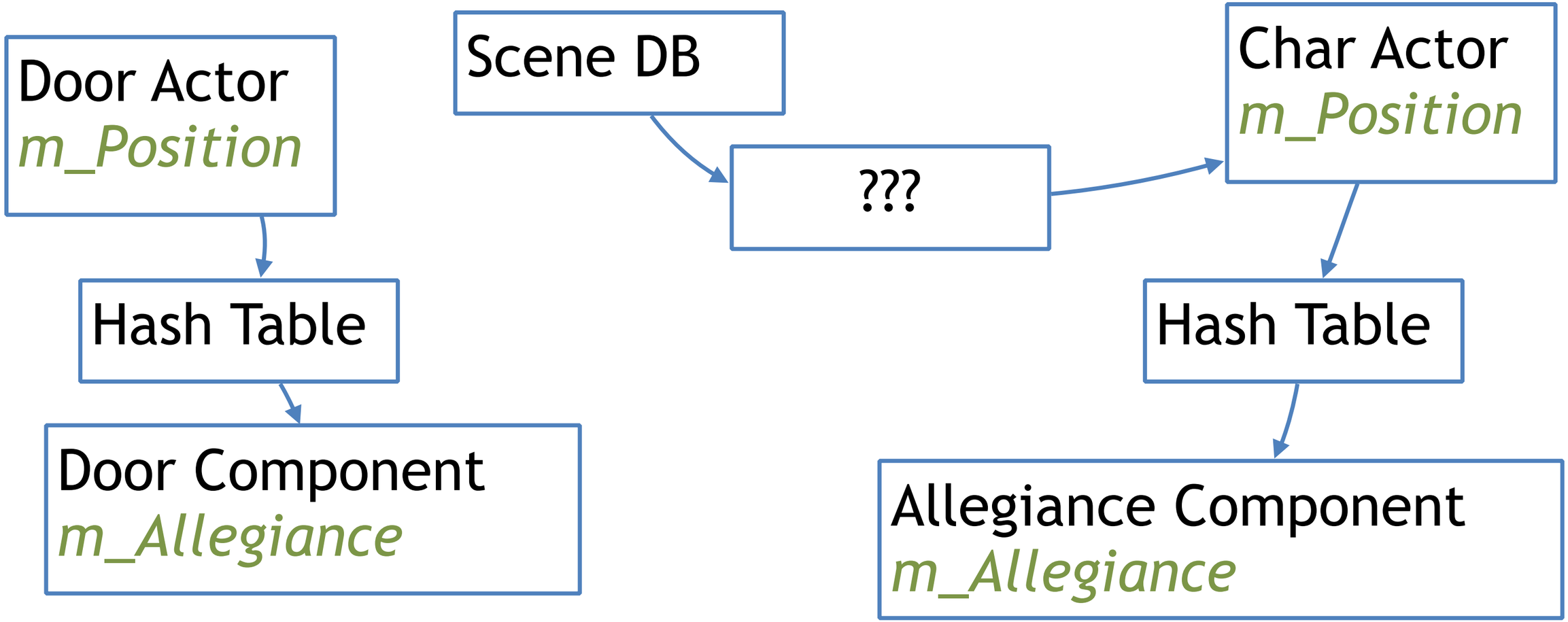
Door Con
m_Allegi

Char Actor
m_Position

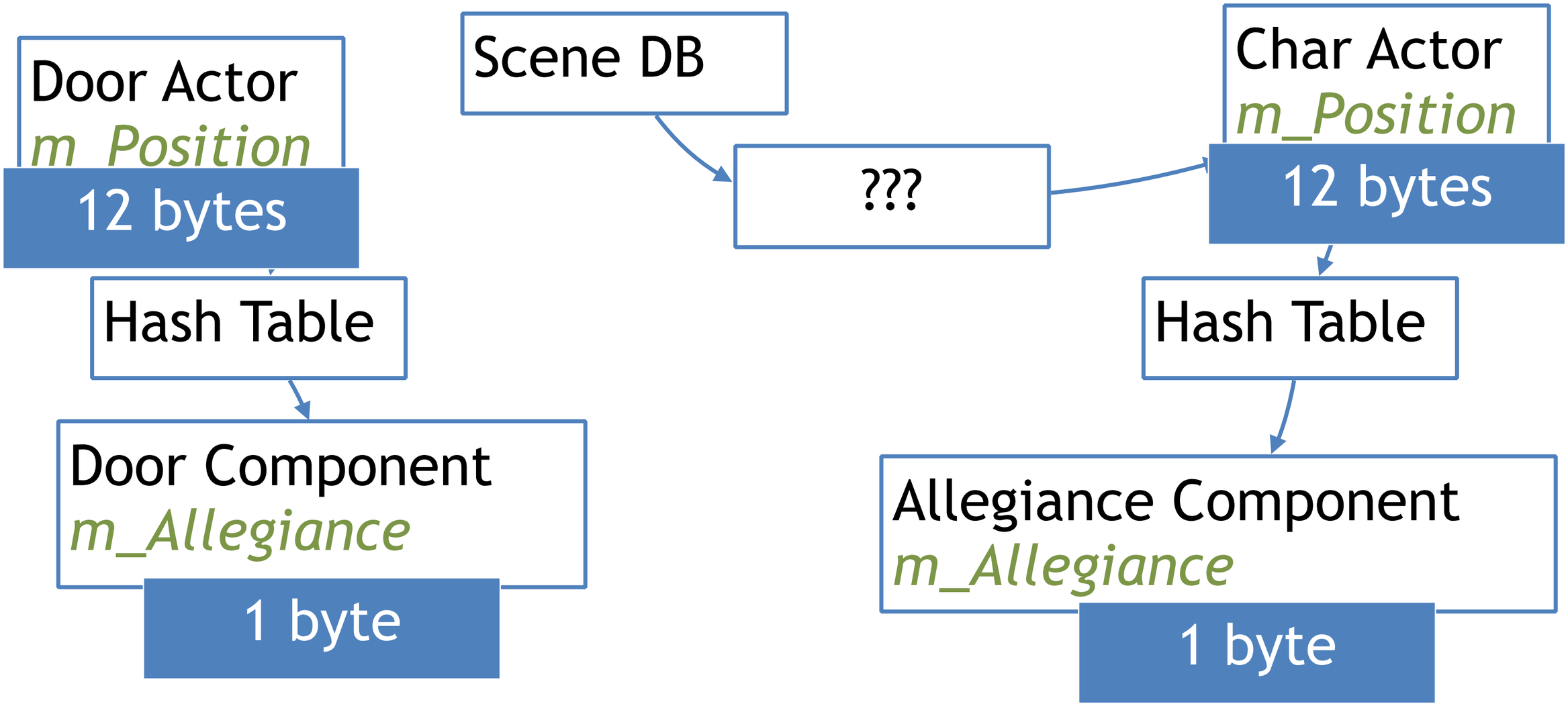
Hash Table

Component
nce

Input Data in Original Update



Input Data in Original Update



What is it actually computing?

```
for each door:
  door.should_be_open = 0
  for each character:
    if InRadius(…) and door.team == char.team:
      door.should_be_open = 1
```

What's the radius test computing?

- Tests if a point is within a sphere
  - Inputs: Two points x0,y0,z0 and x1,y1,z1 + a squared radius
  - Outputs: yes or no
  - (It's just trying to avoid a square root)

What's the radius test computing?

- Tests if a point is within a sphere
  - Inputs: Two points x0,y0,z0 and x1,y1,z1 + a squared radius
  - Outputs: yes or no
  - (It's just trying to avoid a square root)

$$(x_0-x_1)^2 + (y_0-y_1)^2 + (z_0-z_1)^2 <= r^2$$

What's the radius test computing?

- Tests if a point is within a sphere
  - Inputs: Two points x0,y0,z0 and x1,y1,z1 + a squared radius
  - Outputs: yes or no
  - (It's just trying to avoid a square root)

$$(x_0-x_1)^2 + (y_0-y_1)^2 + (z_0-z_1)^2 <= r^2$$

- OK, now we understand all the pieces

SIMD Prep Work

- Move door data to central place
  - Really just a bag of values in SOA form
  - Good approach as doors are rarely created & destroyed
  - Each door has an index into central data stash

SIMD Prep Work

- Move door data to central place
  - Really just a bag of values in SOA form
  - Good approach as doors are rarely created & destroyed
  - Each door has an index into central data stash

- Build actor tables locally in update
  - Once per update, not 100 times
  - Stash in simple array on stack (alloca for variable size)

Door Update Data Design

```
// In memory, SOA
struct DoorData {
  uint32_t  Count;
  float    *X;
  float    *Y;
  float    *Z;
  float    *RadiusSq;
  uint32_t *Allegiance;
  // Output data
  uint32_t *ShouldBeOpen;
} s_Doors;
```

```
// On the stack, AOS
struct CharData {
  float    X;
  float    Y;
  float    Z;
  uint32_t Allegiance;
} c[MAXCHARS];
```

SIMD Door Update

- New update does all doors in one go
  - Test 4 doors vs 1 actor in inner loop

SIMD Door Update

- New update does all doors in one go
  - Test 4 doors vs 1 actor in inner loop
- Massive benefits from the data layout
  - All compute naturally falls out as SIMD operations

Outer Loop Prologue

```
for (int d = 0; d < door_count; d += 4) {

  __m128  dx    = _mm_load_ps(&s_Doors.X[d]);
  __m128  dy    = _mm_load_ps(&s_Doors.Y[d]);
  __m128  dz    = _mm_load_ps(&s_Doors.Z[d]);
  __m128  dr    = _mm_load_ps(&s_Doors.RadiusSq[d]);
  __m128i da    = _mm_load_si128((__m128i*) &s_Doors.Allegiance[d]);

  __m128i state = _mm_setzero_si128();
```

Load attributes for 4 doors, clear 4 “open” accumulators

Inner Loop Prologue

```
...

for (int cc = 0; cc < char_count; ++cc) {
  __m128  char_x = _mm_broadcast_ss(&c[cc].x);
  __m128  char_y = _mm_broadcast_ss(&c[cc].y);
  __m128  char_z = _mm_broadcast_ss(&c[cc].z);
  __m128i char_a = _mm_set1_epi32(c[cc].allegiance);

  ...
```

Load attributes for 1 character, broadcast to all 4 lanes

Inner Loop Math

```
...
__m128  ddx     = _mm_sub_ps(dx, char_x);
__m128  ddy     = _mm_sub_ps(dy, char_y);
__m128  ddz     = _mm_sub_ps(dz, char_z);
__m128  dtx     = _mm_mul_ps(ddx, ddx);
__m128  dty     = _mm_mul_ps(ddy, ddy);
__m128  dtz     = _mm_mul_ps(ddz, ddz);
__m128  dst     = _mm_add_ps(_mm_add_ps(dtx, dty), dtz);
...
```

Compute squared distance between character & the 4 doors

Inner Loop Epilogue

```
    ...
    __m128  rmask  = _mm_cmple_ps(dst, dr);
    __m128i amask  = _mm_cmpeq_epi32(da, char_a);
    __m128i mask   = _mm_and_si128(_mm_castps_si128(amask), rmask);

            state  = _mm_or_si128(mask, state);
}
...
```

Compare against door open radii AND allegiance => OR into state

Outer Loop Epilogue

```
    ...

    _mm_store_si128((__m128i*) &s_Doors.ShouldBeOpen[d], state);

}
```

Store “should open” for these 4 doors, ready for next group of 4

```
for (int d = 0; d < door_count; d += 4) {
  __m128  dx    = _mm_load_ps(&s_Doors.X[d]);
  __m128  dy    = _mm_load_ps(&s_Doors.Y[d]);
  __m128  dz    = _mm_load_ps(&s_Doors.Z[d]);
  __m128  dr    = _mm_load_ps(&s_Doors.RadiusSq[d]);
  __m128i da    = _mm_load_si128((__m128i*) &s_Doors.Allegiance[d]);
  __m128i state = _mm_setzero_si128();

  for (int cc = 0; cc < char_count; ++cc) {
    __m128  char_x = _mm_broadcast_ss(&c[cc].x);
    __m128  char_y = _mm_broadcast_ss(&c[cc].y);
    __m128  char_z = _mm_broadcast_ss(&c[cc].z);
    __m128i char_a = _mm_set1_epi32(c[cc].allegiance);

    __m128  ddx    = _mm_sub_ps(dx, char_x);
    __m128  ddy    = _mm_sub_ps(dy, char_y);
    __m128  ddz    = _mm_sub_ps(dz, char_z);
    __m128  dtx    = _mm_mul_ps(ddx, ddx);
    __m128  dty    = _mm_mul_ps(ddy, ddy);
    __m128  dtz    = _mm_mul_ps(ddz, ddz);
    __m128  dst    = _mm_add_ps(_mm_add_ps(dtx, dty), dtz);

    __m128  rmask  = _mm_cmple_ps(dst, dr);
    __m128i amask  = _mm_cmpeq_epi32(da, char_a);
    __m128i mask   = _mm_and_si128(_mm_castps_si128(amask), rmask);

            state  = _mm_or_si128(mask, state);
  }

  _mm_store_si128((__m128i*) &s_Doors.ShouldBeOpen[d], state);
}
```

