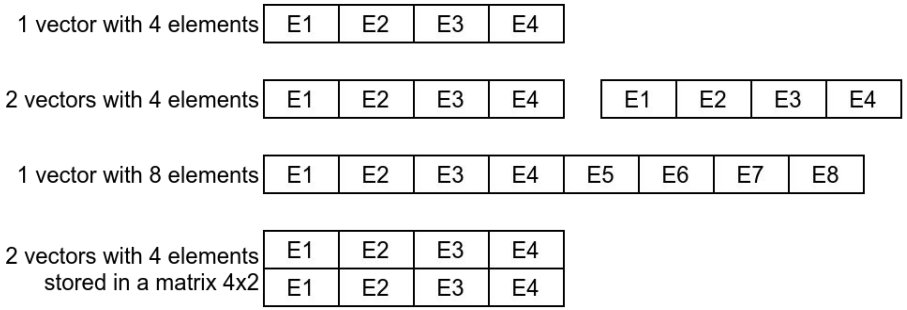
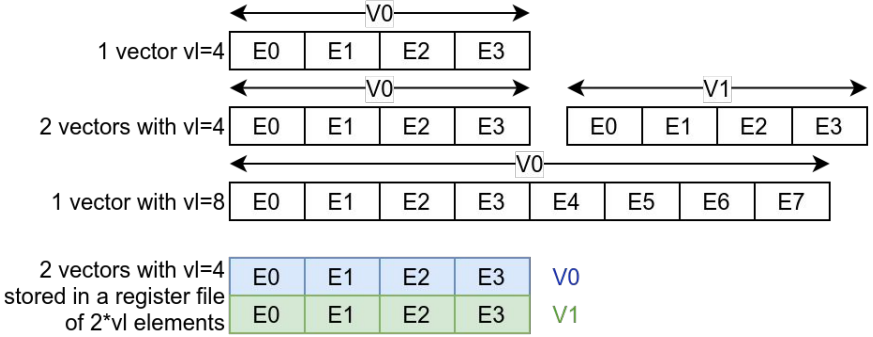
RISC-V Vector Extension

