Generating **Programmable NPUs** from Linalg with MLIR and CIRCT

Josse Van Delm

LLVM dev room
FOSDEM 2026

whoami

Josse Van Delm

PhD Researcher at KU Leuven - MICAS research group

KU LEUVEN

Working on compilers for AI hardware accelerators since 2021

Fun fact! Looking for a new job at the end of this year 😉

What you'll learn in this talk

- *What are NPUs?*
- *How do people design NPUs now?*
- *How people should design NPUs in the future (spoiler: with compilers!)*

NPUs are processors to optimize dense linear algebra

```
void elementwise_mult(float* A, float* B, float* C) {
   for (int i = 0; i < 16; i++) {
       C[i] = A[i] * B[i];
   }
}
```

NPUs are processors to optimize dense linear algebra

```
void elementwise_mult(float* A, float* B, float* C) {
    for (int i = 0; i < 16; i++) {
        C[i] = A[i] * B[i];
    }
}
```

NPUs are processors to optimize dense linear algebra

```
void elementwise_mult(float* A, float* B, float* C) {
    for (int i = 0; i < 16; i++) {
        C[i] = A[i] * B[i];
    }
}
```

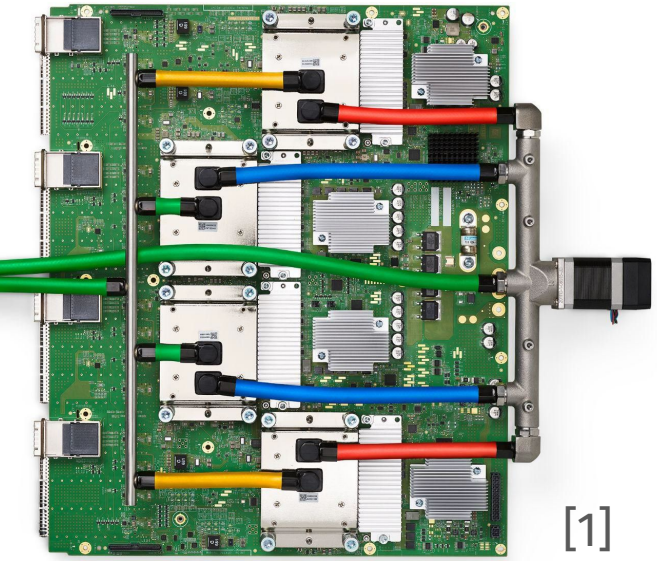
```
void matmul_16×16(float A[16][16], float B[16][16], float C[16][16]) {
    for (int i = 0; i < 16; i++) {
        for (int j = 0; j < 16; j++) {
            C[i][j] = 0;
            for (int k = 0; k < 16; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

NPUs are processors to optimize dense linear algebra

```
void softmax_16(float* input, float* output) {
    float sum = 0.0f;
    float max_val = input[0];
    for (int i = 1; i < 16; i++) {
        if (input[i] > max_val) max_val = input[i];
    }
    for (int i = 0; i < 16; i++) {
        output[i] = expf(input[i] - max_val);
        sum += output[i];
    }
    for (int i = 0; i < 16; i++) {
        output[i] /= sum;
    }
}
```

Aside: MLIR represents this in the "linalg" dialect

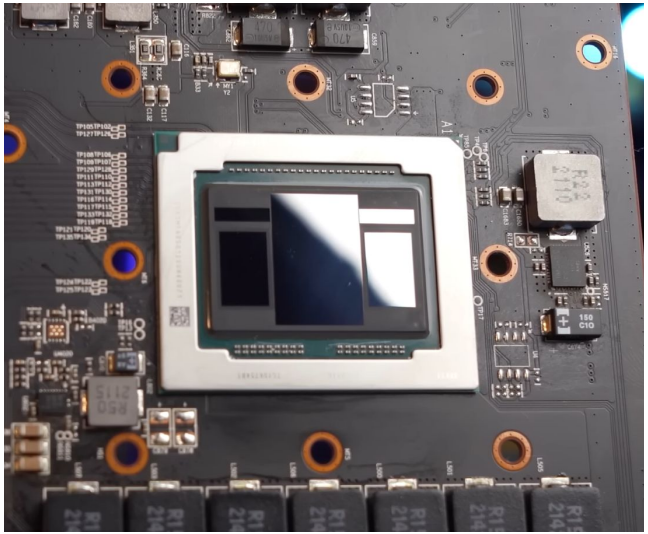
```
void elementwise_mult(float* A, float* B, float* C) {
   for (int i = 0; i < 16; i++) {
       C[i] = A[i] * B[i];
   }
}

%3 = linalg.mul ins(%arg0, %2 : tensor<16xf32>, tensor<16xf32>) outs(%0 : tensor<16xf32>) →
tensor<16xf32>