Inner Loop Code Generation

```
vbroadcastss    xmm6, dword ptr [rcx-8]
vbroadcastss    xmm7, dword ptr [rcx-4]
vbroadcastss    xmm1, dword ptr [rcx]
vbroadcastss    xmm2, dword ptr [rcx+4]
vsubps          xmm6, xmm8, xmm6
vsubps          xmm7, xmm9, xmm7
vsubps          xmm1, xmm3, xmm1
vmulps          xmm6, xmm6, xmm6
vmulps          xmm7, xmm7, xmm7
vmulps          xmm1, xmm1, xmm1
vaddps          xmm6, xmm6, xmm7
vaddps          xmm1, xmm6, xmm1
vcmpps          xmm1, xmm1, xmm4, 2
vpcmpeqd        xmm2, xmm5, xmm2
vpand           xmm1, xmm2, xmm1
vpor            xmm0, xmm1, xmm0
add             rcx, 10h
dec             edi
jnz             .loop
```

- ~6 cycles per 4 door/actor tests
- 100 doors x 30 actors = ~4500 c

Door Results

- 20-100x speedup
  - Even more possible now that we can reason about the data

Door Results

- 20-100x speedup
  - Even more possible now that we can reason about the data
- Brute force SIMD for “reasonable # of things”
  - Lots of “reasonable # of things” problems in a game!
  - Removing cache misses + SIMD ALU can be huge win

Door Results

- 20-100x speedup
  - Even more possible now that we can reason about the data
- Brute force SIMD for “reasonable # of things”
  - Lots of “reasonable # of things” problems in a game!
  - Removing cache misses + SIMD ALU can be huge win
- Solves “death by a thousand cuts” problems
  - This type of transform typically takes it off the radar

Techniques & Tricks

- Need to cope with messy data & constraints
  - Want largest possible scope for SIMD throughout codebase

Techniques & Tricks

- Need to cope with messy data & constraints
  - Want largest possible scope for SIMD throughout codebase
- We'll look at two tricks
  - Left packing
  - Dynamic mask generation

Techniques & Tricks

- Need to cope with messy data & constraints
  - Want largest possible scope for SIMD throughout codebase
- We'll look at two tricks
  - Left packing
  - Dynamic mask generation
- SSSE3+ greatly expands the trick repertoire
  - We'll start with SSSE3+ features
  - We'll come back to SSE2 (the dark ages) after that..

Problem: Filtering Data

- Discarding data while streaming
  - Not a 1:1 relationship between input and output
  - N inputs, M outputs, M <= N
  - Not writing multiple of SIMD register width to output!
- Want to express as SIMD kernel, but how?

Scalar Filtering

```
int FilterFloats_Reference(const float input[], float output[],
                           int count, float limit)
{
  float *outputp = output;

  for (int i = 0; i < count; ++i) {
    if (input[i] >= limit)
      *outputp++ = input[i];
  }

  return (int) (outputp - output);
}
```

Scalar Filtering

```
int FilterFloats_Reference(const float input[], float output[],
                           int count, float limit)
{
  float *outputp = output;

  for (int i = 0; i < count; ++i) {
    if (input[i] >= limit)
      *outputp++ = input[i];
  }

  return (int) (outputp - output);
}
```

SIMD Filtering Skeleton..

```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

SIMD Filtering Skeleton..

```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

Load 4 floats

SIMD Filtering Skeleton..

```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

Perform 4 compares => mask

SIMD Filtering Skeleton..

```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

Left-pack valid elements to front of register

SIMD Filtering Skeleton..

```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

Store unaligned to current output position

SIMD Filtering Skeleton..

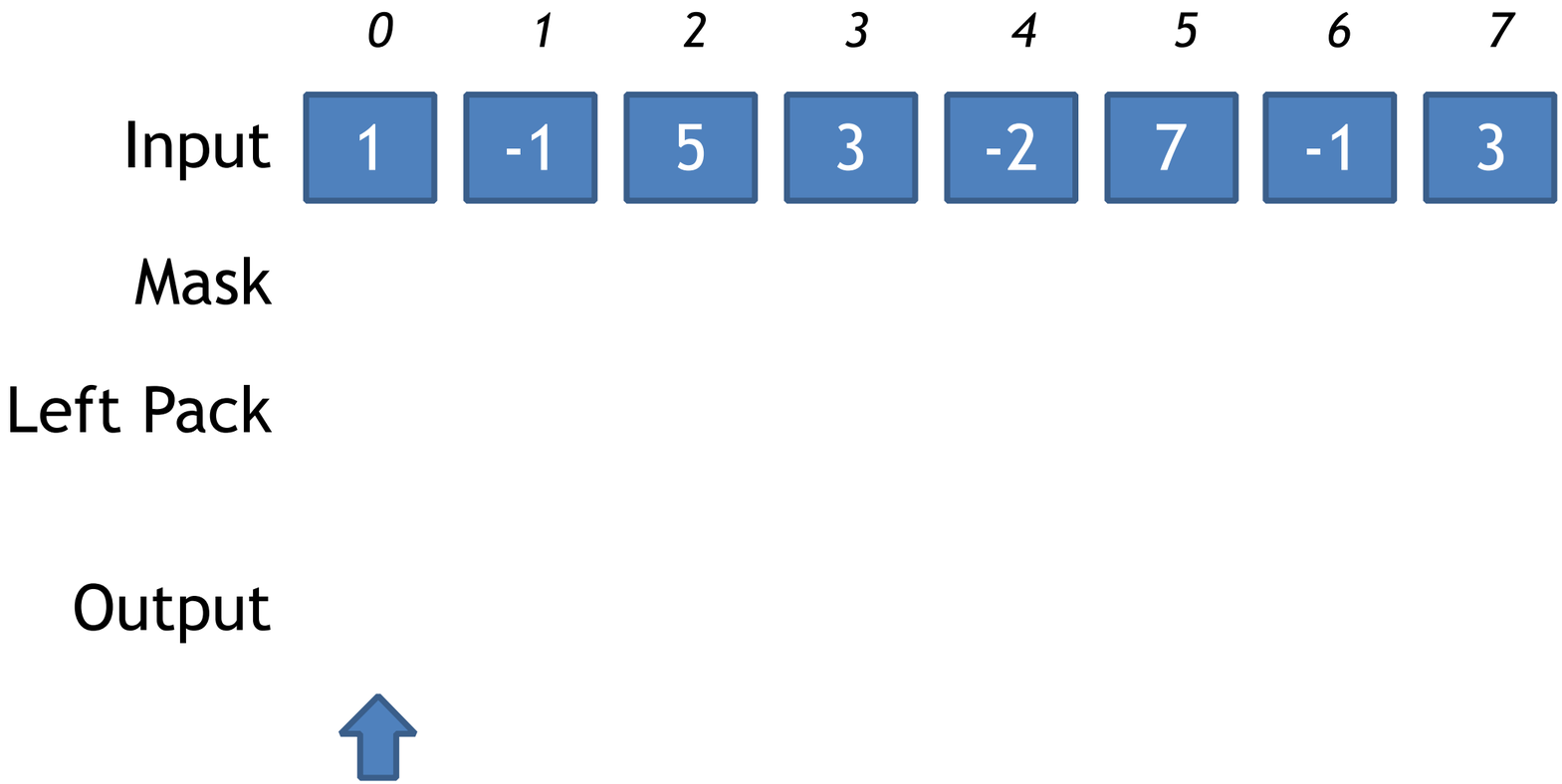
```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

Advance output position based on mask

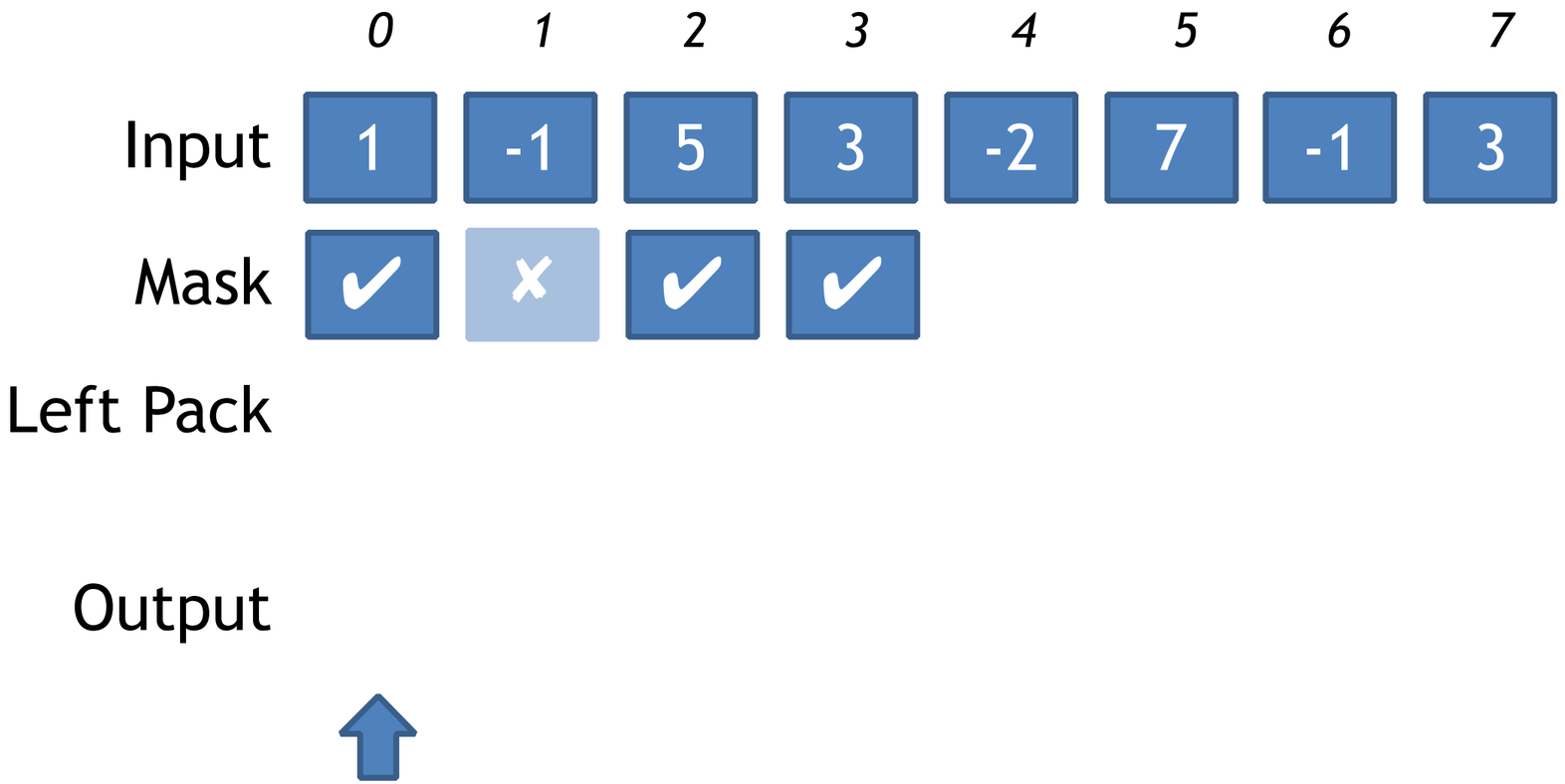
SIMD Filtering Skeleton..

```
for (int i = 0; i < count; i += 4) {
  __m128 val      = _mm_load_ps(input + i);
  __m128 mask     = _mm_cmpge_ps(val, _mm_set1_ps(limit));

  __m128 result   = LeftPack(mask, val);

  _mm_storeu_ps(output, result);

  output += _popcnt(_mm_movemask_ps(mask));
}
```