aka RVV

Vectors… as we know them in math

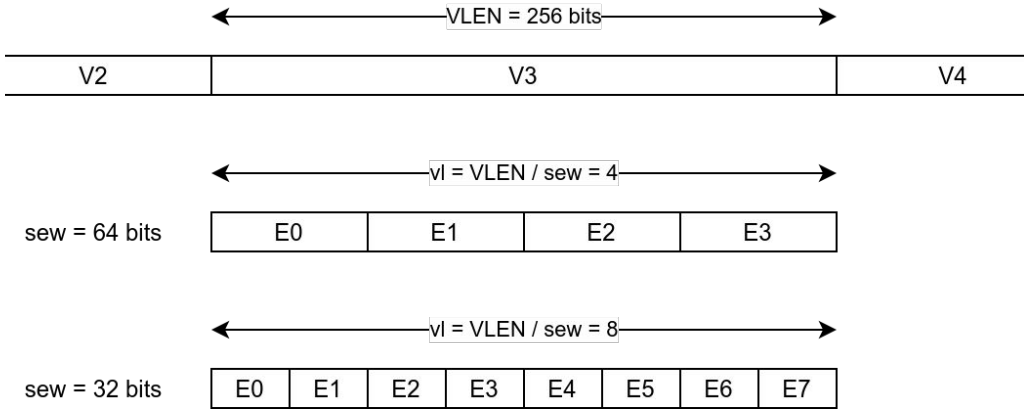




⚠️ For mapping this into hardware, we need to fix some parameters

Implementation parameters

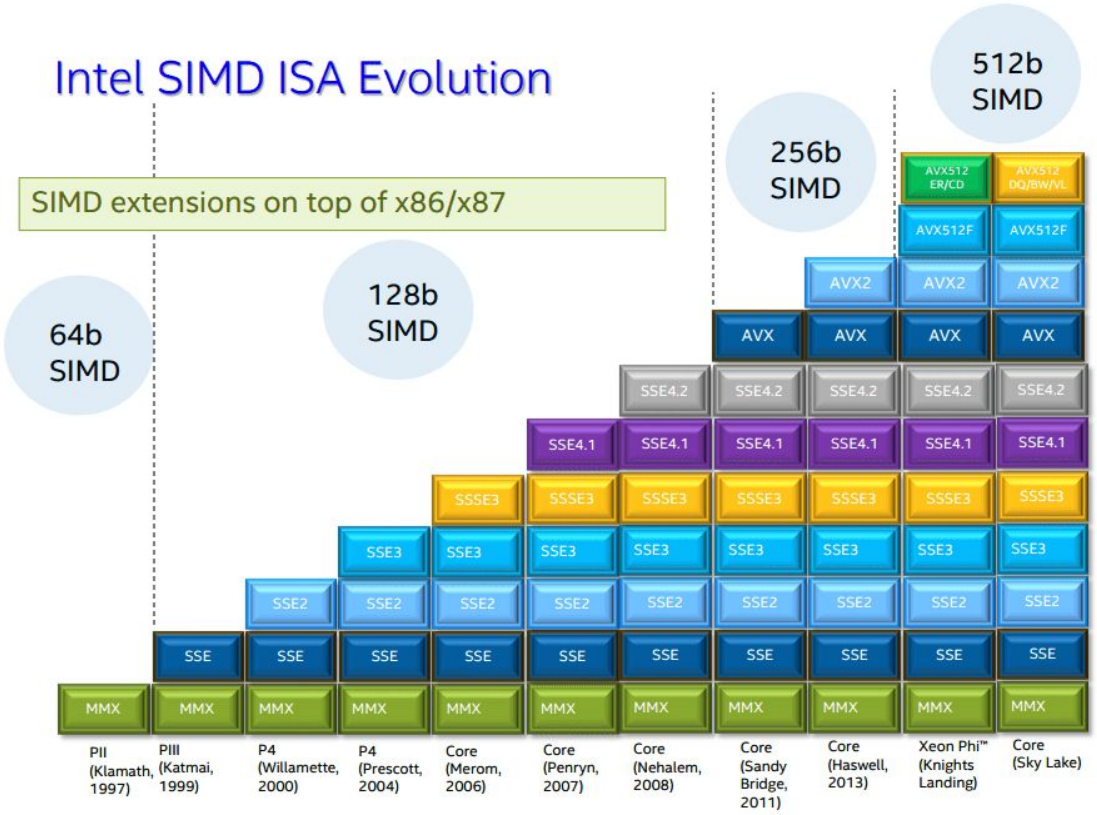
- VLEN, size of the vectors in bits
  - Micro-architectural parameter
  - RISC-V, VLEN ∈ [128; 2^{16}] bits
  - RISC-V EPAC: 2^{15} = 16384 bits
  - x86_64 SSE: 128 bits
  - x86_64 AVX: 256 bits
  - x86_64 AVX512: 512 bits
  - Arm SVE: VLEN ∈ [128; 2^{11}] bits
  - Arm Neon: 128 bits



- sew, Selected Element Width, size of the element of a vector in bits
  - You can see it as 8*sizeof(type), 64 bits for doubles/long, 32 bits for floats/int, 16 bits for (half)/short
- vl = VLEN/sew, the number of elements operated by the vector instruction

An example: x86

- VLEN = 128
  - SSE
- VLEN = 256
  - AVX
- VLEN = 512
  - AVX512



An example: AXPY with x86 intrinsics

```c
int main()
{
  int n = 32;
  float alpha = 2.0;

  // Allocate memory for vectors x and y
  float* x = (float*)malloc(n * sizeof(float));
  float* y = (float*)malloc(n * sizeof(float));

  // Initialize vectors x and y with sample values
  for (int i = 0; i < n; i++) {
    x[i] = i + 1;
    y[i] = i + 6;
  }

  printf("Original y:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  axpy(n, alpha, x, y);
  //axpy_avx512(n, alpha, x, y);
  //axpy_avx512_tail(n, alpha, x, y);

  printf("Resulting y after AXPY operation:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  // Free the allocated memory
  free(x);
  free(y);

  return 0;
}
```

```c
void axpy(int n, float alpha, float *x, float *y)
{
  for (int i = 0; i < n; i++) {
    y[i] = alpha * x[i] + y[i];
  }
}
```