#map = affine_map<(d0) → (d0)>
%3 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
ins(%arg0, %2 : tensor<16xf32>, tensor<16xf32>) outs(%0 : tensor<16xf32>) {
 ^bb0(%in: f32, %in_0: f32, %out: f32):
   %5 = arith.mulf %in, %in_0 : f32
   linalg.yield %5 : f32
} → tensor<16xf32>
```



[1]


H9HCNNDBMMLSR NEH
VTD1KC 2031 1A
U9G269Z5LL 2036
APL1102 339S00833 S

[2]

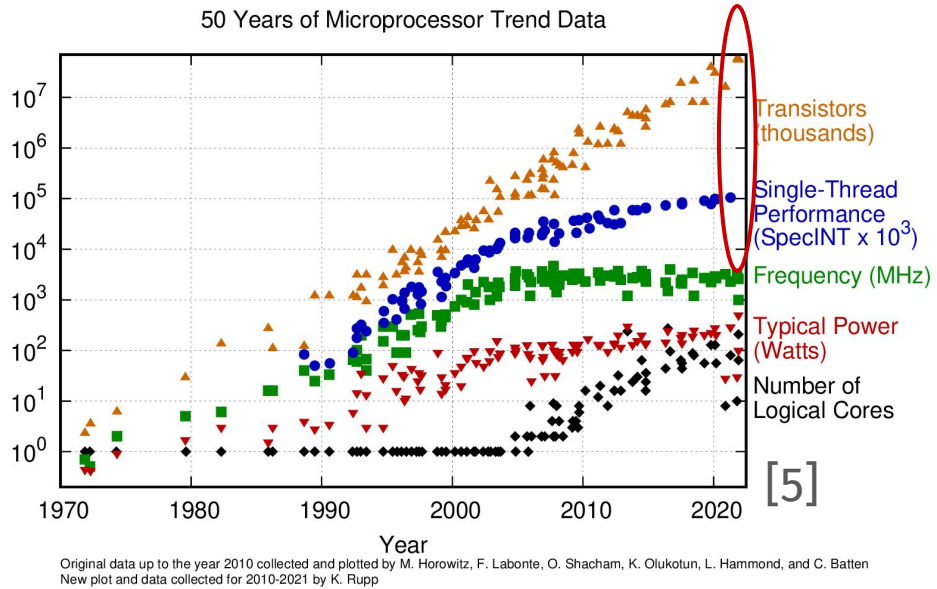


[3]

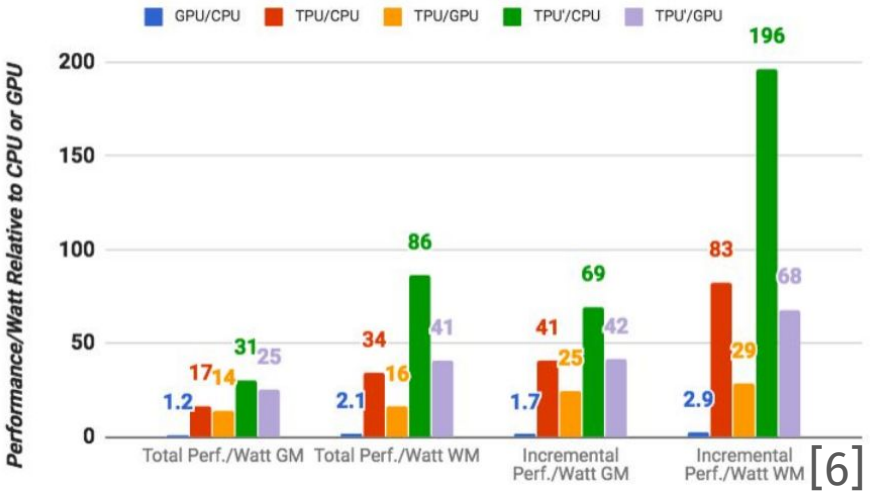

NVIDIA
NVIDIA
NVIDIA

[4]

Are they useful? According to research: yes!



[5]



[6]

There is more available silicon area on chip for specialized processors

TPU = much faster and energy efficient at linear algebra than CPUs or GPUs

But are they really useful?

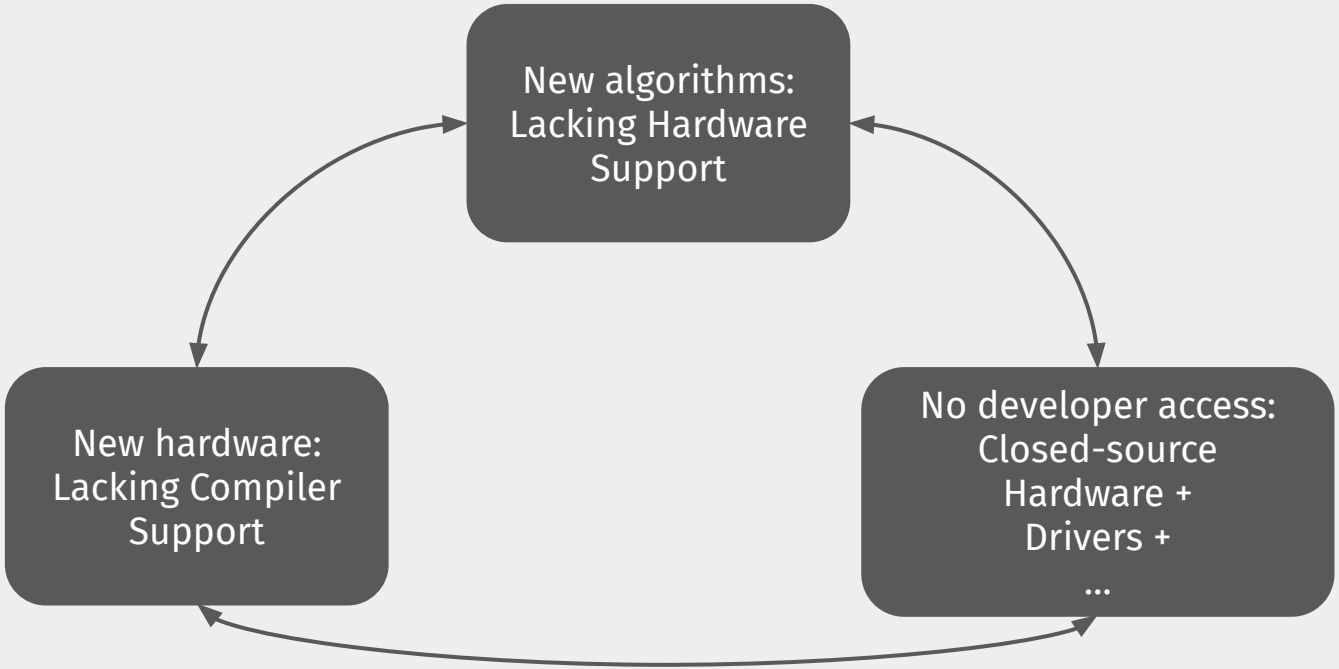