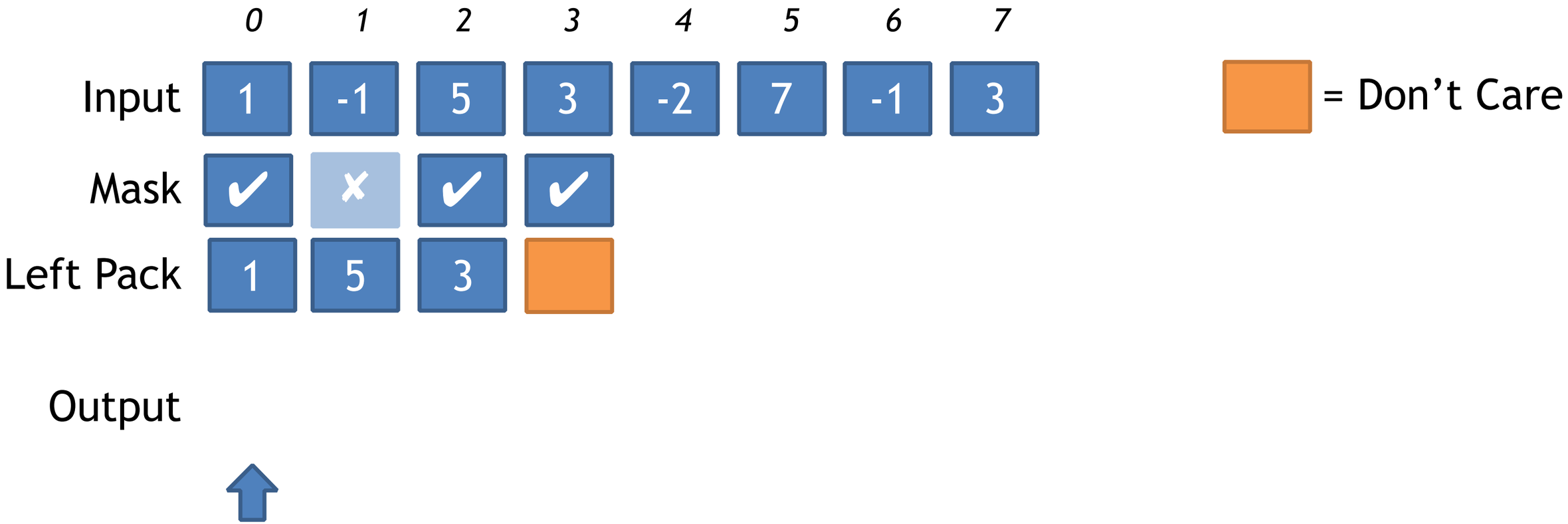
Left Packing Problem (4-wide, limit=0)



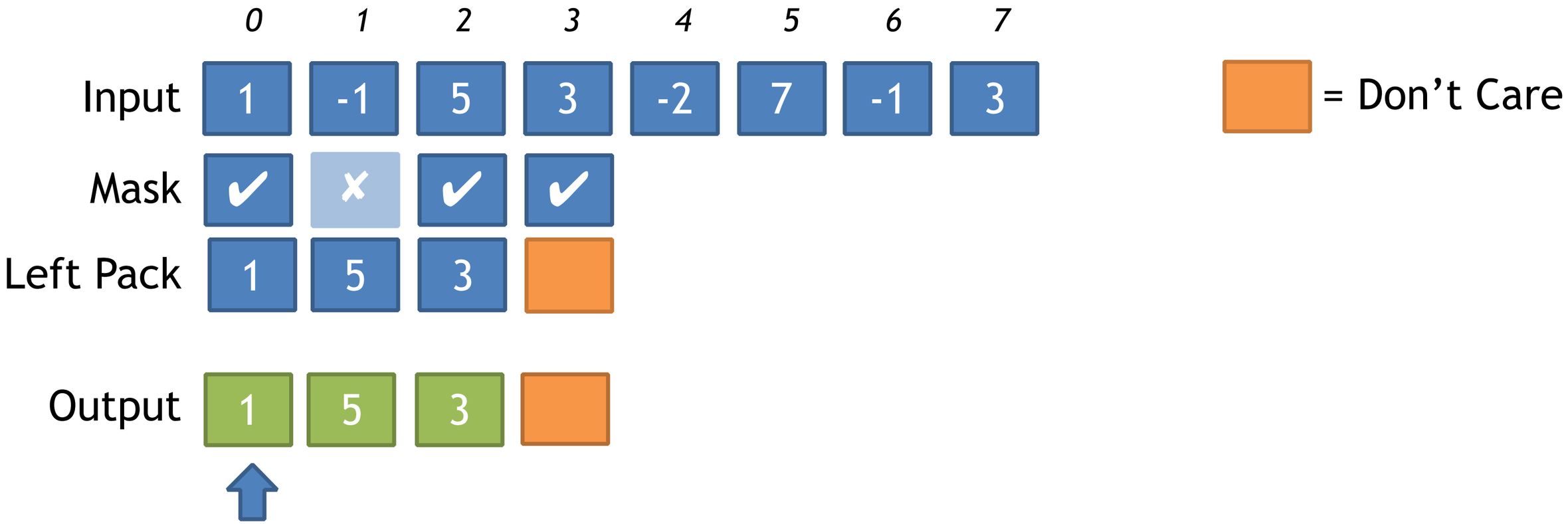
Left Packing Problem (4-wide, limit=0)



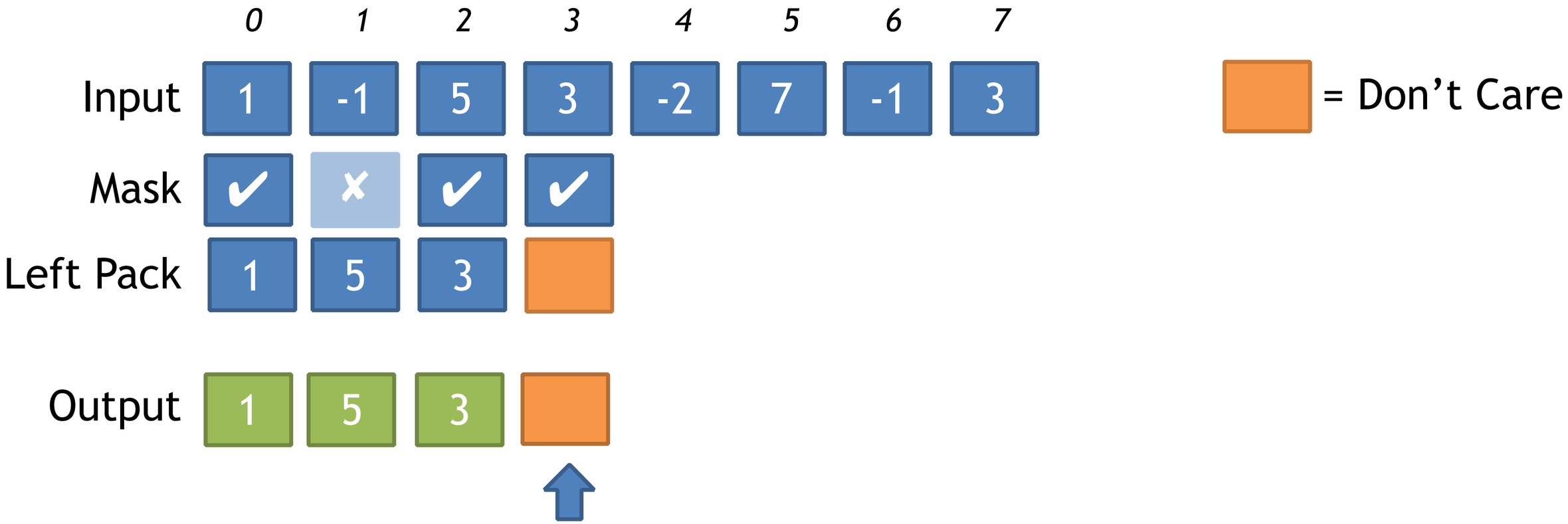
Left Packing Problem (4-wide, limit=0)



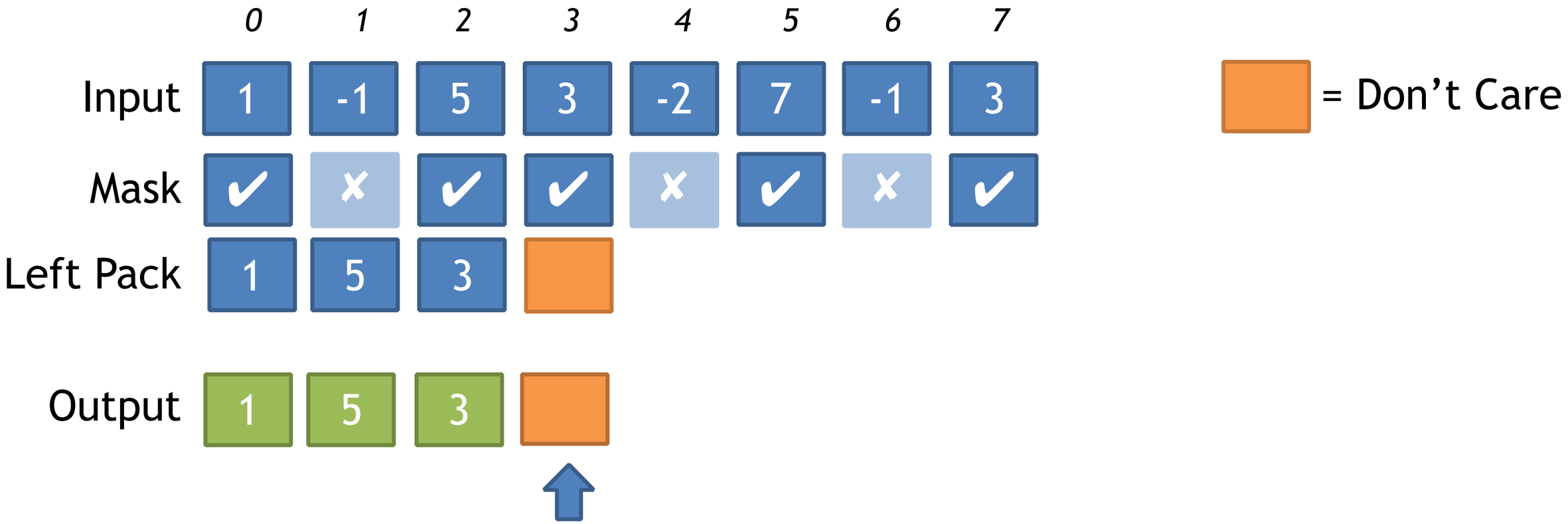
Left Packing Problem (4-wide, limit=0)



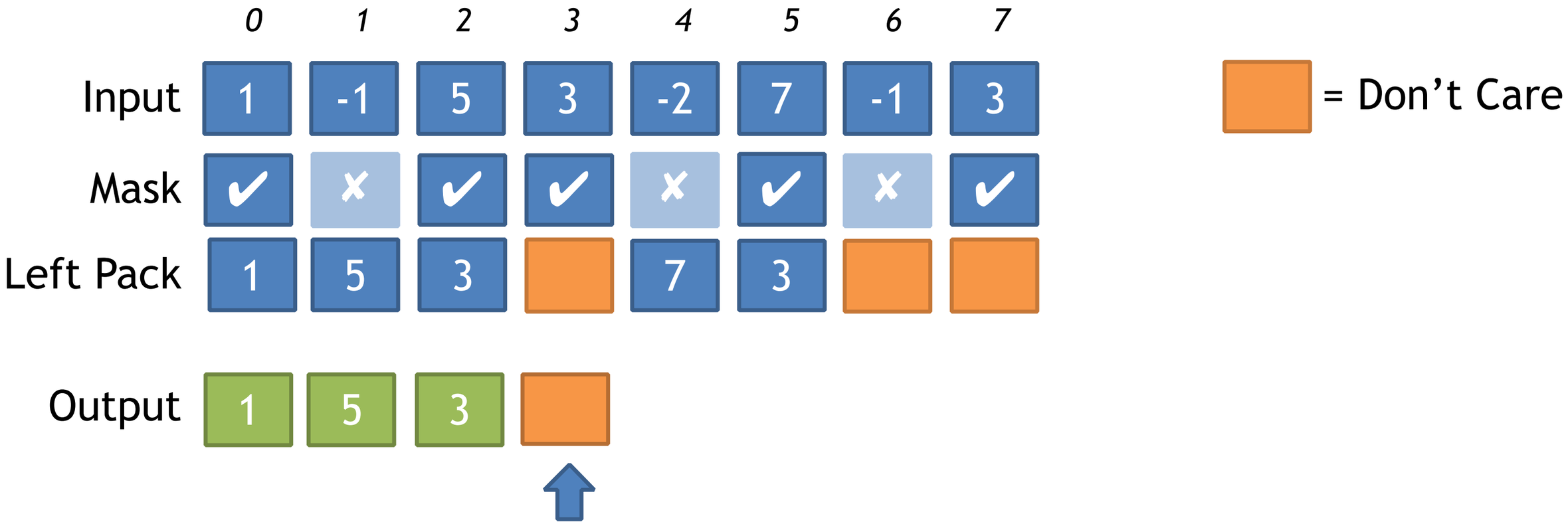
Left Packing Problem (4-wide, limit=0)