An example: AXPY with x86 intrinsics

```c
int main()
{
  int n = 32;
  float alpha = 2.0;

  // Allocate memory for vectors x and y
  float* x = (float*)malloc(n * sizeof(float));
  float* y = (float*)malloc(n * sizeof(float));

  // Initialize vectors x and y with sample values
  for (int i = 0; i < n; i++) {
    x[i] = i + 1;
    y[i] = i + 6;
  }

  printf("Original y:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  axpy(n, alpha, x, y);
  //axpy_avx512(n, alpha, x, y);
  //axpy_avx512_tail(n, alpha, x, y);

  printf("Resulting y after AXPY operation:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  // Free the allocated memory
  free(x);
  free(y);

  return 0;
}
```

```c
void axpy_avx512(int n, float alpha, float *x, float *y)
{
  int i;
  __m512 alpha_vec = _mm512_set1_ps(alpha);
  for (i = 0; i < n; i += 16) {
    __m512 x_vec = _mm512_loadu_ps(&x[i]);
    __m512 y_vec = _mm512_loadu_ps(&y[i]);
    __m512 result = _mm512_fmadd_ps(alpha_vec, x_vec, y_vec);
    _mm512_storeu_ps(&y[i], result);
  }
}
```

An example: AXPY with x86 intrinsics

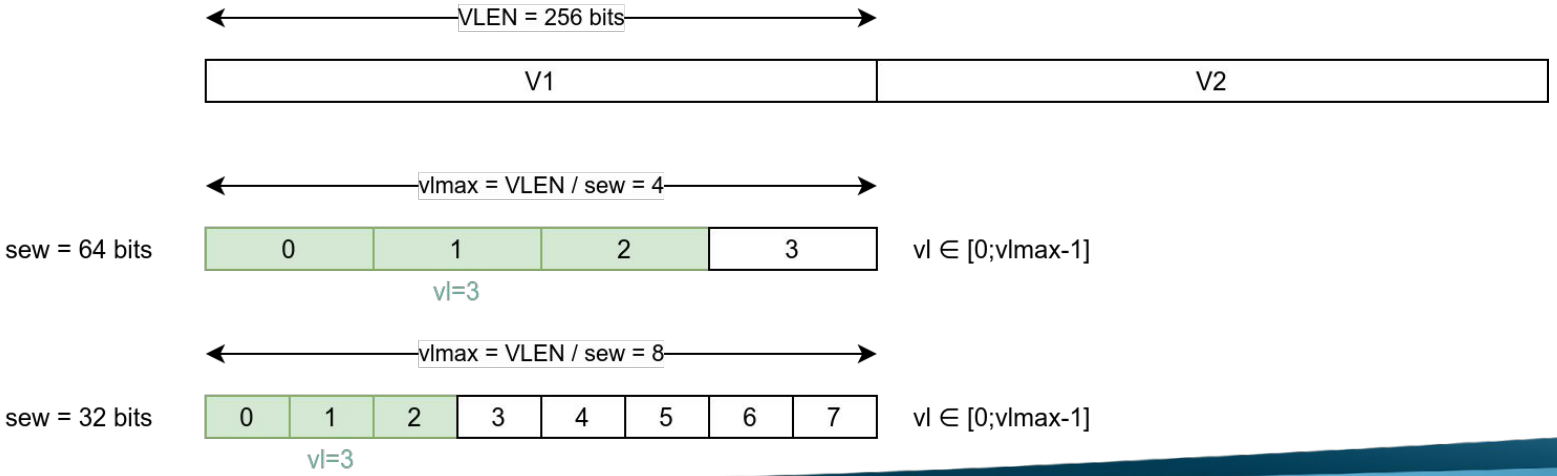
```c
int main()
{
  int n = 32;
  float alpha = 2.0;

  // Allocate memory for vectors x and y
  float* x = (float*)malloc(n * sizeof(float));
  float* y = (float*)malloc(n * sizeof(float));

  // Initialize vectors x and y with sample values
  for (int i = 0; i < n; i++) {
    x[i] = i + 1;
    y[i] = i + 6;
  }

  printf("Original y:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  axpy(n, alpha, x, y);
  //axpy_avx512(n, alpha, x, y);
  //axpy_avx512_tail(n, alpha, x, y);

  printf("Resulting y after AXPY operation:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  // Free the allocated memory
  free(x);
  free(y);

  return 0;
}
```

For a generic size of X and Y, we must handle “loop tails”

```c
void axpy_avx512_tail(int n, float alpha, float *x, float *y)
{
  int i;
  __m512 alpha_vec = _mm512_set1_ps(alpha);
  int avx512_loop_size = n - (n % 16);

  for (i = 0; i < avx512_loop_size; i += 16) {
    __m512 x_vec = _mm512_loadu_ps(&x[i]);
    __m512 y_vec = _mm512_loadu_ps(&y[i]);
    __m512 result = _mm512_fmadd_ps(alpha_vec, x_vec, y_vec);
    _mm512_storeu_ps(&y[i], result);
  }

  for (; i < n; i++) {
    y[i] = alpha * x[i] + y[i];
  }
}
```