- Yes. … but mostly for marketing

  In reality it is often too difficult to use them
- These units can be too **specific** for newer workloads

- Most compilers can’t leverage them

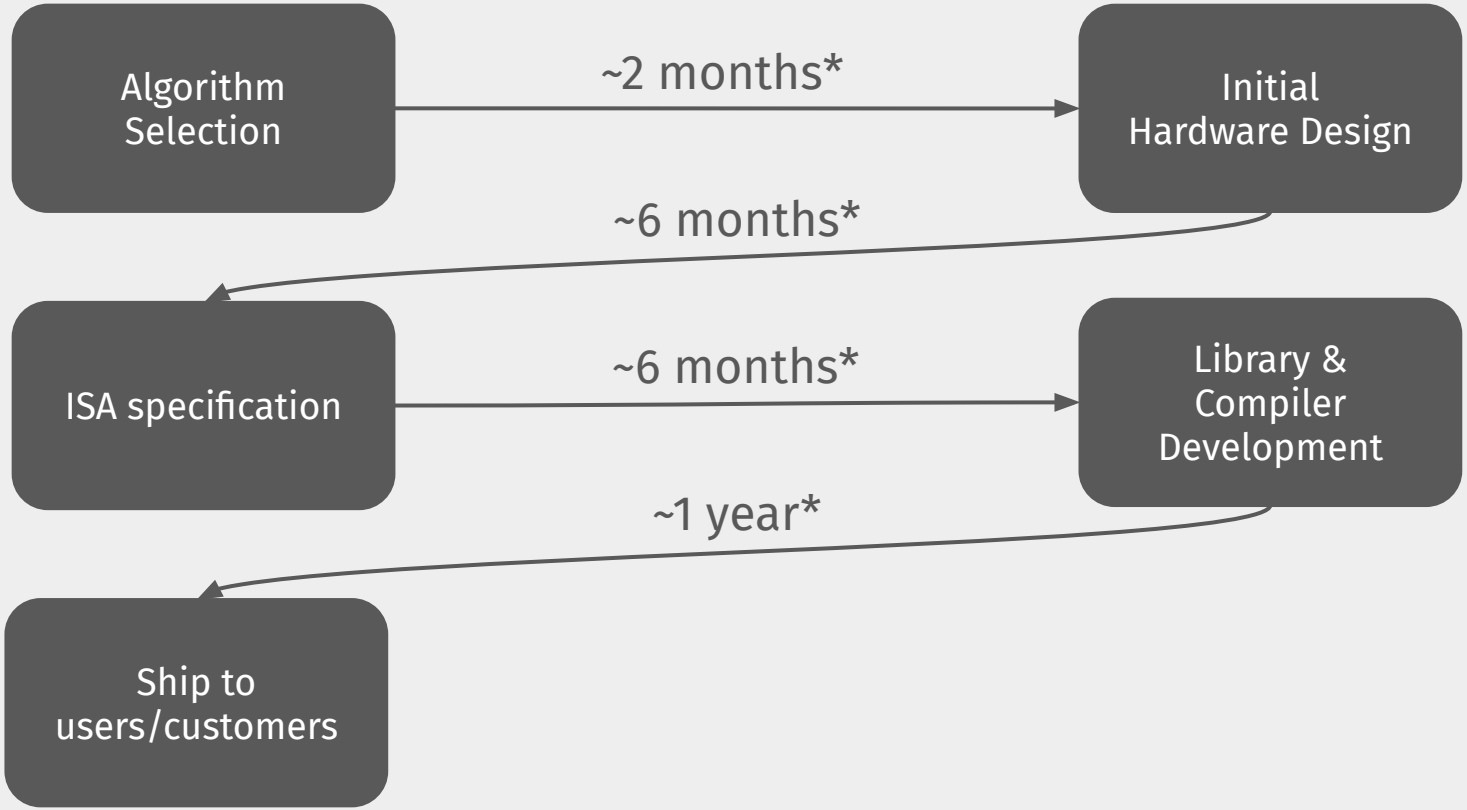
- Hardware, Firmware, Libraries often **closed source!**

There are multiple interdependent problems

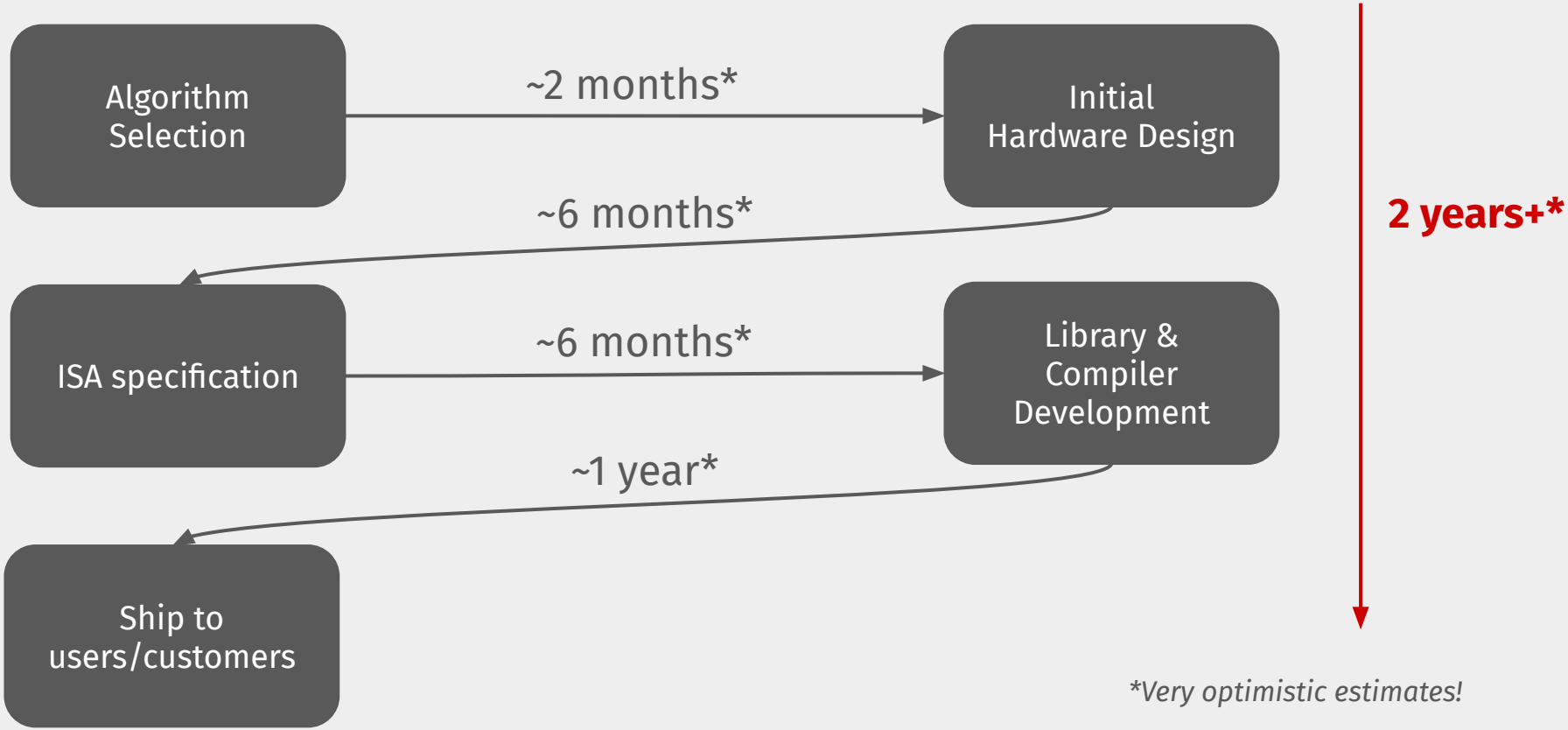


How did we get here?

How do people typically Create NPUs?



How do people typically Create NPUs?



Very optimistic estimates!

This is too slow! Can we do this faster?

Yes! With compilers we can enable **programmable hardware synthesis**

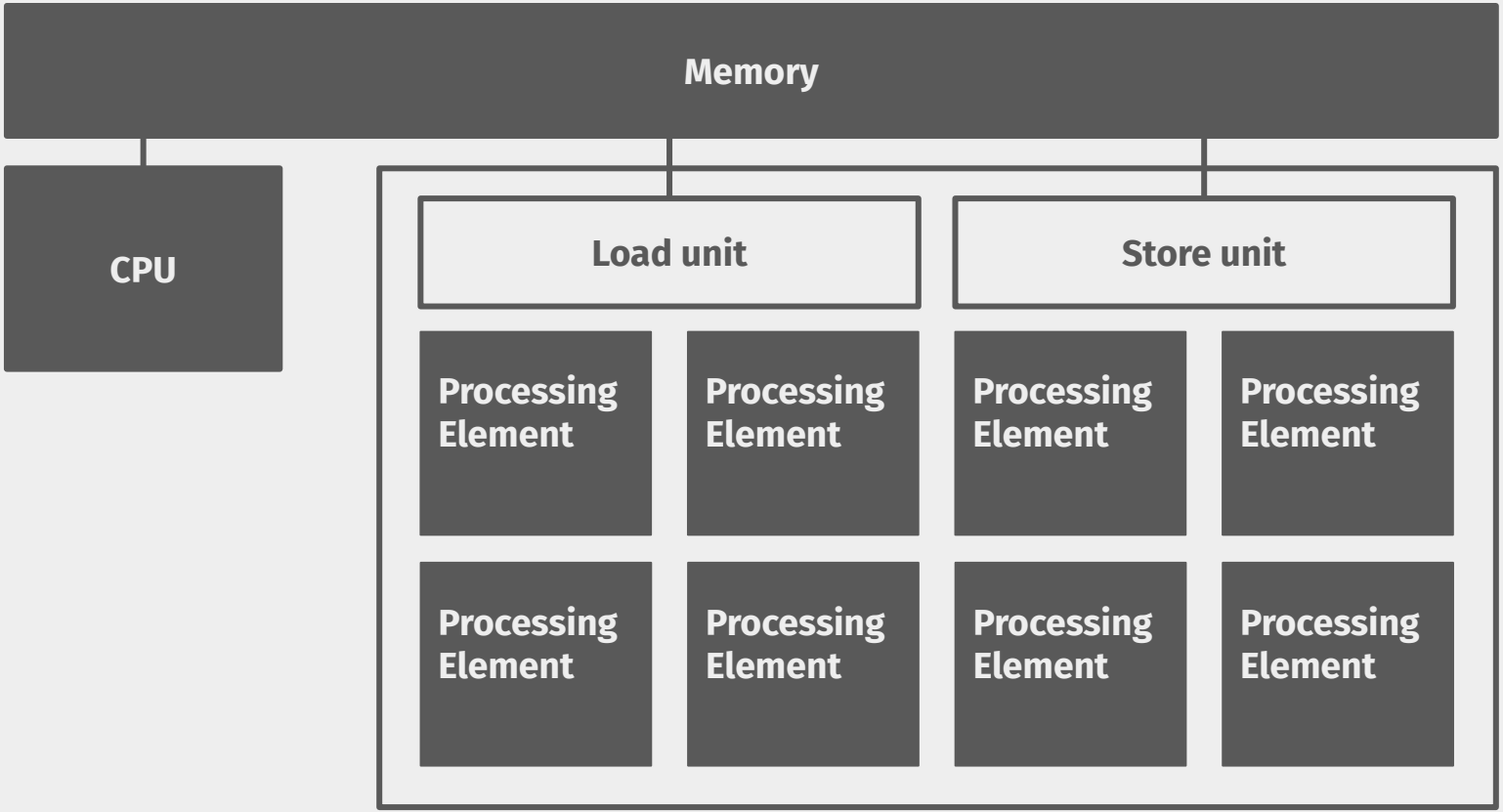
But how?

- → How do we quickly create an ML compiler? 🐉 **MLIR**
- → How do we quickly generate hardware? ⚡ **CIRCT**