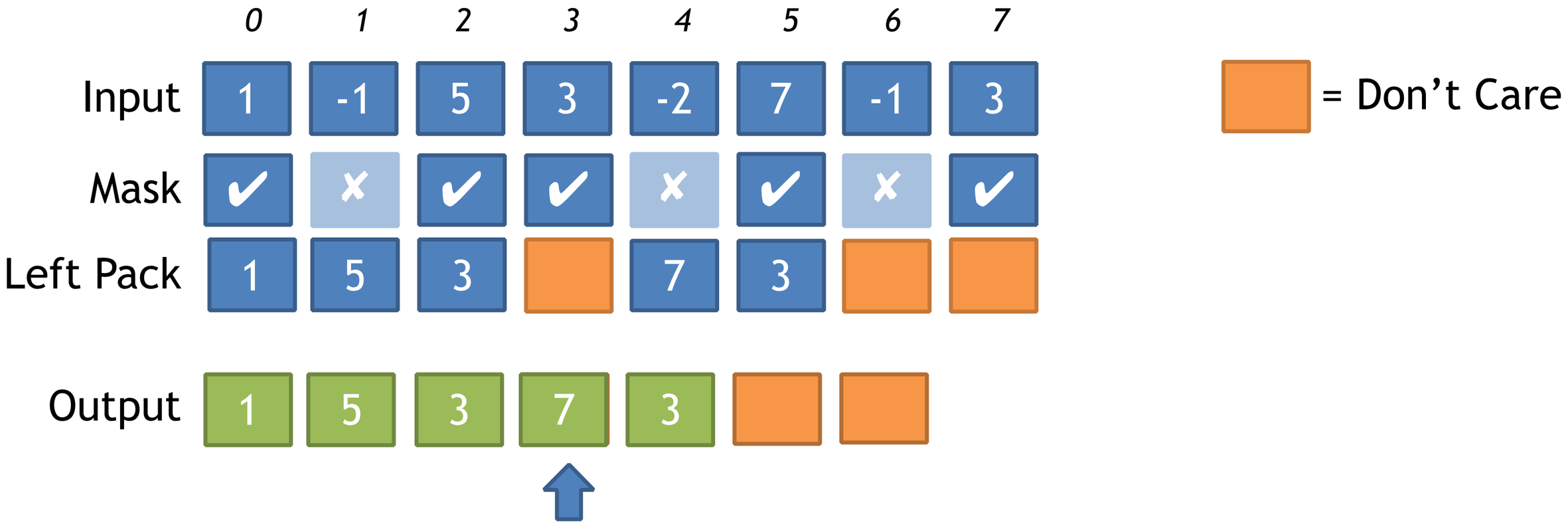
Left Packing Problem (4-wide, limit=0)



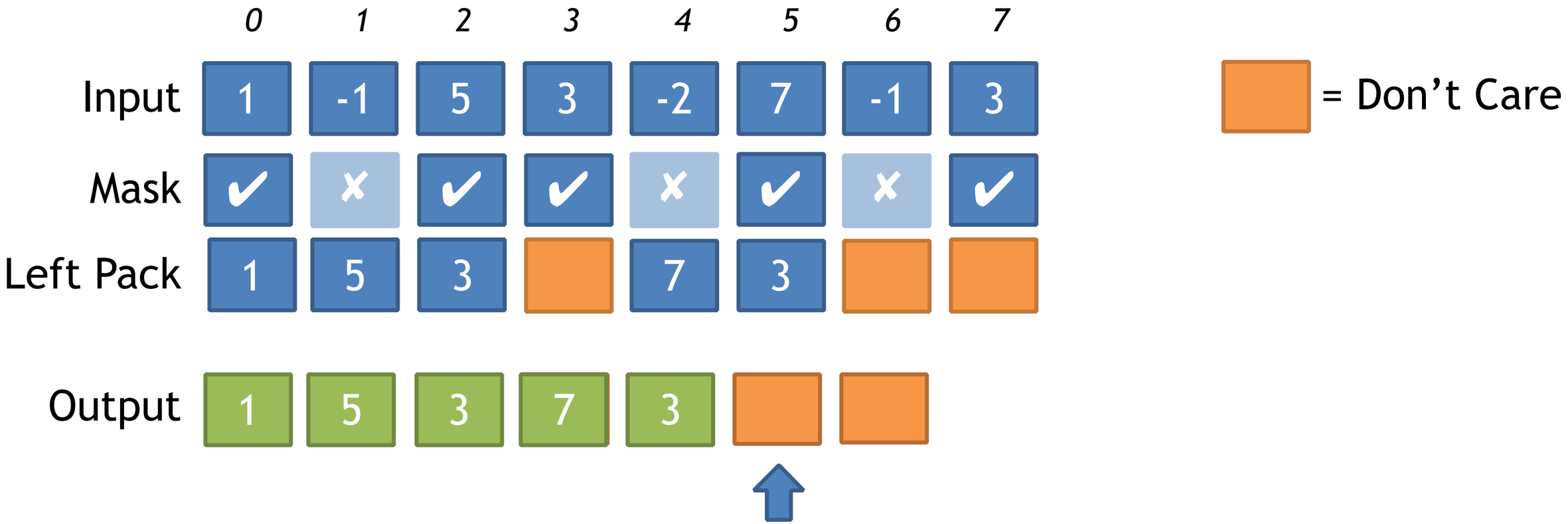
Left Packing Problem (4-wide, limit=0)



Left Packing Problem (4-wide, limit=0)



Left Packing Problem (4-wide, limit=0)



Left Packing (SSSE3+)

- `_mm_movemask_ps()` = bit mask of valid lanes
  - Value in the range 0-15

Left Packing (SSSE3+)

- `_mm_movemask_ps()` = bit mask of valid lanes
  - Value in the range 0-15
- Leverage indirect shuffle via PSHUFB
  - a.k.a `_mm_shuffle_epi8()`

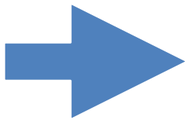
Left Packing (SSSE3+)

- `_mm_movemask_ps()` = bit mask of valid lanes
  - Value in the range 0-15
- Leverage indirect shuffle via PSHUFB
  - a.k.a `_mm_shuffle_epi8()`
- Lookup table of 16 shuffles (4-wide case)
  - Each shuffle moves valid lanes to the left
  - Need 16 x 16 = 256 bytes (4 cache lines) of LUT data

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Left Packing Code (SSSE3+)



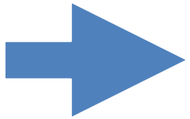
```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```



04	05	06	07	0C	0D	0E	0F	80	80	80	80	80	80	80	80

= YW00

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Left Packing Code (SSSE3+)

```
__m128i LeftPack_SSSE3(__m128 mask, __m128 val)
{
  // Move 4 sign bits of mask to 4-bit integer value.
  int    mask = _mm_movemask_ps(mask);

  // Select shuffle control data
  __m128i shuf_ctrl = _mm_load_si128(&shufmasks[mask]);

  // Permute to move valid values to front of SIMD register
  __m128i packed    = _mm_shuffle_epi8(_mm_castps_si128(val), shuf_ctrl);

  return packed;
}
```

Problem: Dynamic Low Masking

- Want mask that isolates ***n*** lower bits per lane
  - Useful for dynamic fixed point & many other things
- Easy in scalar code: `(1 << n) - 1`
- No straight forward SSE equivalent
  - ***n*** varies across SIMD register
  - No instruction to do variable shifts per lane

LUT Low Mask Generation, n = 17

LUT

0	00
	01
	03
	07
	0F
	1F
	3F
	7F
8	FF

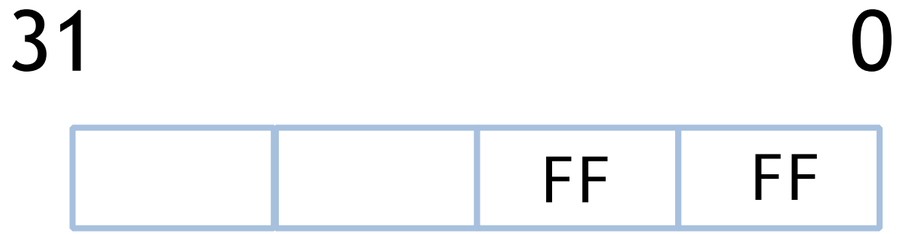
LUT Low Mask Generation, n = 17

Index 0: Clamp(n , 0, 8) = 8

LUT

0	00
	01
	03
	07
	0F
	1F
	3F
	7F
8	FF

LUT Low Mask Generation, n = 17

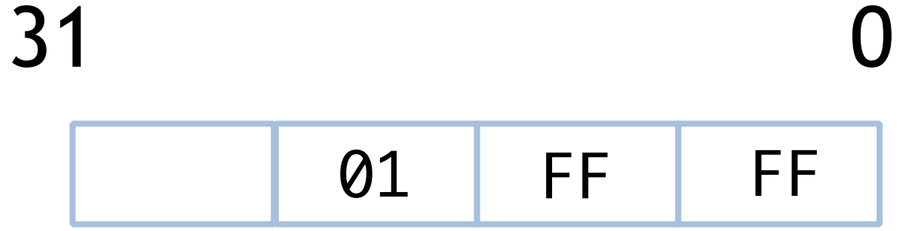



```
Index 0: Clamp(n   , 0, 8)     = 8
Index 1: Clamp(n-8 , 0, 8)     = 8
```

LUT

0	00
	01
	03
	07
	0F
	1F
	3F
	7F
8	FF

LUT Low Mask Generation, n = 17

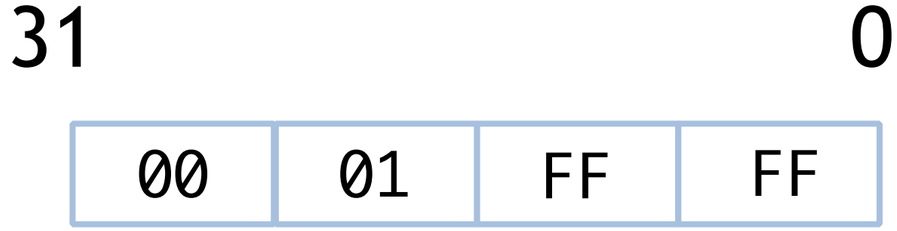



```
Index 0: Clamp(n   , 0, 8)    = 8
Index 1: Clamp(n-8 , 0, 8)    = 8
Index 2: Clamp(n-16, 0, 8)    = 1
```

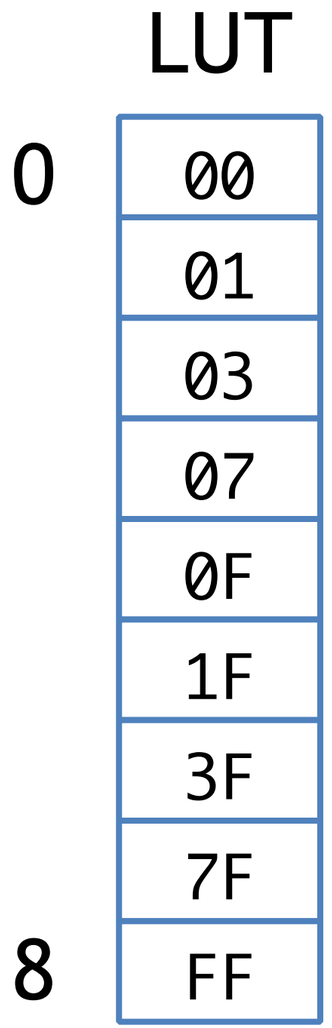
LUT

0	00
	01
	03
	07
	0F
	1F
	3F
	7F
8	FF

LUT Low Mask Generation, n = 17