A bit more flexible: variable vl

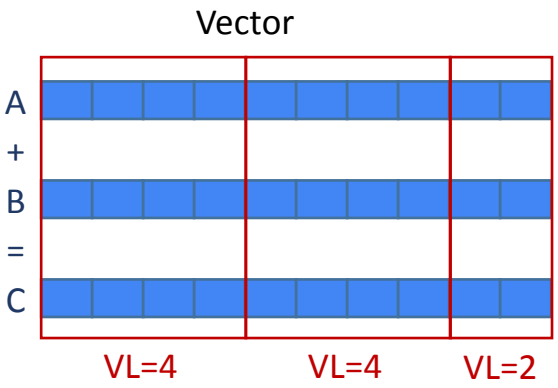
- VLEN, size of the vectors in bits
- sew, Selected Element Width, size of the element of a vector in bits
- vlmax = VLEN/sew, the number of elements operated by the vector instruction
- vl ∈ [0; vlmax-1]
  - vl = 0 → yes, it's a nop!
  - vl = vlmax → back to previous case



A bit more elegant: Variable Vector Length

- VL (Vector Length register) is a register (implemented in the hardware) that stores the size of the vector register available at any given point in time
  - VL can be loaded prior to executing the vector instruction with a special instruction (vsetvl)
  - No need to handle “loop tails”
  - Makes the code “vector length agnostic”

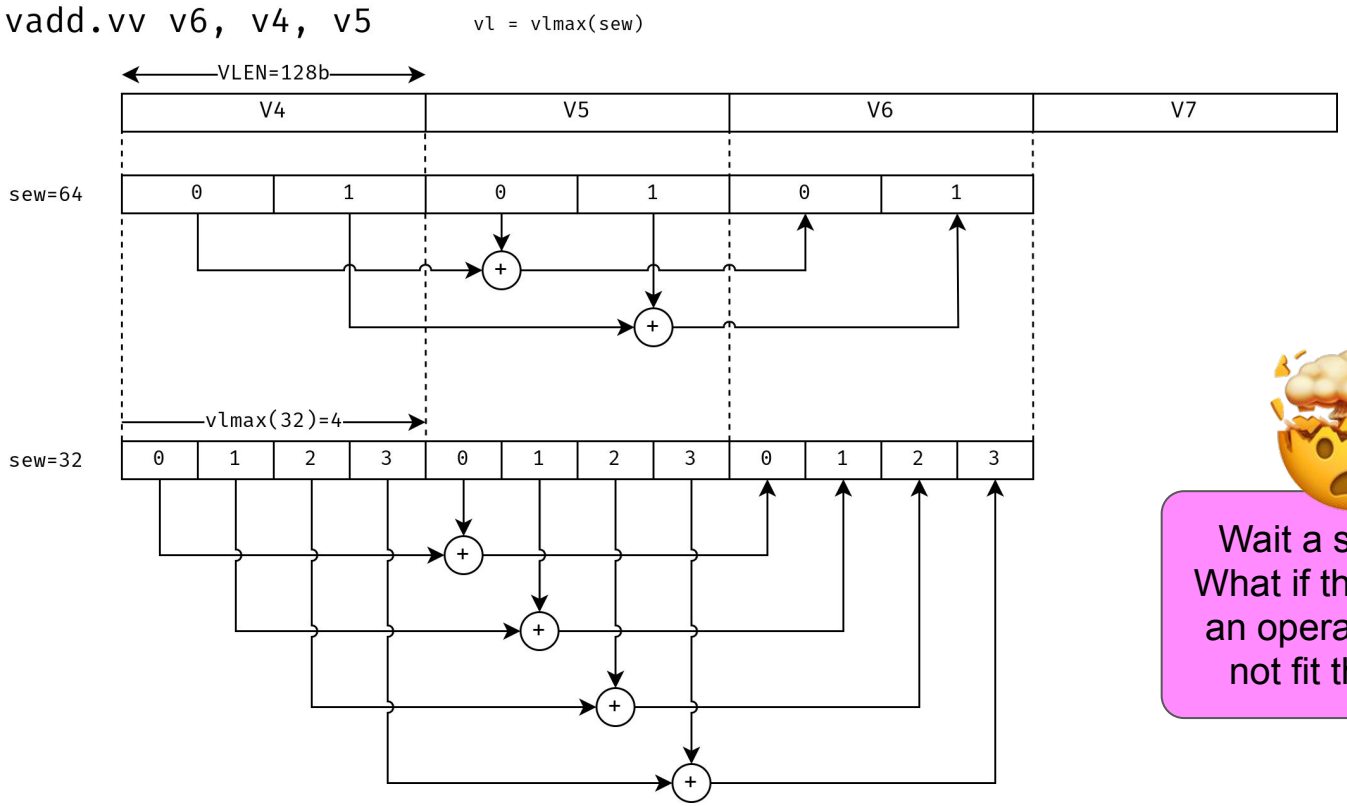
```
void axpy(double a, double *dx, double *dy, int n) {
  int i;

  long gvl = __builtin_epi_vsetvl(n, __epi_e64, __epi_m1);
  __epi_1xf64 v_a = _MM_SET_f64(a, gvl);

  for (i = 0; i < n; i += gvl) {
    gvl = __builtin_epi_vsetvl(n - i, __epi_e64, __epi_m1);
    __epi_1xf64 v_dx = _MM_LOAD_f64(&dx[i], gvl);
    __epi_1xf64 v_dy = _MM_LOAD_f64(&dy[i], gvl);
    __epi_1xf64 v_res = _MM_MACC_f64(v_dy, v_a, v_dx, gvl);
    _MM_STORE_f64(&dy[i], v_res, gvl);
  }
```