- → How do we adapt the compiler to the hardware? 🏃 **Dynamic Backend**
- → How do we quickly develop this? 🪄 **Secret shortcuts**

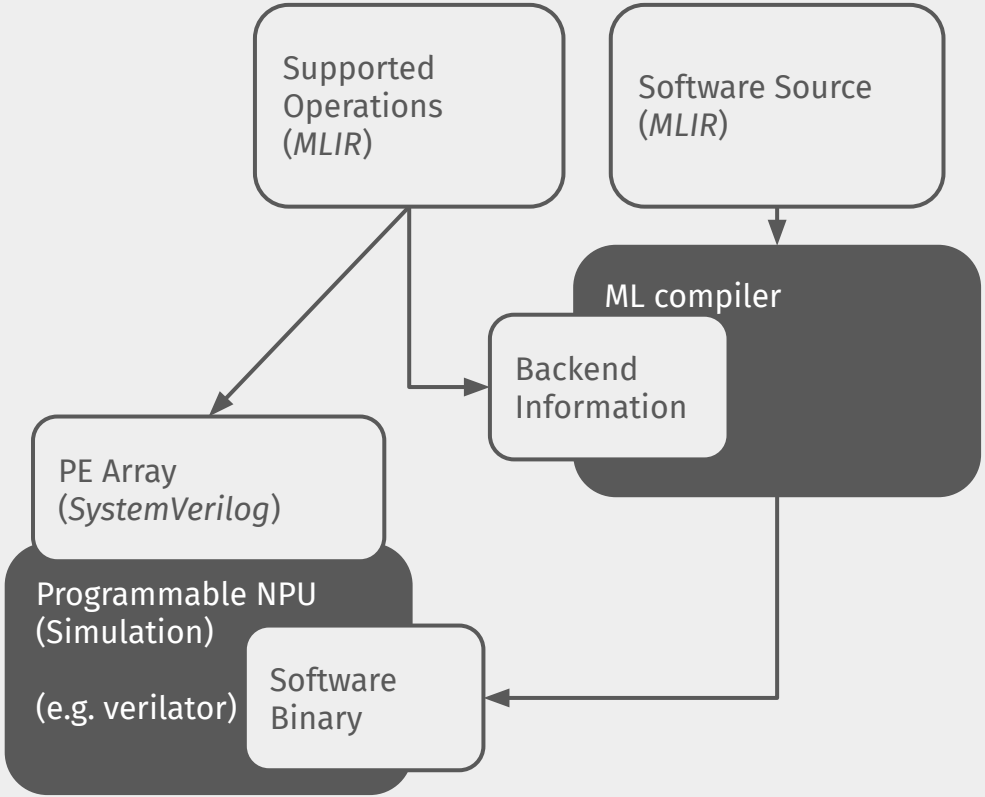
Fully Open source 🦾

What are we trying to create?

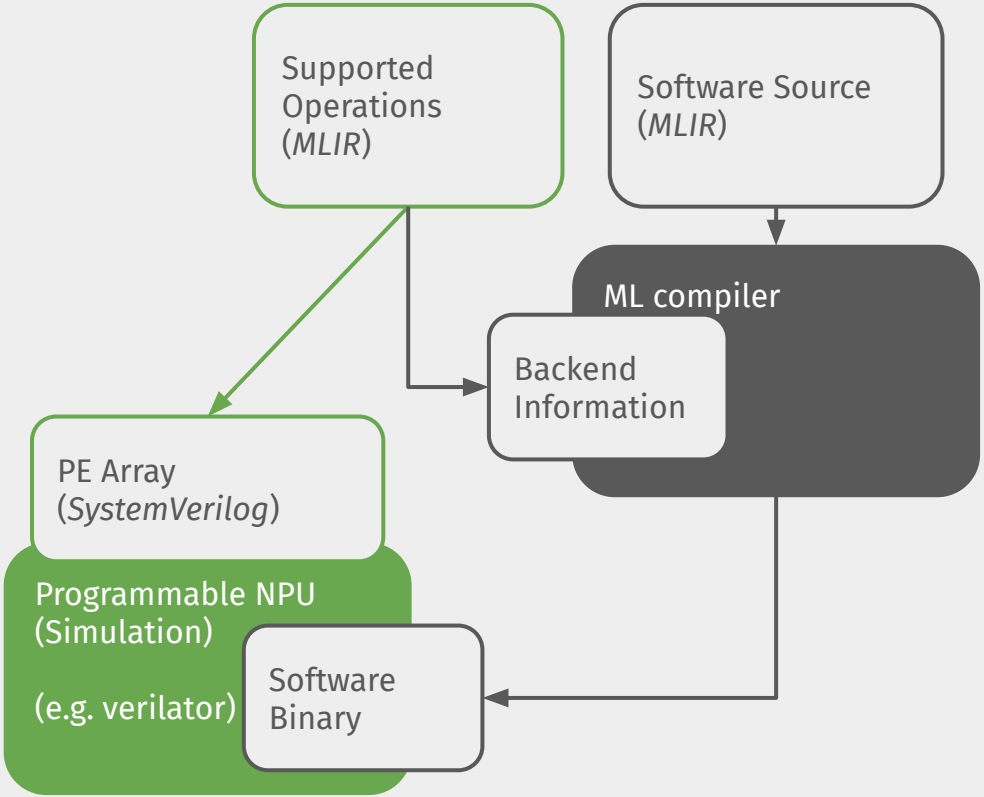
Anatomy of an NPU platform

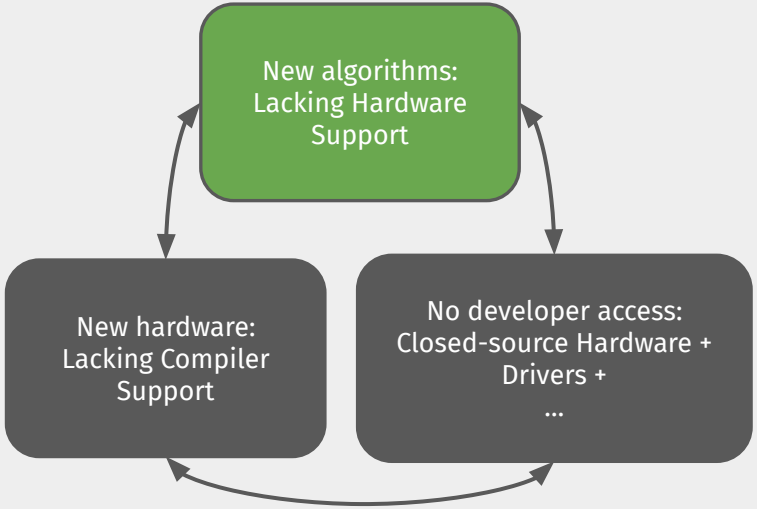


The programmable hardware synthesis flow

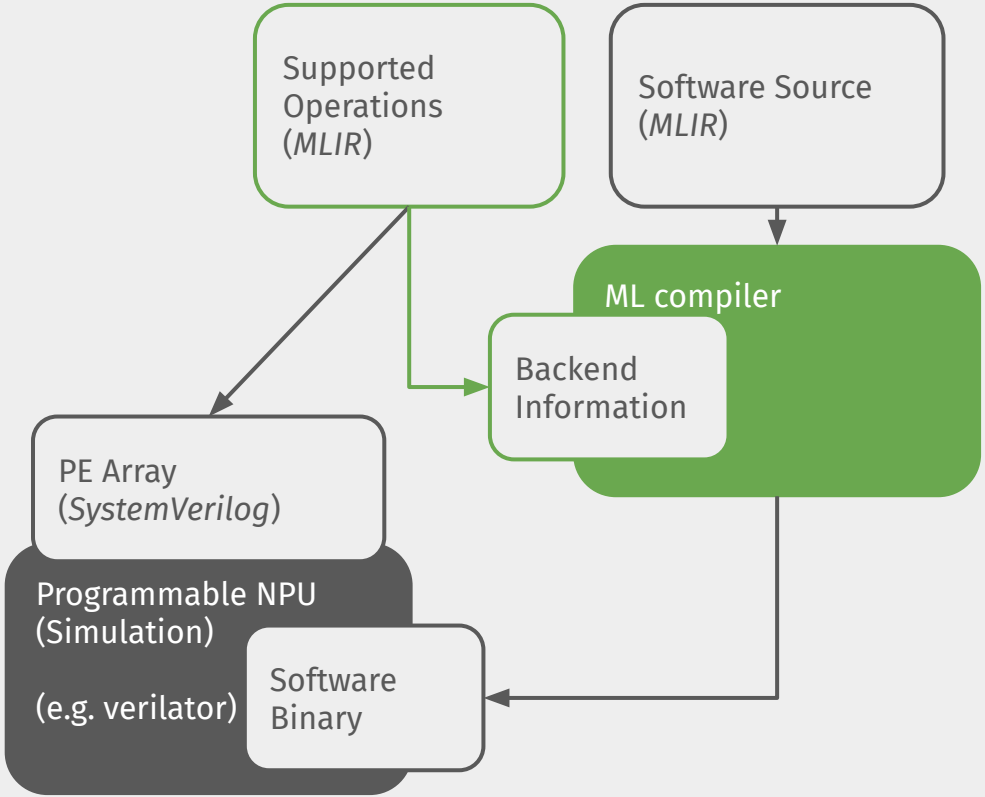


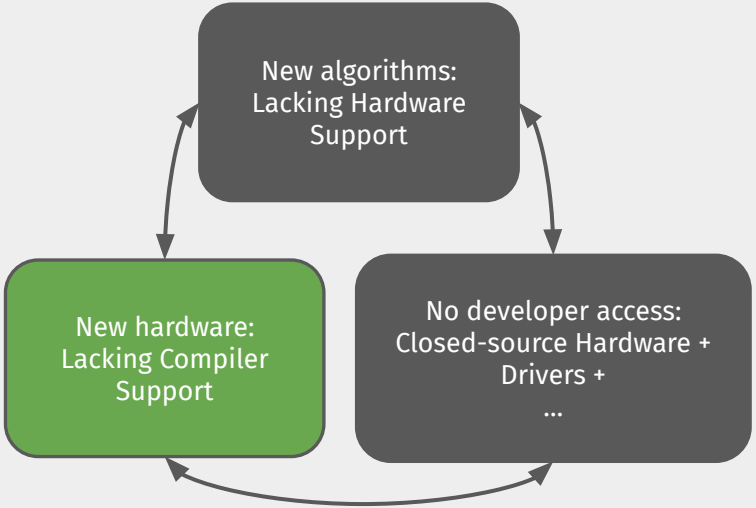
The programmable hardware synthesis flow



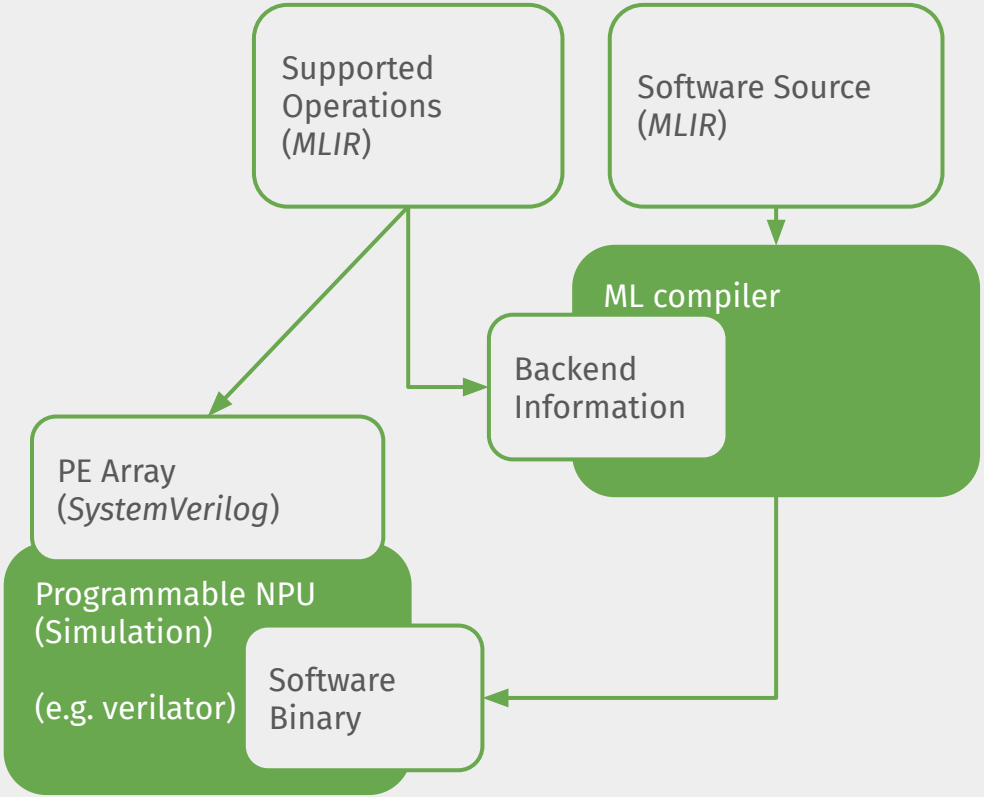


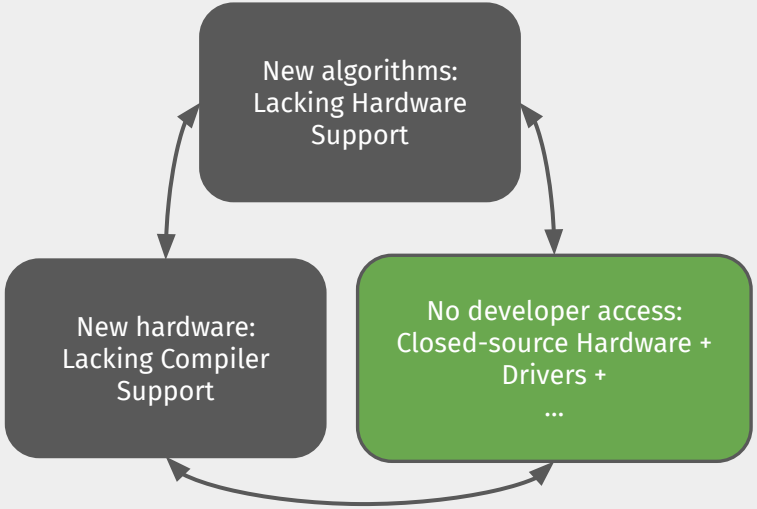
The programmable hardware synthesis flow



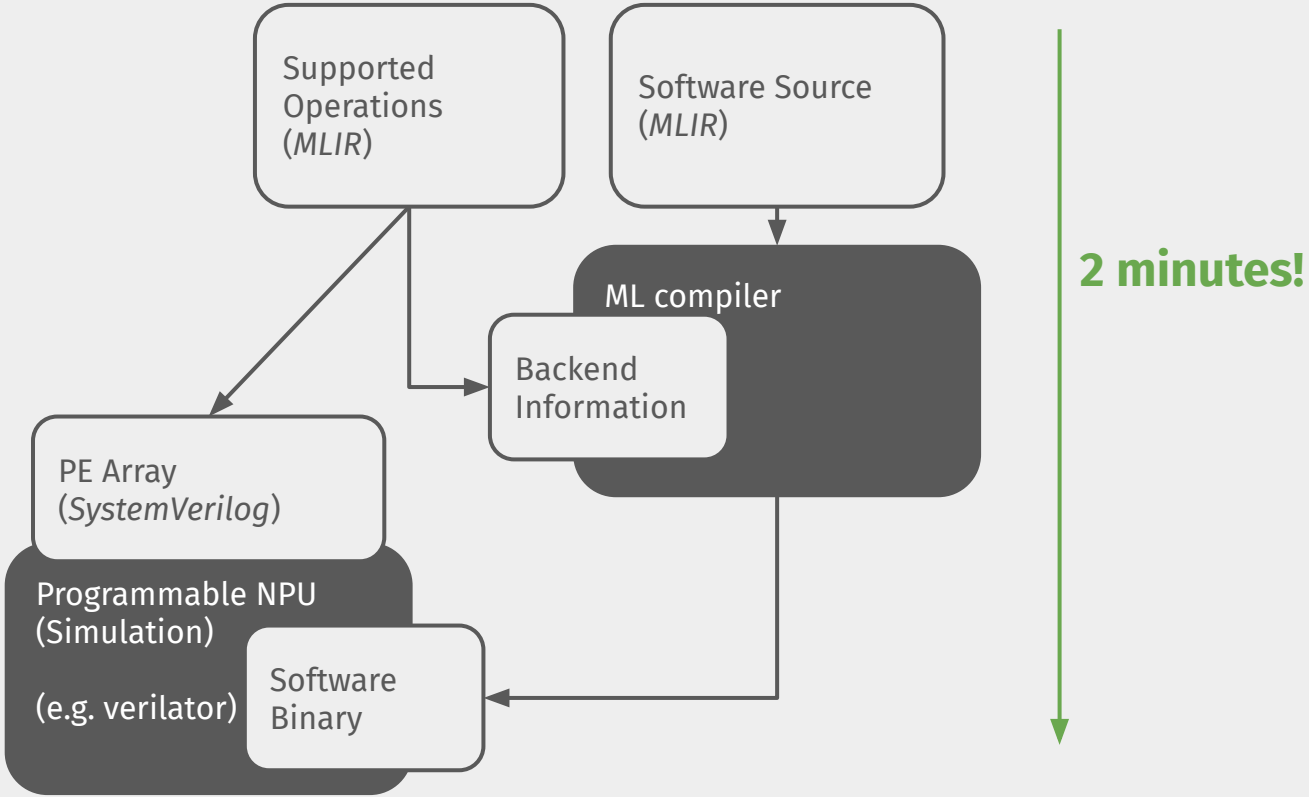


The programmable hardware synthesis flow

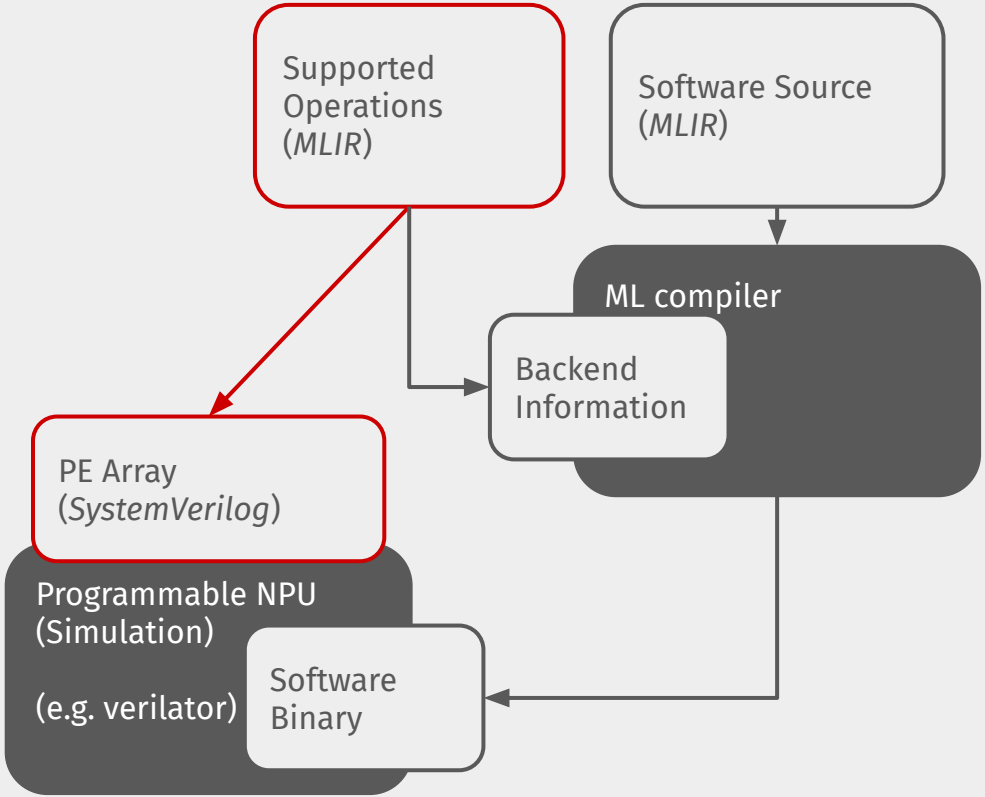




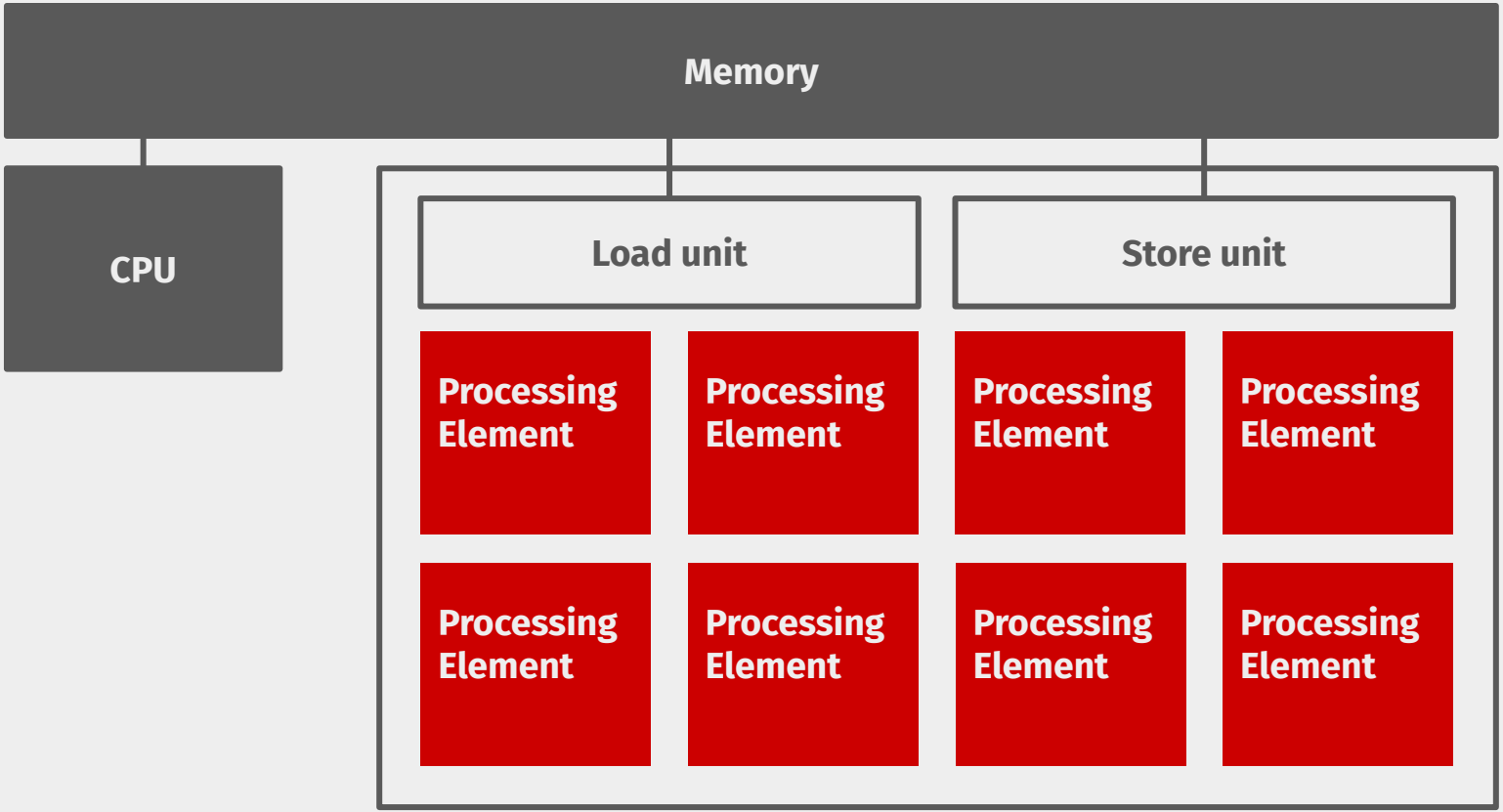