```
Index 0: Clamp(n   , 0, 8)    = 8
Index 1: Clamp(n-8 , 0, 8)    = 8
Index 2: Clamp(n-16, 0, 8)    = 1
Index 3: Clamp(n-24, 0, 8)    = 0
```



Dynamic Low Masking (SSSE3+)

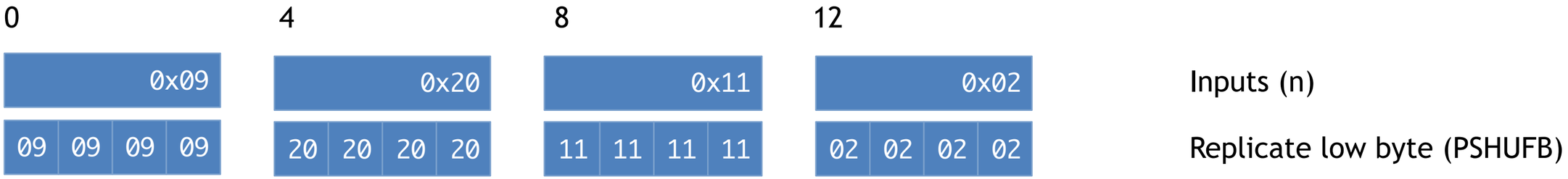
- PSHUFB can be used as nibble->byte lookup
  - 16 parallel lookups
  - Works for low masking because we only have 9 cases
- Index clamping
  - Use saturated addition & subtraction
  - Compute all 16 byte lookup indices in parallel

Dynamic Low Masking in Action

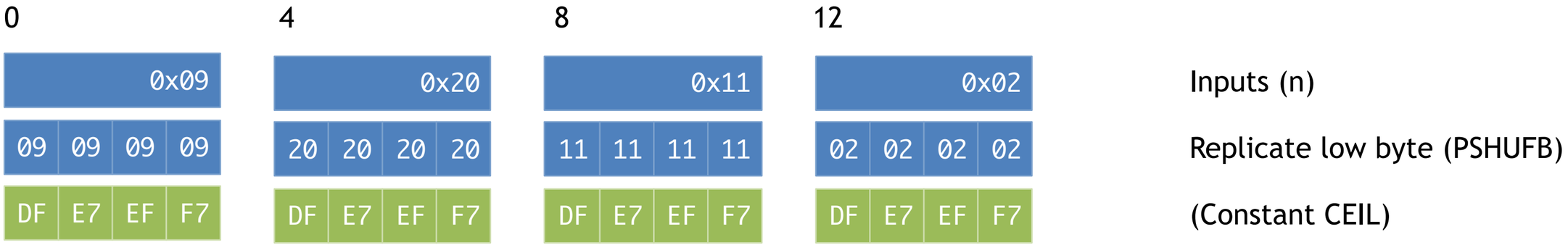
0 4 8 12

Dynamic Low Masking in Action

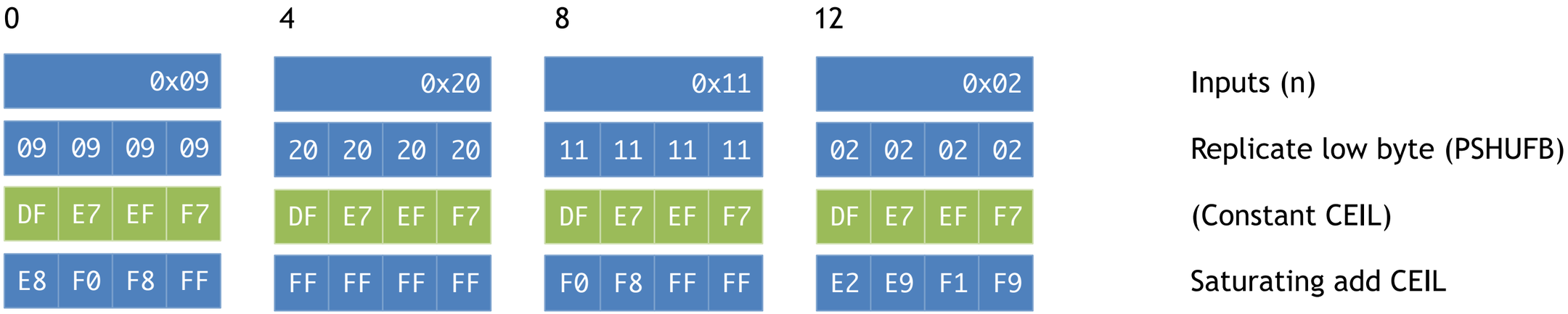
Dynamic Low Masking in Action



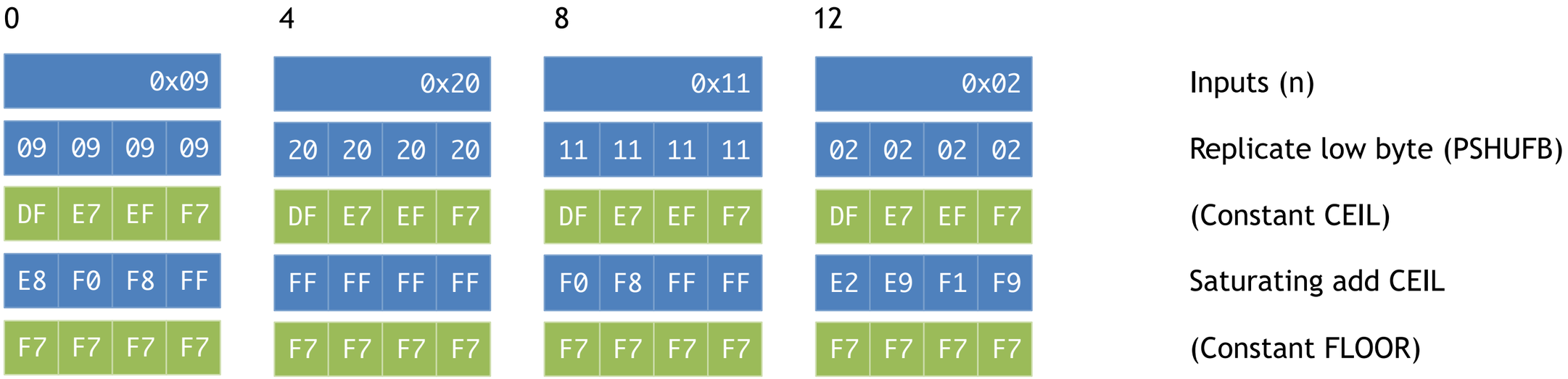
Dynamic Low Masking in Action



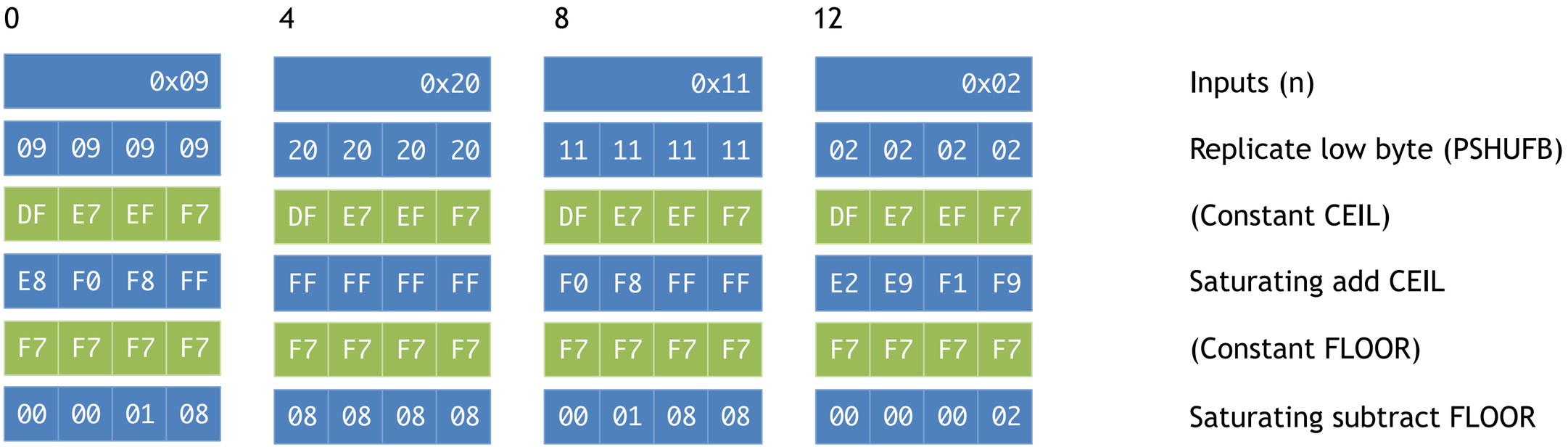
Dynamic Low Masking in Action



Dynamic Low Masking in Action



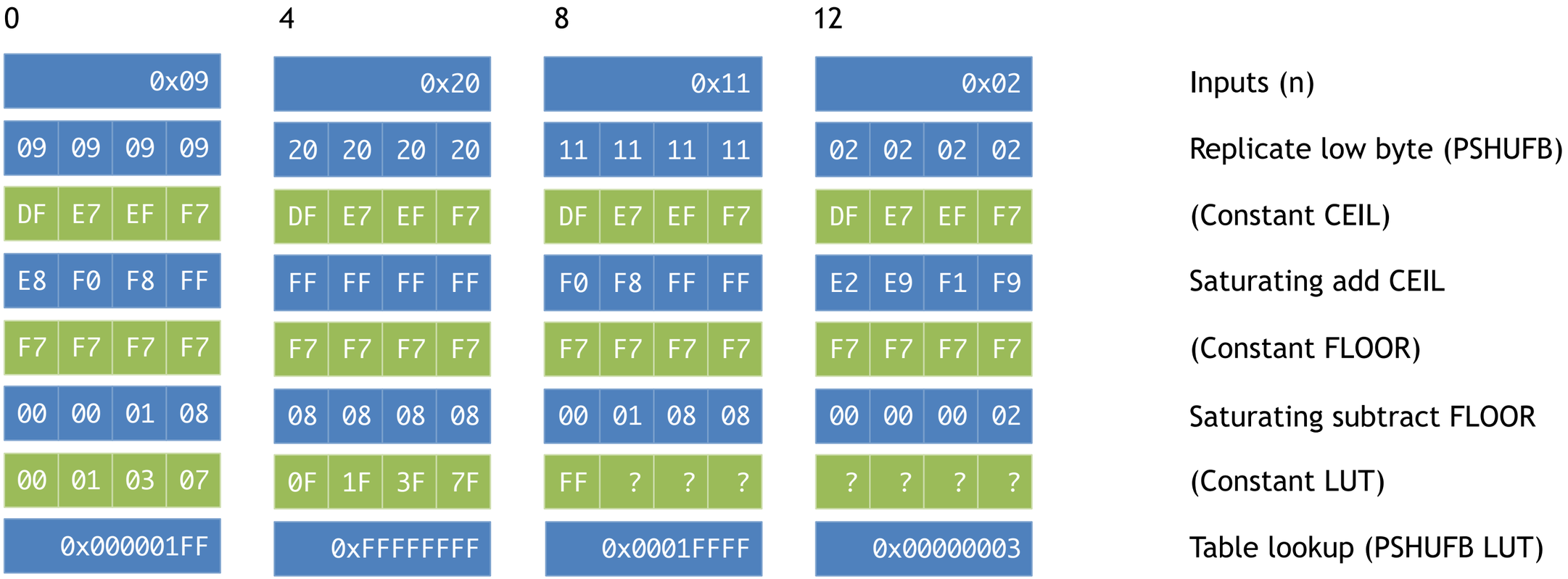
Dynamic Low Masking in Action



Dynamic Low Masking in Action

0	4	8	12	
0x09	0x20	0x11	0x02	Inputs (n)
09 09 09 09	20 20 20 20	11 11 11 11	02 02 02 02	Replicate low byte (PSHUFB)
DF E7 EF F7	DF E7 EF F7	DF E7 EF F7	DF E7 EF F7	(Constant CEIL)
E8 F0 F8 FF	FF FF FF FF	F0 F8 FF FF	E2 E9 F1 F9	Saturating add CEIL
F7 F7 F7 F7	F7 F7 F7 F7	F7 F7 F7 F7	F7 F7 F7 F7	(Constant FLOOR)
00 00 01 08	08 08 08 08	00 01 08 08	00 00 00 02	Saturating subtract FLOOR
00 01 03 07	0F 1F 3F 7F	FF ? ? ?	? ? ? ?	(Constant LUT)

Dynamic Low Masking in Action



Dynamic Low Masking Routine (SSSE3)