Take-home message

- RISC-V offers an extension for handling vectors (RVV)
  - It support vectors up to 2^{16} bit register (very large vectors!)
  - VLEN must be defined by the implementor
  - It supports different types of data
  - It supports variable vector length
  - All these features makes RVV Vector Length Agnostic

An example vector operation with vl=vlmax



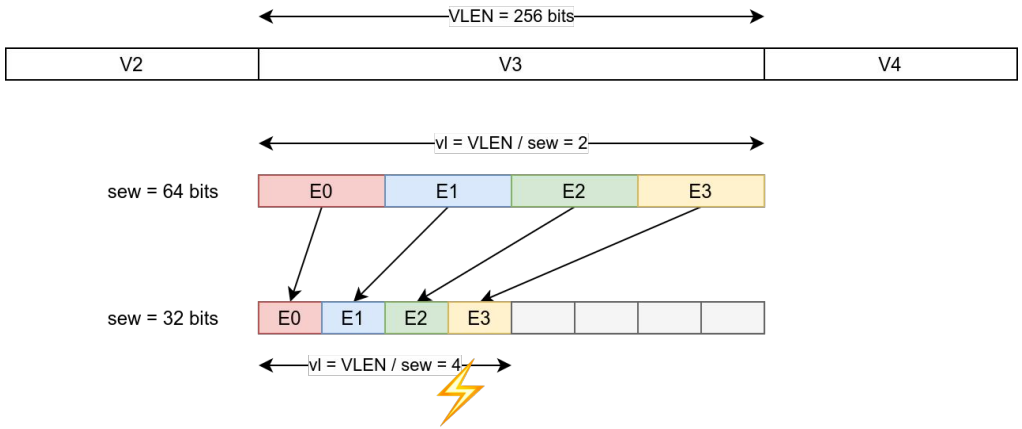
Wait a second… What if the result of an operation does not fit the sew?

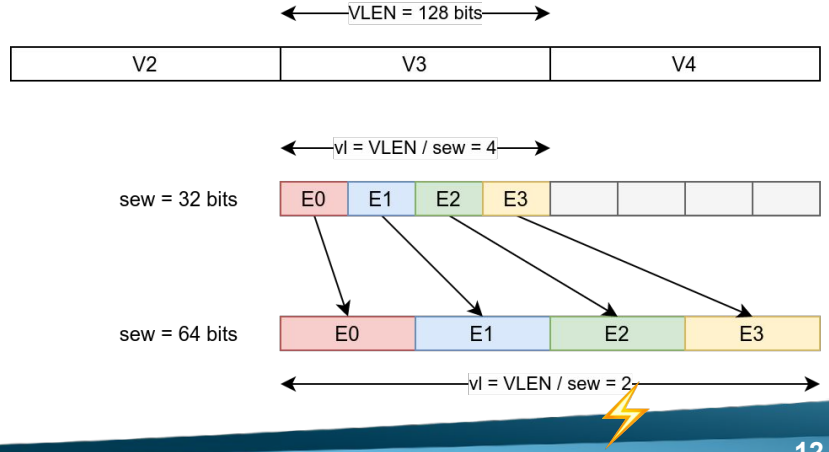
Acknowledgement: Roger Ferrer, Compiler team leader at BSC