The programmable hardware synthesis flow



Today we'll talk about PE array conversion

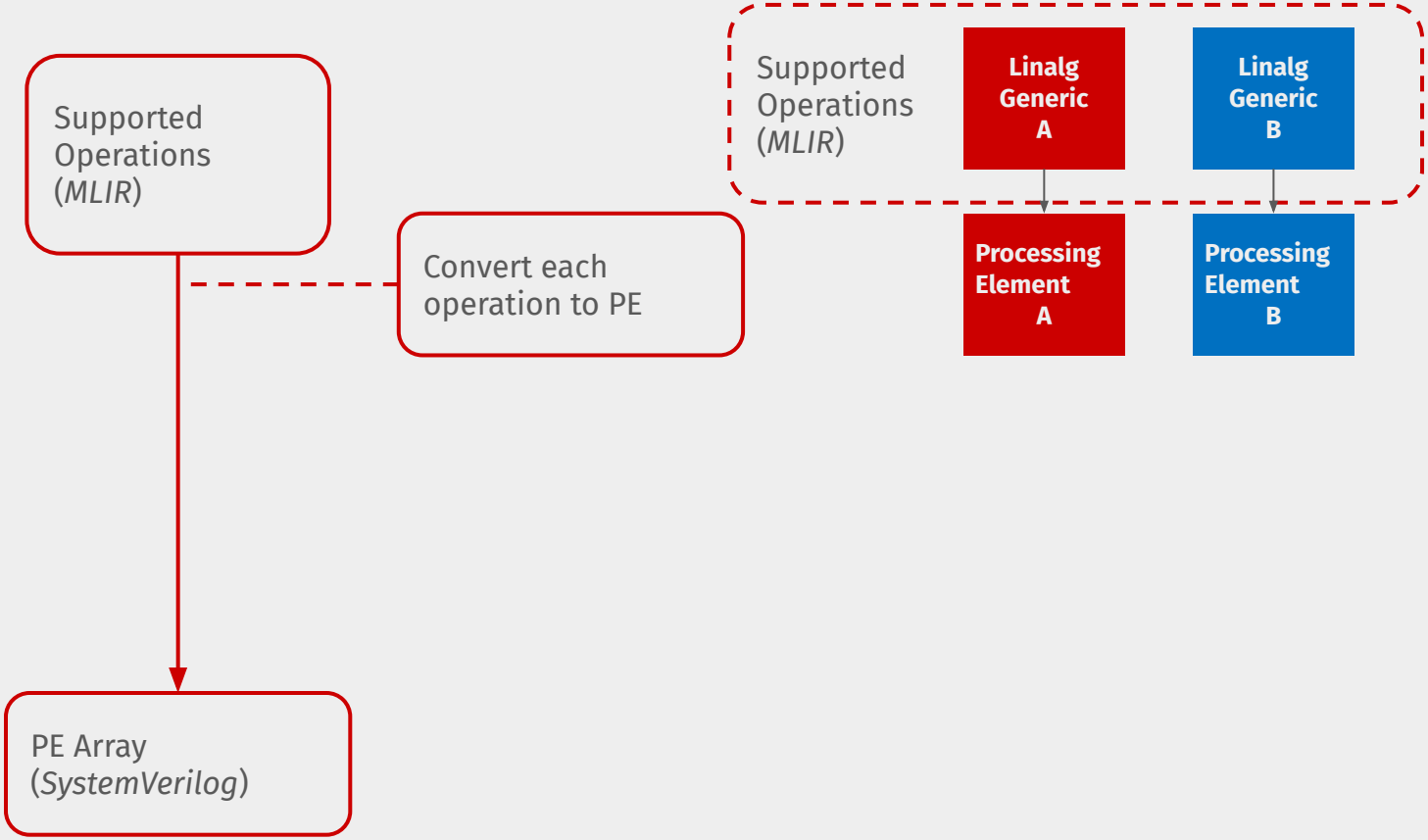


Today we'll talk about PE array conversion

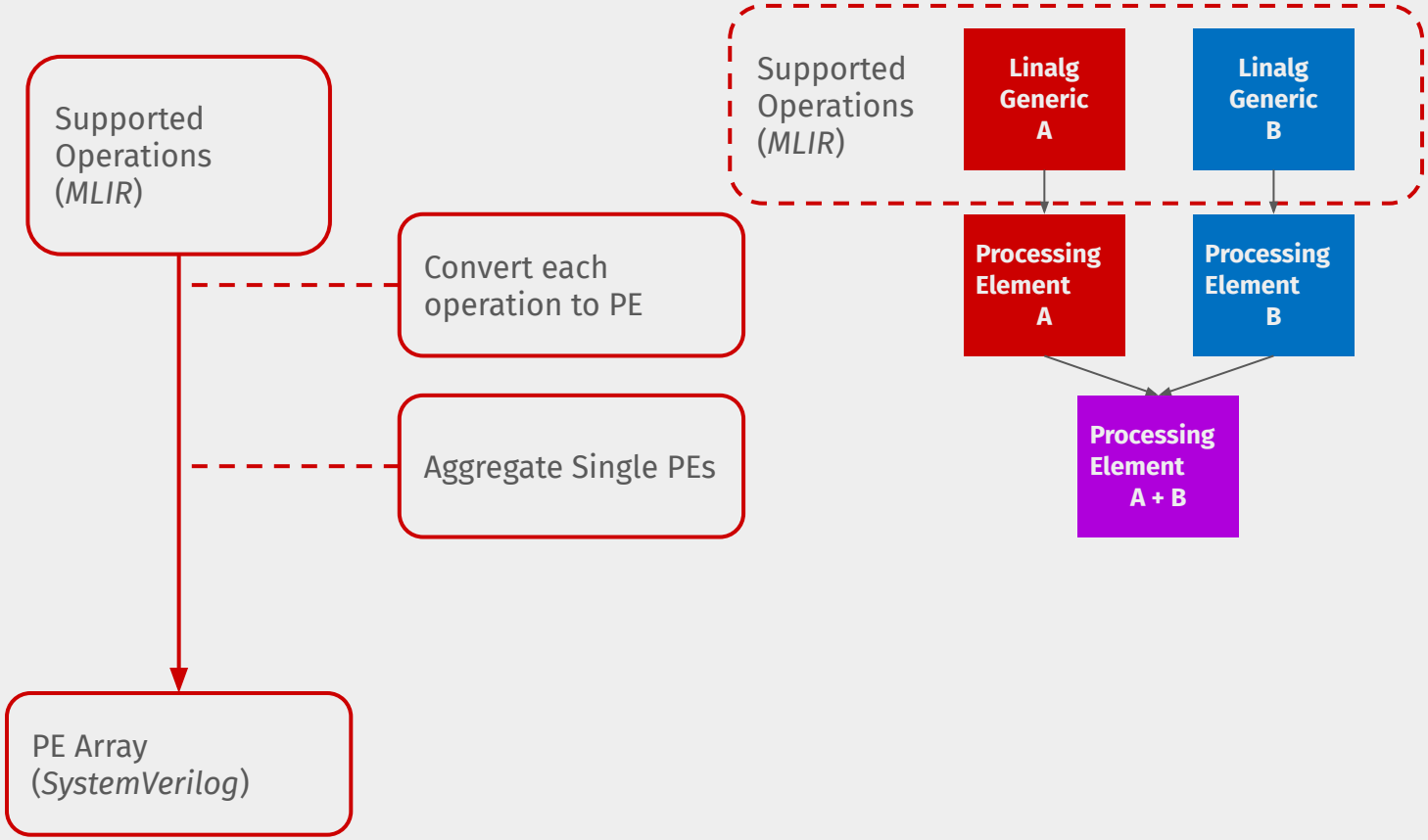


Create a PE Array for these two supported operations

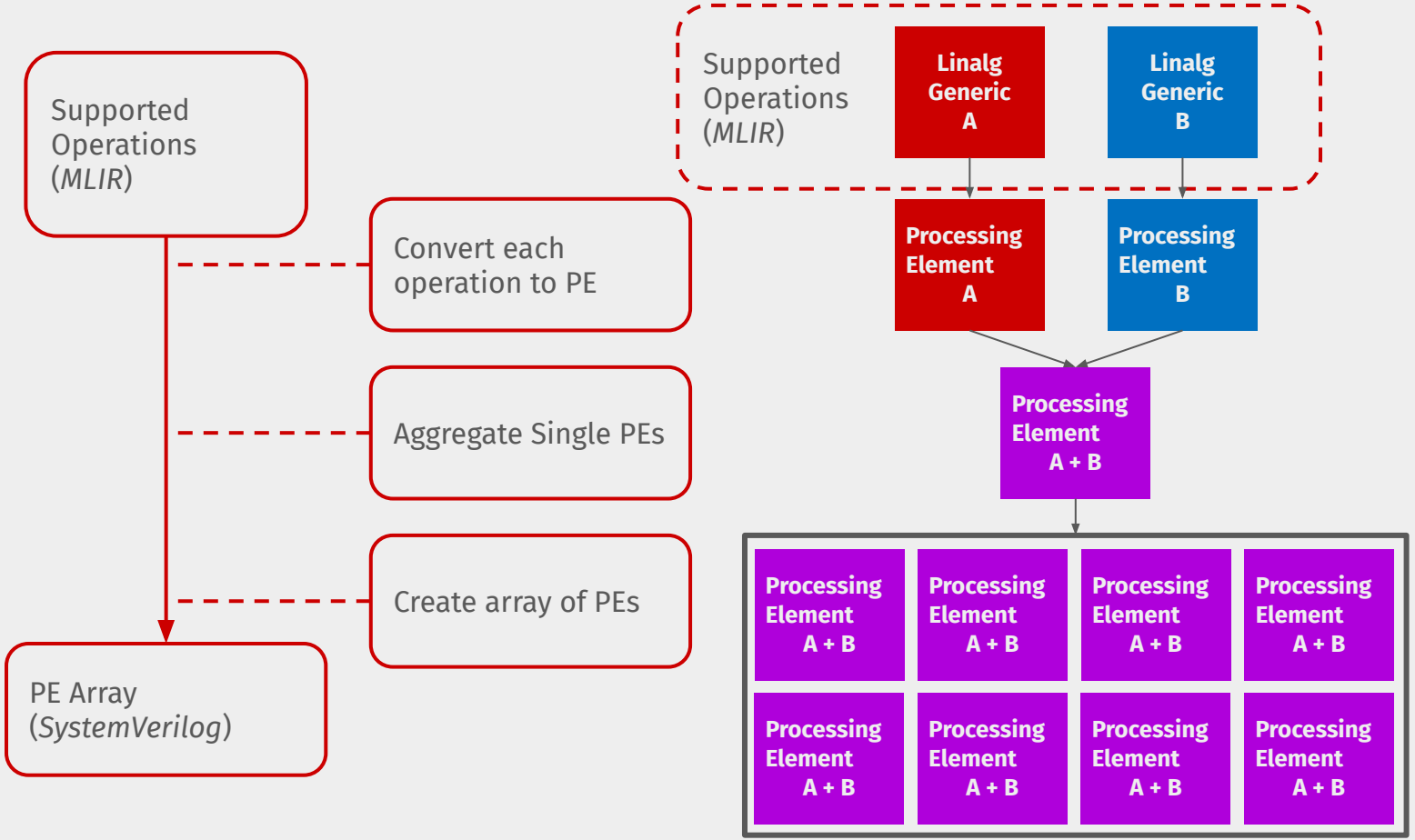
Create a PE Array for these two supported operations



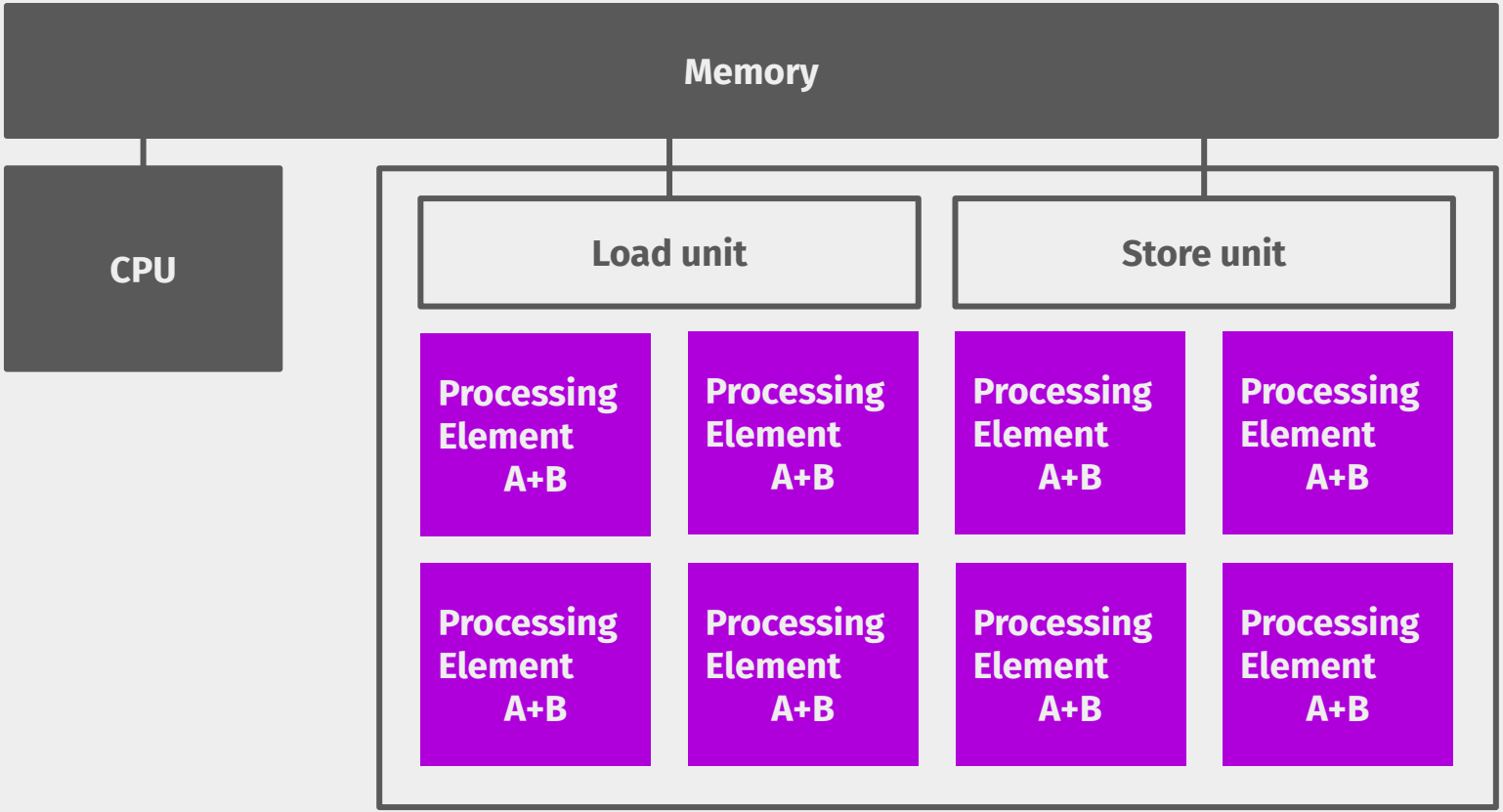
Create a PE Array for these two supported operations



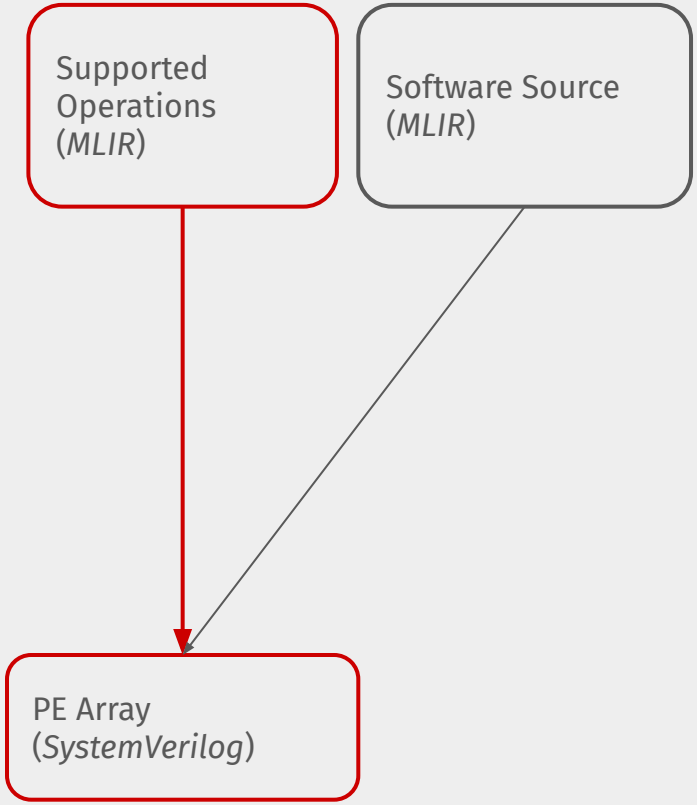
Create a PE Array for these two supported operations

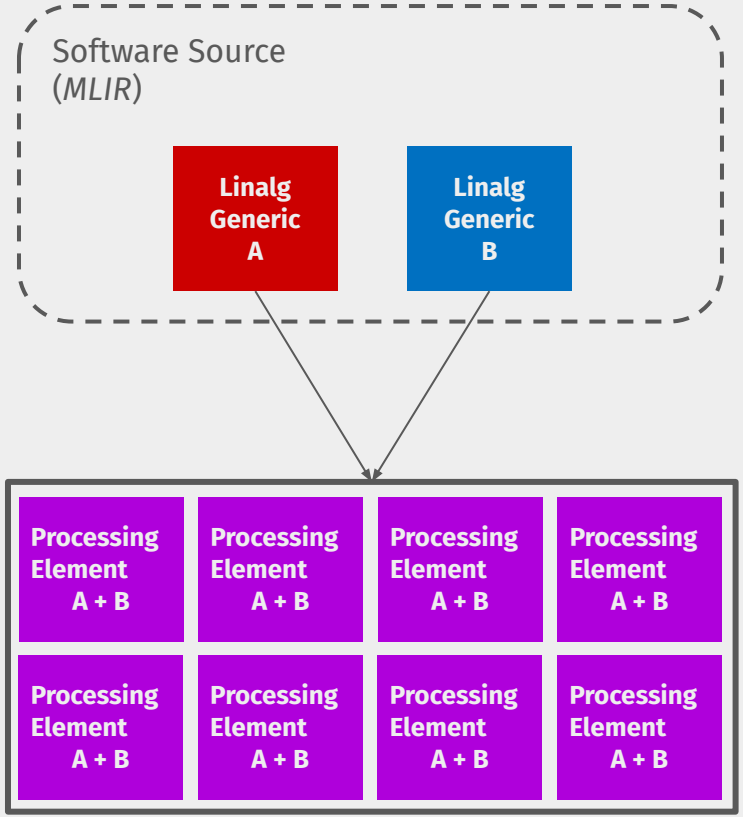


This NPU now supports linalg.generic A and B



If the source contains supported operations, they are guaranteed to work on this NPU!





Examples

Simple example no. 1