```
__m128i MaskLowBits_SSSE3(__m128i n)
{
  __m128i ii = _mm_shuffle_epi8(n, BYTES);
  __m128i si = _mm_adds_epu8(ii, CEIL);
          si = _mm_subs_epu8(si, FLOOR);
  return       _mm_shuffle_epi8(LUT, si);
}
```

SSE2 Techniques

- SSE2 is ancient, but fine for basic SOA SIMD
  - Massive speedups still possible
  - Sometimes basic SSE2 will beat SSE4.1 on same HW
    - Emulation in terms of “simpler” instructions can be faster
- Trickier to do “unusual” things with SSE2
  - Only fixed shuffles
  - Integer support lackluster

SSE2 Left Packing: Move Distances

- No dynamic shuffles
  - Need divide & conquer algorithm
- How far does each lane have to travel?

Mask	Output	Move Distances			
0000	....	0	0	0	0
0001	X...	0	0	0	0
0010	Y...	0	1	0	0
0011	XY..	0	0	0	0
0100	Z...	0	0	2	0
0101	XZ..	0	0	1	0
0110	YZ..	0	1	1	0
0111	XYZ.	0	0	0	0
1000	W...	0	0	0	3
1001	XW..	0	0	0	2
1010	YW..	0	1	0	2
1011	XYW.	0	0	0	1
1100	ZW..	0	0	2	2
1101	XZW.	0	0	1	1
1110	YZW.	0	1	1	1
1111	XYZW	0	0	0	0

SSE2 Left Packing: Move Distances

- No dynamic shuffles
  - Need divide & conquer algorithm
- How far does each lane have to travel?

Mask	Output	Move Distances			
0000	....	0	0	0	0
0001	X...	0	0	0	0
0010	Y...	0	1	0	0
0011	XY..	0	0	0	0
0100	Z...	0	0	2	0
0101	XZ..	0	0	1	0
0110	YZ..	0	1	1	0
0111	XYZ.	0	0	0	0
1000	W...	0	0	0	3
1010	YW..	0	1	0	2
1100	ZW..	0	0	2	2
1101	XZW.	0	0	1	1
1110	YZW.	0	1	1	1
1111	XYZW	0	0	0	0

Left Packing with Move Distances

- Process move distances (MD) bit by bit
  - Rotate left by 1 – Select based on Bit 0 of MD
  - Rotate left by 2 – Select based on Bit 1 of MD
  - And so on..

Left Packing with Move Distances

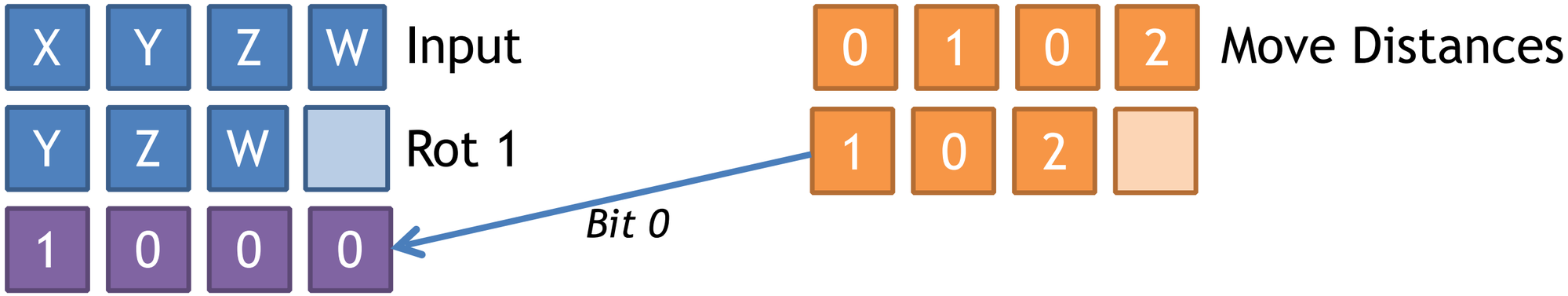
- Process move distances (MD) bit by bit
  - Rotate left by 1 – Select based on Bit 0 of MD
  - Rotate left by 2 – Select based on Bit 1 of MD
  - And so on..
- Generalizes to wider registers & more elements
  - $\log_2(n)$ rotates + selects required
  - For example 16-bit left pack, or 8x AVX float left pack
  - 2 for 4-wide case, 3 for 8-wide case, …

Left-packing `YW..` (simplified)

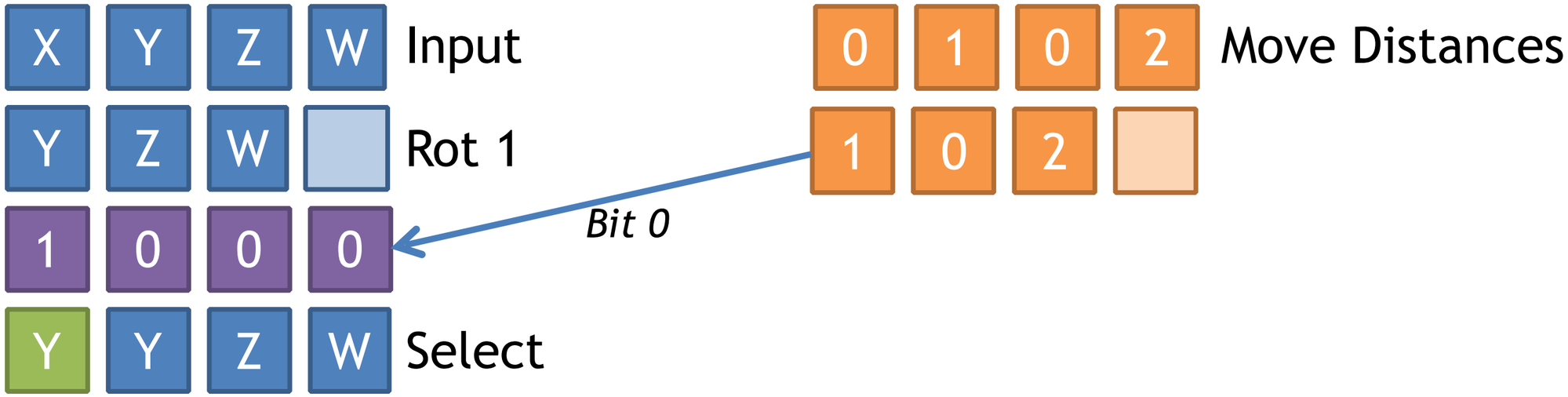
Left-packing `YW..` (simplified)

Left-packing `YW..` (simplified)

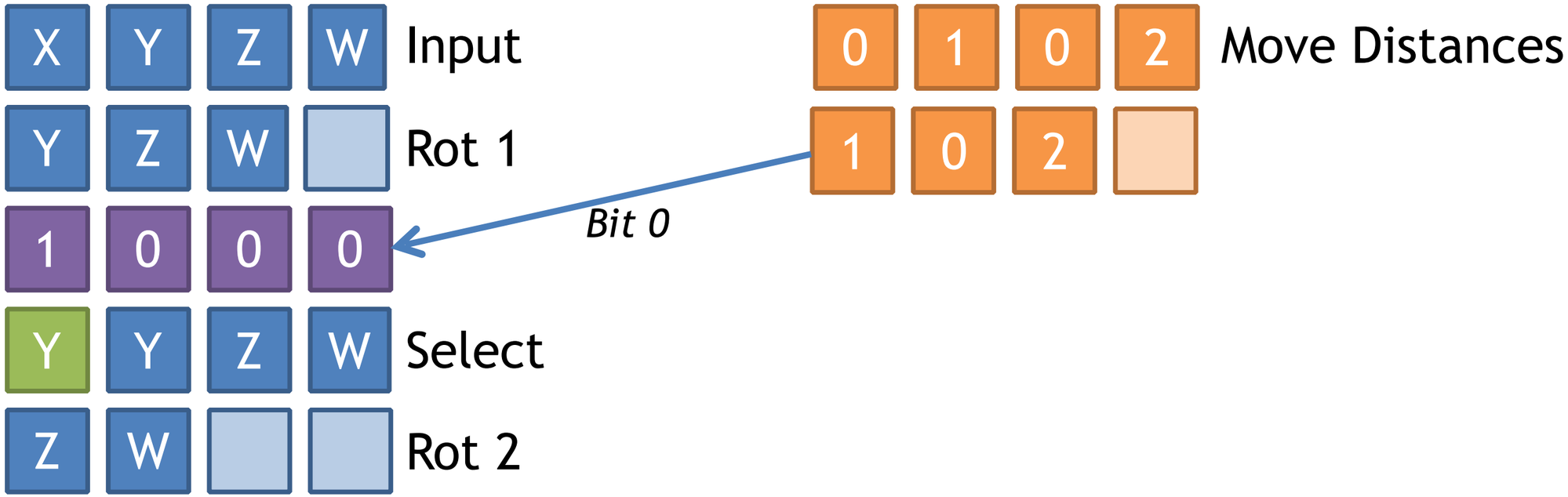
Left-packing `YW..` (simplified)



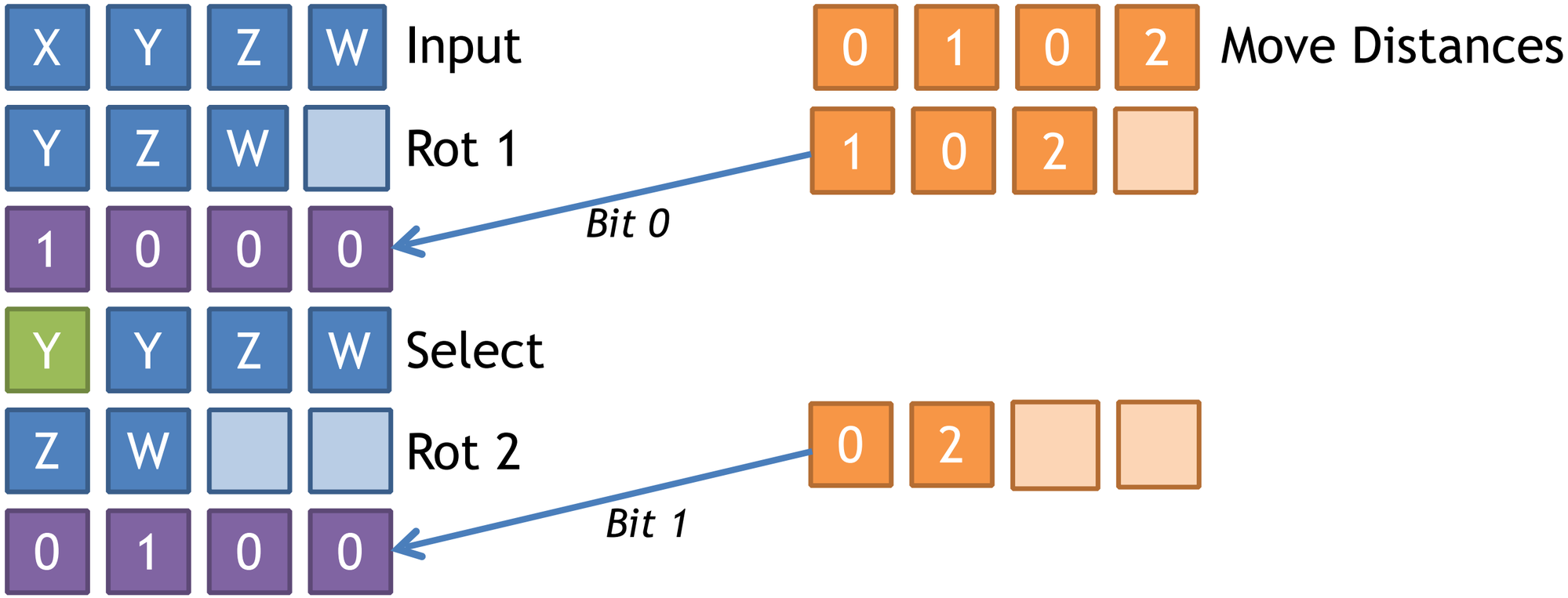
Left-packing `YW..` (simplified)



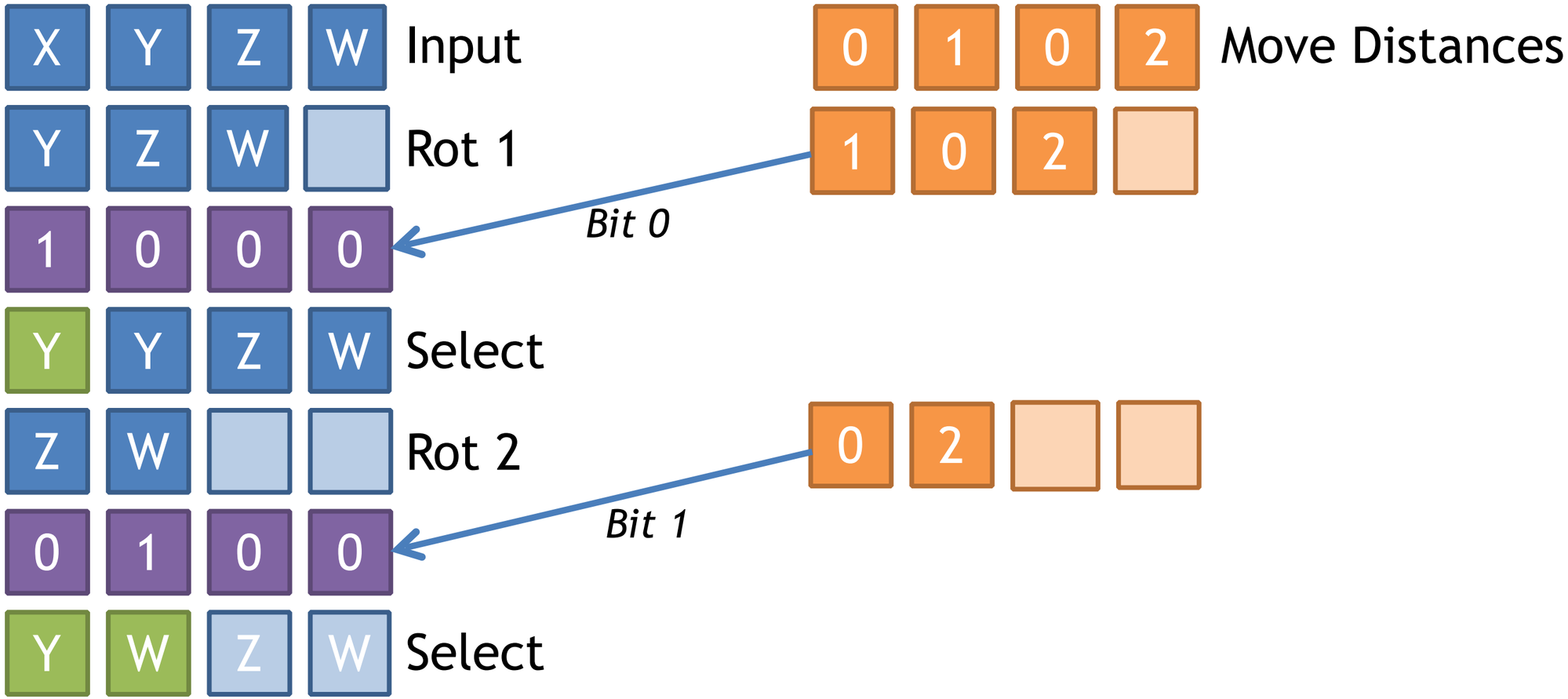
Left-packing `YW..` (simplified)



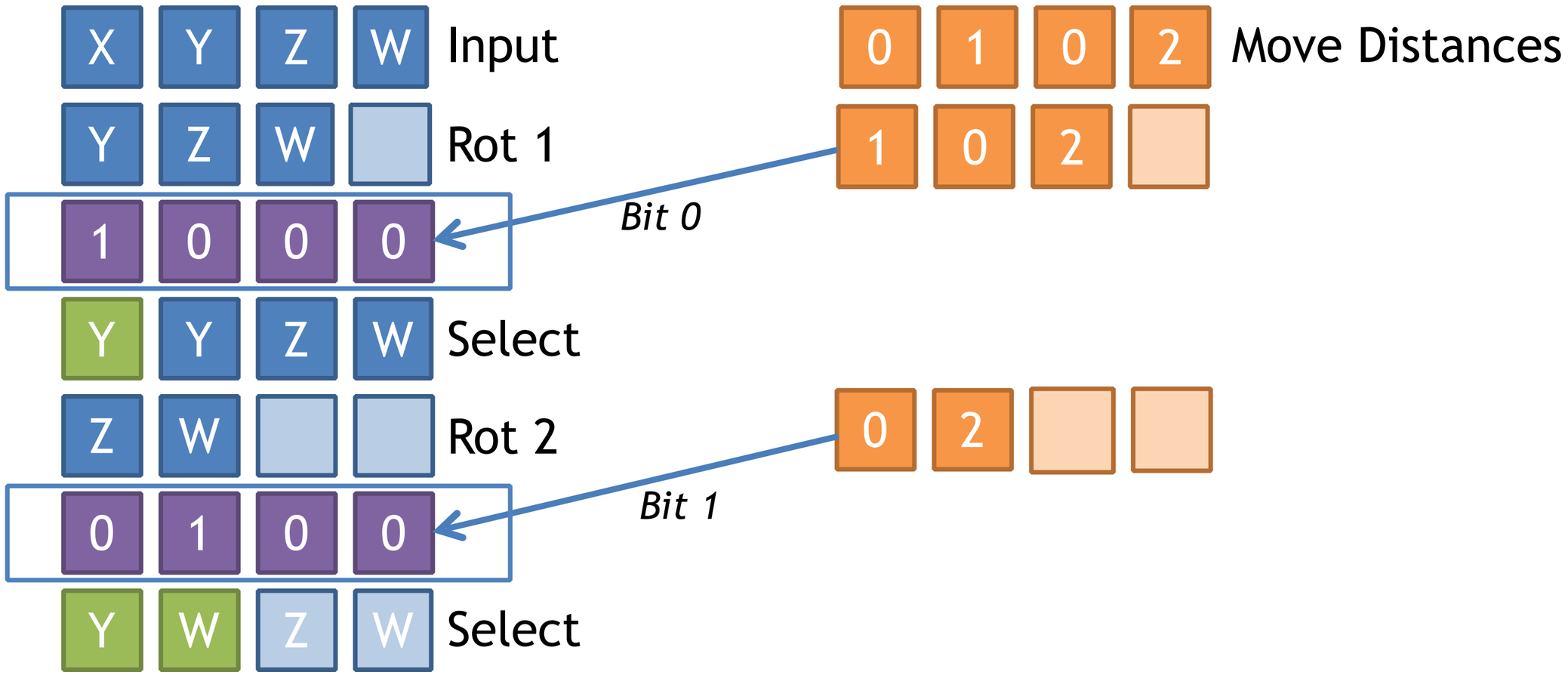
Left-packing `YW..` (simplified)



Left-packing `YW..` (simplified)



Left-packing `YW..` (simplified)



Store selection masks in LUT

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

Grab mask of valid elements

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

Load precomputed selection masks from LUT

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

First round of rotate+select

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

Second round of rotate+select

SSE2 Left Packing Code (4-wide)