Widening and narrowing

RISC-V RVV provides specific instructions for widening and narrowing operations. Here are some examples:

1. **Widening Instructions**:
   - `vwadd.vv`: Vector widening add (e.g., 16-bit → 32-bit).
   - `vwmul.vv`: Vector widening multiply (e.g., 16-bit → 32-bit).
   - `vfwadd.vv`: Vector widening floating-point add (e.g., 32-bit → 64-bit).
2. **Narrowing Instructions**:
   - `vnclip.wv`: Vector narrowing integer clip (e.g., 32-bit → 16-bit).
   - `vfncvt.f.x.w`: Vector narrowing floating-point convert (e.g., 64-bit → 32-bit).





Widening and narrowing: Practical Use Cases

1. Machine Learning:
   - Widening: Accumulating 16-bit gradients into 32-bit during training.
   - Narrowing: Storing 32-bit weights as 16-bit for inference.

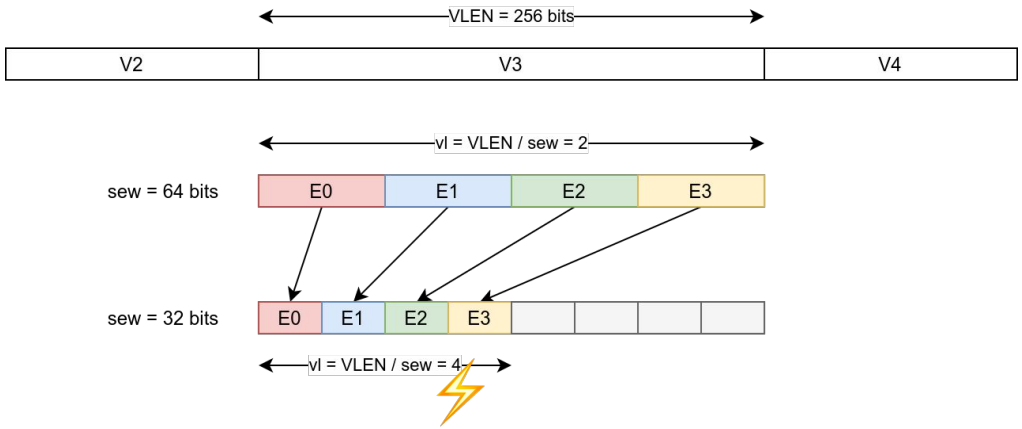
2. Signal Processing:
   - Widening: Performing high-precision filtering on 16-bit audio samples.
   - Narrowing: Storing processed audio in 16-bit format.

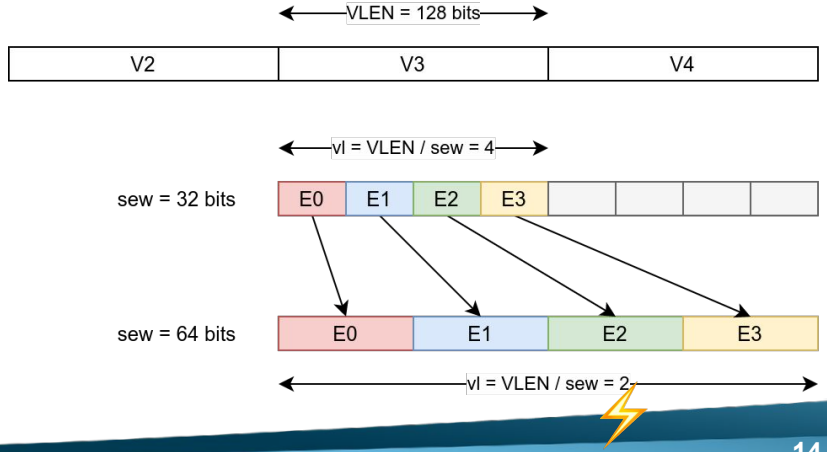
3. Scientific Computing:
   - Widening: Avoiding precision loss in intermediate calculations.
   - Narrowing: Reducing memory usage for large datasets.