Create a 4-way parallel PE array that supports Elementwise operations:

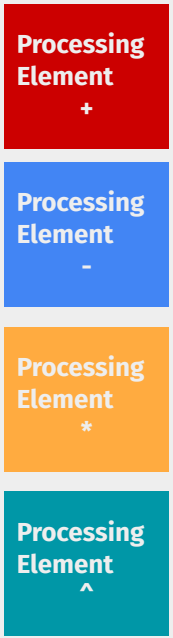
- Addition
- Subtraction
- Multiplication
- Bitwise Exclusive OR

We want to support the following operations

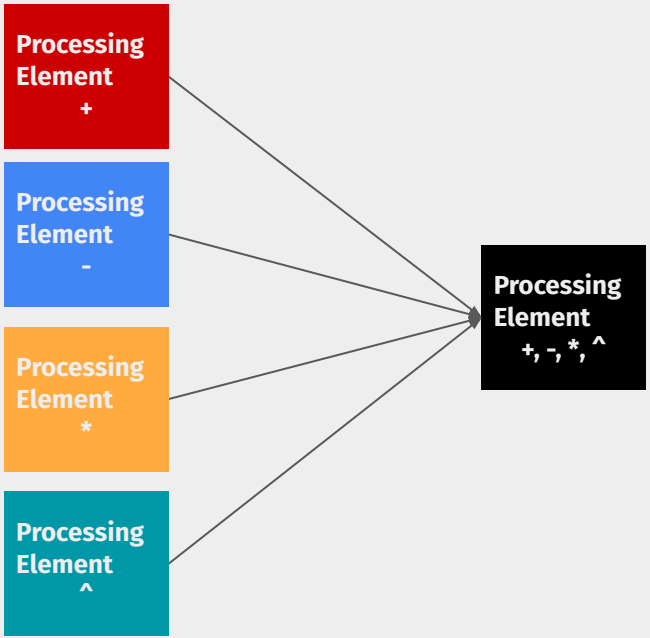
```
#map = affine_map<(d0) -> (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) -> tensor<16xi64> {
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.add ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) ->
tensor<16xi64>
   %2 = linalg.sub ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) -> tensor<16xi64>
   %3 = linalg.mul ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) -> tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]} ins(%arg0, %3 :
tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```