```
__m128 PackLeft_SSE2(__m128 mask, __m128 val)
{
  int    valid = _mm_movemask_ps(mask);

  __m128 mask0 = _mm_load_ps((float *)(&g_Masks[valid][0]));
  __m128 mask1 = _mm_load_ps((float *)(&g_Masks[valid][4]));

  __m128 s0    = _mm_shuffle_ps(val, val, _MM_SHUFFLE(0, 3, 2, 1));
  __m128 r0    = _mm_or_ps(_mm_and_ps(mask0, s0), _mm_andnot_ps(mask0, val));

  __m128 s1    = _mm_shuffle_ps(r0, r0, _MM_SHUFFLE(1, 0, 3, 2));
  __m128 r1    = _mm_or_ps(_mm_and_ps(mask1, s1), _mm_andnot_ps(mask1, r0));

  return r1;
}
```

Dynamic Low Masking in SSE2

- Recall IEEE floating point format
  - sign * $2^{exponent}$ * mantissa

Dynamic Low Masking in SSE2

- Recall IEEE floating point format
  - sign * $2^{exponent}$ * mantissa
- That exponent sure looks like a shifter..
  - $2^n = 1 << n$

Dynamic Low Masking in SSE2

- Recall IEEE floating point format
  - sign * $2^{exponent}$ * mantissa
- That exponent sure looks like a shifter..
  - $2^n = 1 << n$
- Idea:
  - Craft special float by populating exponent with biased ***n***
  - Convert to integer, then subtract 1

Overflow woes

- Conversion from float to int is signed

Overflow woes

- Conversion from float to int is signed
- When n >= 31, can’t fit in signed integer
  - INT_MAX = `0x7fffffff`
  - Overflow is clamped to “integer indeterminate”

Overflow woes

- Conversion from float to int is signed
- When n >= 31, can't fit in signed integer
  - INT_MAX = `0x7fffffff`
  - Overflow is clamped to "integer indeterminate"
- Which happens to be.. `0x80000000`
  - Exactly what we need for n=31
  - n > 31 will clamp to 31

Dynamic Low Masking in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return          _mm_sub_epi32(intv, c_1);
}
```

Dynamic Low Masking in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return       _mm_sub_epi32(intv, c_1);
}
```

Add 127 to generate biased exponent

Dynamic Low Masking in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return       _mm_sub_epi32(intv, c_1);
}
```

Move exponent into place to make it pass as a float

Dynamic Low Masking in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return       _mm_sub_epi32(intv, c_1);
}
```

Convert the float to an int yielding 2^n as an integer

Dynamic Low Masking in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return       _mm_sub_epi32(intv, c_1);
}
```

Subtract one to generate mask

Dynamic Low Masking in SSE2

```
__m128i MaskLowBits_SSE2(__m128i n)
{
  SSE_CONSTANT_4(c_1,   uint32_t, 1);
  SSE_CONSTANT_4(c_127, uint32_t, 127);

  __m128i exp  = _mm_add_epi32(n, c_127);

  __m128i fltv = _mm_slli_epi32(exp, 23);

  __m128i intv = _mm_cvtps_epi32(_mm_castsi128_ps(fltv));

  return          _mm_sub_epi32(intv, c_1);
}
```

Best Practices: Branching

- Guideline: Avoid branches in general
  - Mispredicted branch still extremely costly on most H/W
  - Don’t want hard-to-predict branches in inner loops

Best Practices: Branching

- Guideline: Avoid branches in general
  - Mispredicted branch still extremely costly on most H/W
  - Don’t want hard-to-predict branches in inner loops
- Can be OK to branch if *very* predictable
  - Branch should be predicted correctly 99+% to make sense
  - E.g. a handful of expensive things in a sea of data
  - Use `_mm_movemask_X()+ if` on SSE2
  - Consider: `_mm_testz_si128()` and friends on SSE4.1+

Alternatives to Branching

- GPU-style “compute both branches” + select
  - Works fine for many smaller problems
  - Start here for small branches

Alternatives to Branching

- GPU-style “compute both branches” + select
  - Works fine for many smaller problems
  - Start here for small branches
- Separate input data + kernel per problem
  - Yields best performance when possible

Alternatives to Branching

- GPU-style “compute both branches” + select
  - Works fine for many smaller problems
  - Start here for small branches
- Separate input data + kernel per problem
  - Yields best performance when possible
- Consider partitioning index sets
  - Run fast kernel to partition index data into multiple sets
  - Run optimized kernel on each subset
  - Prefetching can be useful unless most indices are visited

Best Practices: Prefetching

- Absolutely necessary on previous generation
  - Not a good idea to carry this forward blindly to x86

Best Practices: Prefetching

- Absolutely necessary on previous generation
  - Not a good idea to carry this forward blindly to x86
- Guideline: Don’t prefetch linear array accesses
  - Can carry a heavy TLB miss cost chance on some H/W
  - The chip is already prefetching at the cache level for free

Best Practices: Prefetching

- Absolutely necessary on previous generation
  - Not a good idea to carry this forward blindly to x86
- Guideline: Don’t prefetch linear array accesses
  - Can carry a heavy TLB miss cost chance on some H/W
  - The chip is already prefetching at the cache level for free
- Guideline: Maybe prefetch upcoming ptrs/indices
  - **IF:** you know they will be far enough apart/irregular
  - Prefetch instructions vary somewhat between AMD/Intel
  - Test carefully that you’re getting benefit on all H/W

Best Practices: Unrolling

- Common in VMX128/SPU style code
  - Made a lot of sense with in-order machines to hide latency
  - Also had lots of registers!