Widening and narrowing

RISC-V RVV provides specific instructions for widening and narrowing operations. Here are some examples:

1. **Widening Instructions**:
   - `vwadd.vv`: Vector widening add (e.g., 16-bit → 32-bit).
   - `vwmul.vv`: Vector widening multiply (e.g., 16-bit → 32-bit).
   - `vfwadd.vv`: Vector widening floating-point add (e.g., 32-bit → 64-bit).
2. **Narrowing Instructions**:
   - `vnclip.wv`: Vector narrowing integer clip (e.g., 32-bit → 16-bit).
   - `vfncvt.f.x.w`: Vector narrowing floating-point convert (e.g., 64-bit → 32-bit).





Vector Register Group Multiplier: lmul

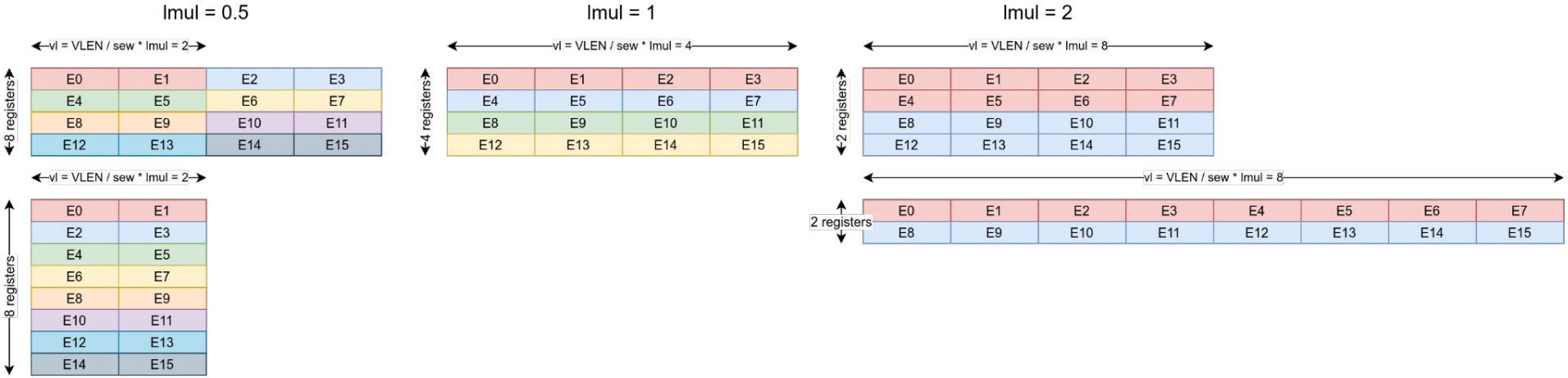
- lmul aka length multiplier
- is a mechanism to cope with different sew
  - if widening makes sew * 2 → lmul = 2
  - if narrowing makes sew / 2 → lmul = ½

Small correction:

- VLEN, size of the vectors in bits
- sew, Selected Element Width, size of the element of a vector in bits
- vlmax = (VLEN/sew)*lmul, the number of elements operated by the vector instruction
- vl ∈ [0; vlmax-1]

Vector Register Group Multiplier: lmul

- RVV allows lmul ∈ [⅛, ¼, ½, 1, 2, 4, 8]
- This creates the illusion of
  - a “smaller” register bank with larger registers → when lmul > 1
  - a “larger” register bank with smaller register → when lmul < 1



Take-home message

- RISC-V vector extension allows flexibility with size of the operands
  - This implies re-arrangements using an extra parameter “length multiplier” lmul
  - It adds flexibility and the illusion of having more/less registers
  - The full support of lmul requires both hardware and software support adding significant complexity to the development of the actual implementation

Take-home message

- RISC-V vector extension allows flexibility with size of the operands
  - This implies re-arrangements using an extra parameter “length multiplier” lmul
  - It adds flexibility and the illusion of having more/less registers
  - The full support of lmul requires both hardware and software support adding significant complexity to the development of the actual implementation

Thank the tech gods somebody took care of the compiler…
So we don’t have to deal with all of this