Create a single PE for each supported operation

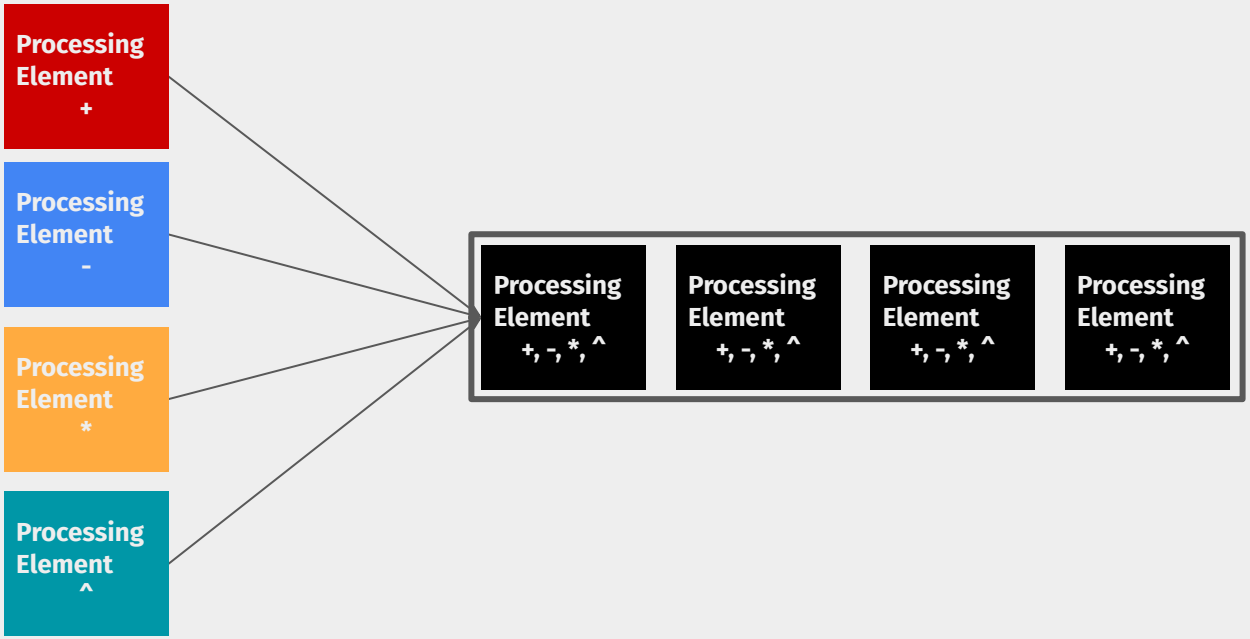
```
#map = affine_map<(d0) → (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) → tensor<16xi64> {
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.addi %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %2 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.subi %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %3 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.muli %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %3 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```



Aggregate PEs into one



Create a 4-way array



Now deploy the algorithm on simple accelerator no.1

Create a 4-way parallel PE array that supports Elementwise operations:

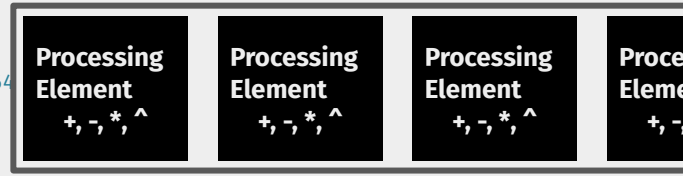
- Addition
- Subtraction
- Multiplication
- Bitwise Exclusive OR

Deploy this on the simple 4-way array