Best Practices: Unrolling

- Common in VMX128/SPU style code
  - Made a lot of sense with in-order machines to hide latency
  - Also had lots of registers!
- Generally not a good idea for SSE/AVX
  - Only 16 (named) registers - H/W has many more internally
  - Out of order execution unrolls for you to some extent

Best Practices: Unrolling

- Common in VMX128/SPU style code
  - Made a lot of sense with in-order machines to hide latency
  - Also had lots of registers!
- Generally not a good idea for SSE/AVX
  - Only 16 (named) registers - H/W has many more internally
  - Out of order execution unrolls for you to some extent
- Guideline: Unroll only up to full register width
  - E.g. unroll 2x 64-bit loop to get 128 bit loop, but no more
  - Can make exceptions for very small loops as needed

Best Practices: Streaming R/W

- Do use streaming reads (>SSE 4.1) and writes
  - Helps avoid cache trashing
  - Especially for kernels using large lookup tables

Best Practices: Streaming R/W

- Do use streaming reads (>SSE 4.1) and writes
  - Helps avoid cache trashing
  - Especially for kernels using large lookup tables
- But don’t forget to fence!!
  - Different options for different architectures
    - `_mm_mfence()` always works but is slow
  - Streaming sidesteps strong x86 memory model
  - Subtle data races *will* happen if you don’t fence

Conclusion: It’s not magic

- You too can be a performance hero!
  - Small investments can yield substantial benefits
- Modern SSE is not bad at all
  - Not a lot of best practices out there
  - Hopefully this talk gives you something to start with!

SSE and AVX Resources

- ISPC - http://ispc.github.io
- Intel Instrinsics Guide
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide
  - Available as Dash DocSet for Mac OS X by yours truly
- Intel Architecture Code Analyzer
  - https://software.intel.com/en-us/articles/intel-architecture-code-analyzer
- Agner Fog’s instruction timings
  - http://www.agner.org/optimize/#manuals

Q & A

Twitter: @deplinenoise
Email: afredriksson@insomniacgames.com

Special thanks:
Fabian Giesen
Mike Day

Bonus Material

SSE in 2015: Where are we?

- SSE on x64 with modern feature set is not bad
  - Has a lot of niceties, especially in SSE 4.1 and later
  - Support heavily fragmented on PC consumer machines

SSE in 2015: Where are we?

- SSE on x64 with modern feature set is not bad
  - Has a lot of niceties, especially in SSE 4.1 and later
  - Support heavily fragmented on PC consumer machines
- Limitation #1: Only 16 registers (x64)
  - Easy to blow with a single splat'd 4x4 matrix!
  - Carefully check generated assembly from intrinsics

SSE in 2015: Where are we?

- SSE on x64 with modern feature set is not bad
  - Has a lot of niceties, especially in SSE 4.1 and later
  - Support heavily fragmented on PC consumer machines
- Limitation #1: Only 16 registers (x64)
  - Easy to blow with a single splat'd 4x4 matrix!
  - Carefully check generated assembly from intrinsics
- Limitation #2: No dynamic two-register shuffles
  - Challenge when porting Altivec/SPU style code

SSE Goodies since SSE2

Technology	*Goodies*
SSSE3	***PSHUFB, Integer Abs***
SSE4.1	***32-bit low mul, Blend, Integer Min+Max, Insert + Extract, PTEST, PACKUSDW, …***
SSE4.2 ***(POPCNT has its own CPUID flag)***	***POPCNT (only)***

SSE Fragmentation Nov 2014

Data kindly provided by Unity

Technology	Web Player	Unity Editor	Year Introduced
SSE2	100%	100%	2001
SSE3	100%	100%	2004
SSSE3	75%	93%	2006
SSE4.1	51%	83%	2007
SSE4.2	44%	77%	2008
AVX	23%	61%	2011
AVX2	4%	19%	2013

So.. What about AVX?

- Great when supported on Intel chips!
  - 2x gain for compute bound problems
  - Can easily become memory bound for simpler problems!

So.. What about AVX?

- Great when supported on Intel chips!
  - 2x gain for compute bound problems
  - Can easily become memory bound for simpler problems!
- Low availability in PC consumer space

So.. What about AVX?

- Great when supported on Intel chips!
  - 2x gain for compute bound problems
  - Can easily become memory bound for simpler problems!
- Low availability in PC consumer space
- Crippled on AMD micro architectures
  - Splits to 2 x 128 bit ALU internally (high latency)

So.. What about AVX?

- Great when supported on Intel chips!
  - 2x gain for compute bound problems
  - Can easily become memory bound for simpler problems!
- Low availability in PC consumer space
- Crippled on AMD micro architectures
  - Splits to 2 x 128 bit ALU internally (high latency)
- Not worth it for us, except for some PC tools

Cross-platform SSE in practice

- Full SSE4+ with all bells and whistles on consoles
  - Blend, population count, half<->float, …
  - VEX prefix encoding = free performance

Cross-platform SSE in practice

- Full SSE4+ with all bells and whistles on consoles
  - Blend, population count, half<->float, …
  - VEX prefix encoding = free performance
- Still need SSE2/3 compatibility for PC builds
  - Tools (and games) running on older PCs
  - Dedicated cloud servers with ancient SSE support

Cross-platform SSE in practice

- Full SSE4+ with all bells and whistles on consoles
  - Blend, population count, half<->float, …
  - VEX prefix encoding = free performance
- Still need SSE2/3 compatibility for PC builds
  - Tools (and games) running on older PCs
  - Dedicated cloud servers with ancient SSE support
- Straightforward to emulate most SSE4+ insns
  - Establish wrappers early for cross-platform projects

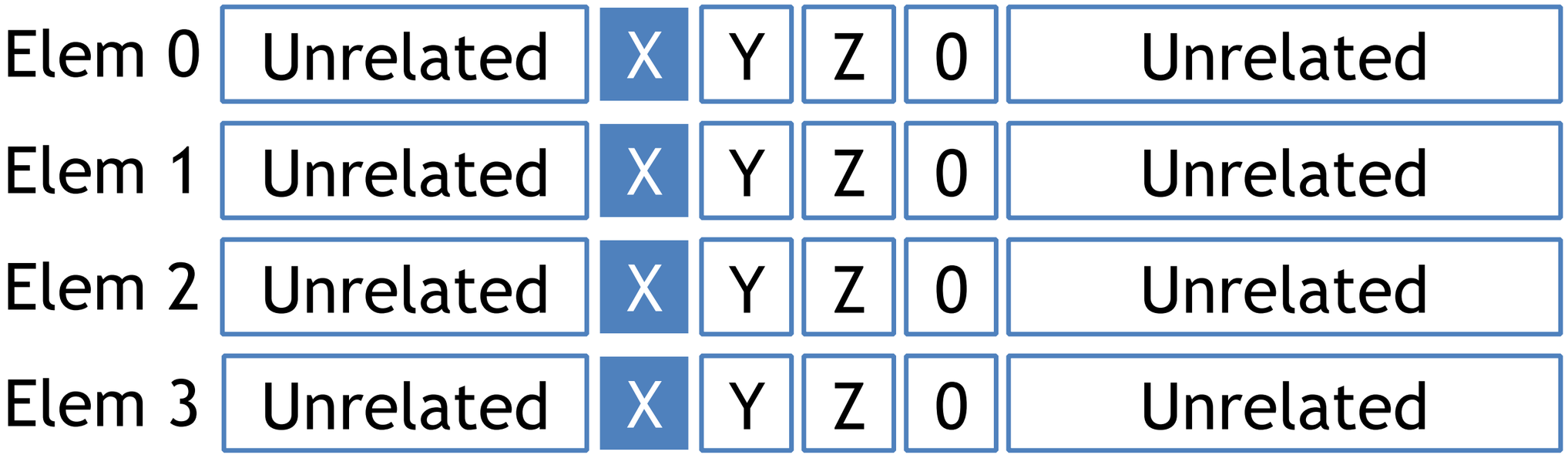
Data Layout Recap

- Two basic choices
  - AOS - Array of Structures
  - SOA - Structure of Arrays
  - Hybrid layouts possible

Data Layout Recap

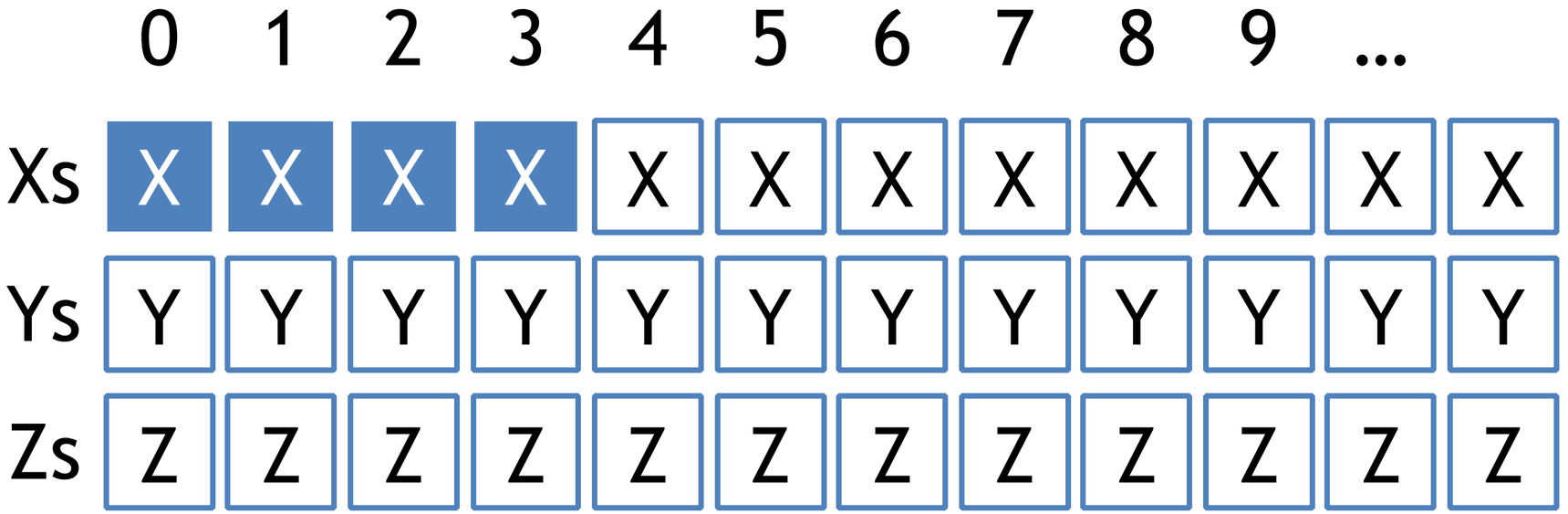
- Two basic choices
  - AOS - Array of Structures
  - SOA - Structure of Arrays
  - Hybrid layouts possible
- Most scalar code tends to be AOS
  - C++ structs and classes make that design choice implicitly
  - Clashes with desire to use SIMD instructions
  - This is probably 75% of the work to fix/compensate for

AOS Data

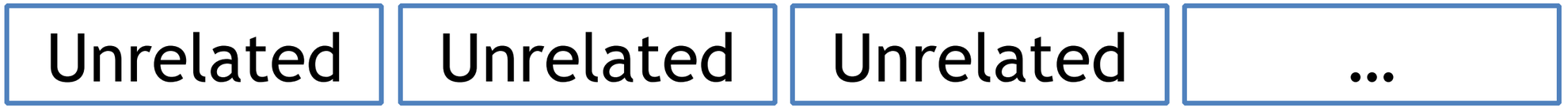

Elem 0
Unrelated
X
Y
Z
0
Unrelated
Elem 1
Unrelated
X
Y
Z
0
Unrelated
Elem 2
Unrelated
X
Y
Z
0
Unrelated
Elem 3
Unrelated
X
Y
Z
0
Unrelated

…

SOA Data



Other..



Hybrid: Tiled storage (x4)