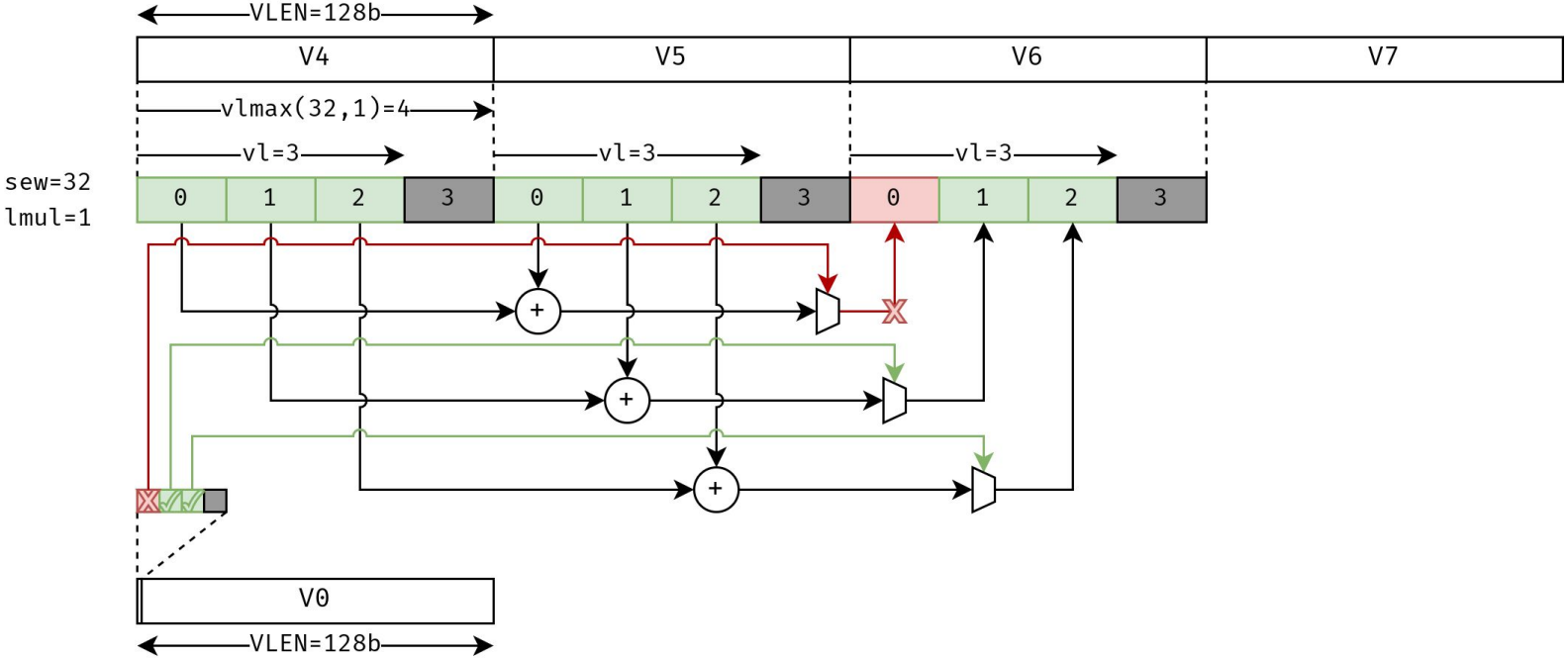
Masked execution

- A mask vector is a vector of elements of 1 bit in a (regular) vector register (as if sew=1)
  - An element of the mask set to 0 disables the corresponding element in the vector operation
  - An element of the mask set to 1 enables the corresponding element in the vector operation
- The ISA provides several instructions that generate mask vectors
  - e.g., comparisons, OR-ing two masks, etc.
  - in mnemonics “m” is used to distinguish mask vectors from regular data vectors, marked with “v”
- Instructions can be masked using mask vectors
  - The mask vector of a masked instruction is always v0
- Unmasked instructions can be (conceptually) understood as having a mask of all ones.
  - They may not be equivalent performance-wise though

Acknowledgement: Roger Ferrer, Compiler team leader at BSC

Masked execution example

```
vadd.vv v6, v4, v5, v0.t      vl=3
```



Acknowledgement: Roger Ferrer, Compiler team leader at BSC

Take-home message

- RISC-V vector extension improves flexibility allowing the use of masks
  - masks allows to filter elements in a vector operation
  - The full support of masks requires both hardware and software support adding significant complexity to the development of the actual implementation
  - Masks are often generated by conditionals inside loops
    - they introduce performance penalties, but maintain the code linear and clean

Thank the tech gods somebody took care of the compiler…
So we don't have to deal with all of this