```
#map = affine_map<(d0) -> (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) -> tensor<16xi64> {
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.add ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) ->
tensor<16xi64>
   %2 = linalg.sub ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) -> tensor<16xi64>
   %3 = linalg.mul ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) -> tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]} ins(%arg0, %3 :
tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```

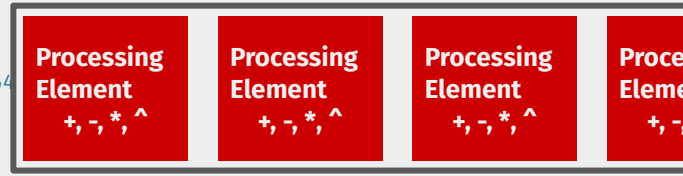
Deploy addition

```
#map = affine_map<(d0) -> (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) -> tensor<16xi64
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.addi %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %2 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.subi %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %3 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.muli %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %3 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```



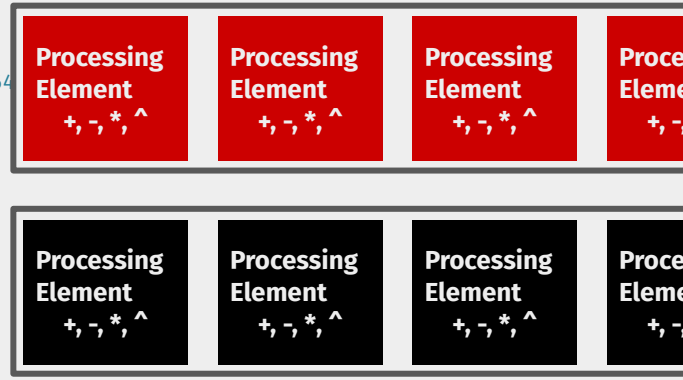
Deploy addition

```
#map = affine_map<(d0) -> (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) -> tensor<16xi64
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.addi %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %2 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.subi %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %3 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.muli %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %3 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```



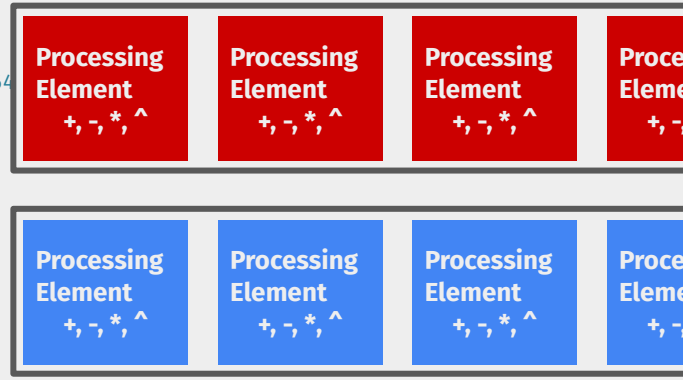
Deploy subtraction

```
#map = affine_map<(d0) -> (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) -> tensor<16xi64
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.addi %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %2 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.subi %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %3 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.muli %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %3 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```



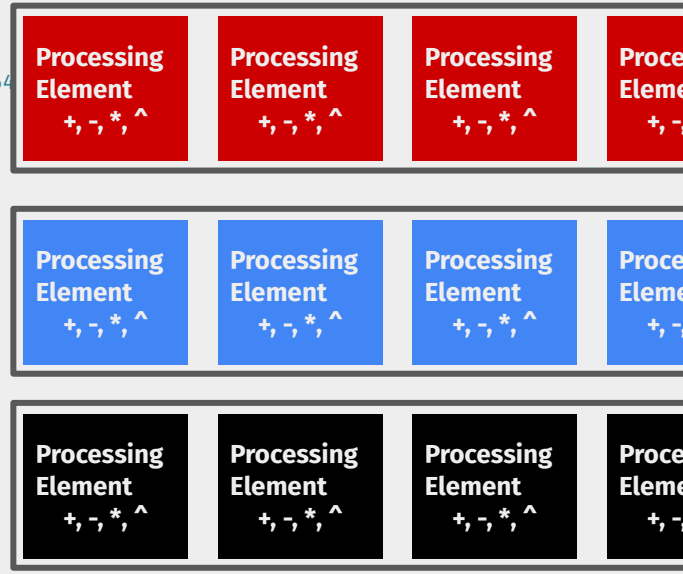
Deploy subtraction

```
#map = affine_map<(d0) → (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) → tensor<16xi64
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.addi %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %2 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.subi %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %3 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.muli %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %3 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```



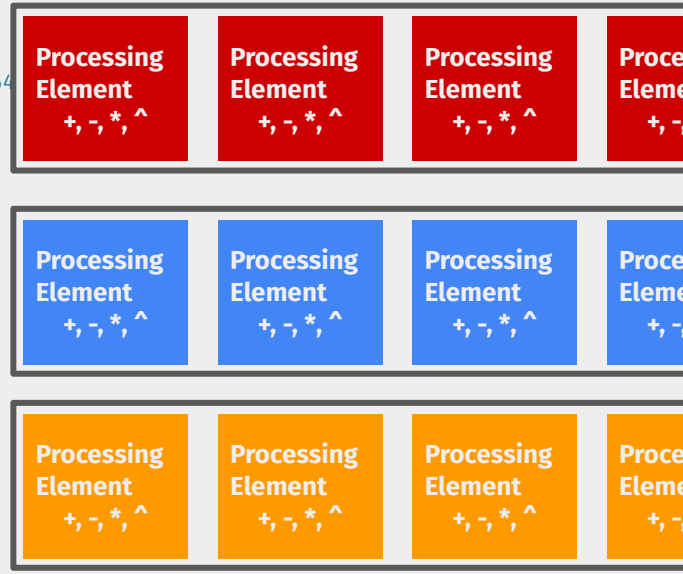
Deploy multiplication

```
#map = affine_map<(d0) -> (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) -> tensor<16xi64
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.addi %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %2 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.subi %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %3 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.muli %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %3 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```



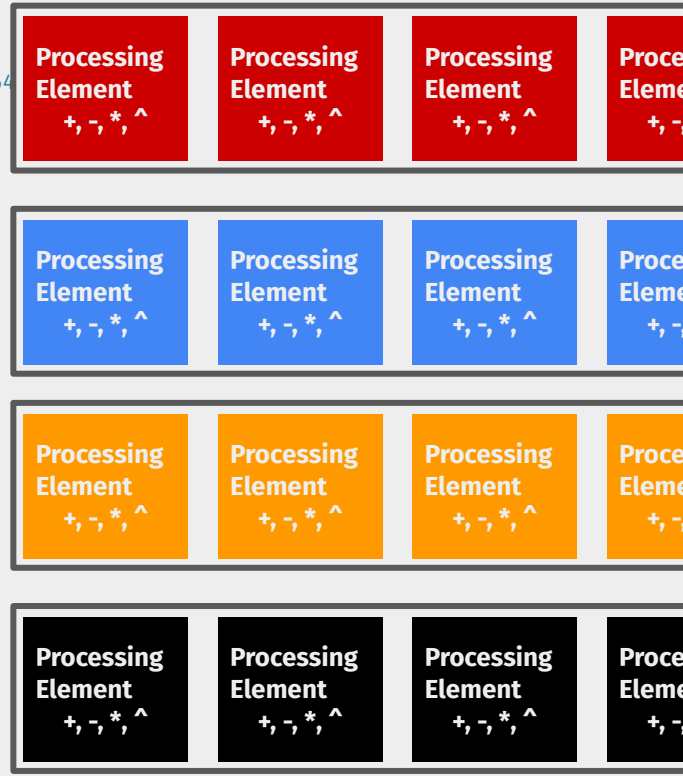
Deploy multiplication

```
#map = affine_map<(d0) → (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) → tensor<16xi64
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.addi %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %2 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.subi %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %3 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.muli %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %3 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```



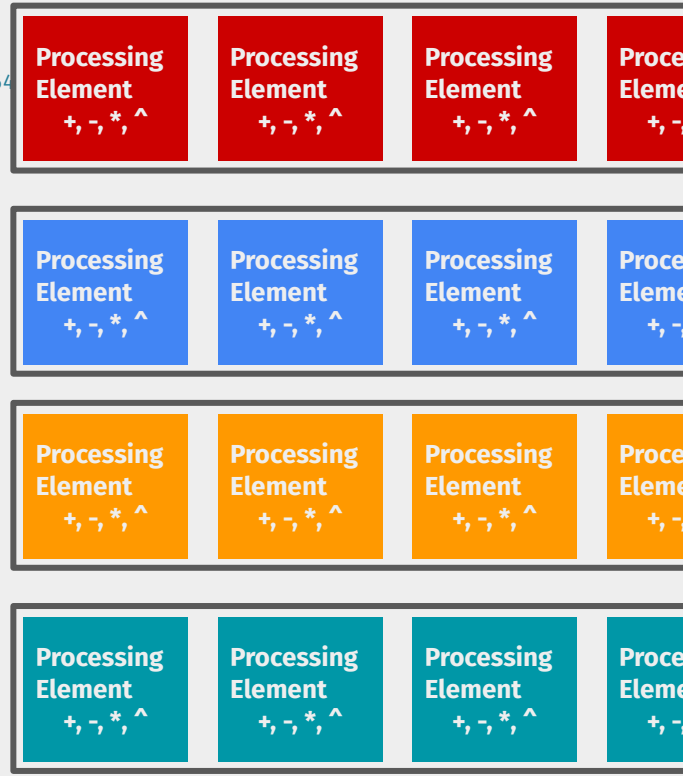
Deploy exclusive or

```
#map = affine_map<(d0) → (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) → tensor<16xi64
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.addi %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %2 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.subi %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %3 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.muli %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %3 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```



Deploy exclusive or

```
#map = affine_map<(d0) → (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) → tensor<16xi64
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.addi %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %2 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.subi %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %3 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.muli %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
        ins(%arg0, %3 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } → tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```



Performance

Load data * 4

Store data * 4

Program accelerator * 4

Simple example no. 2 operator fusion

We can make this much faster if we support operator fusion

- Addition
- Subtraction
- Multiplication
- Bitwise Exclusive OR

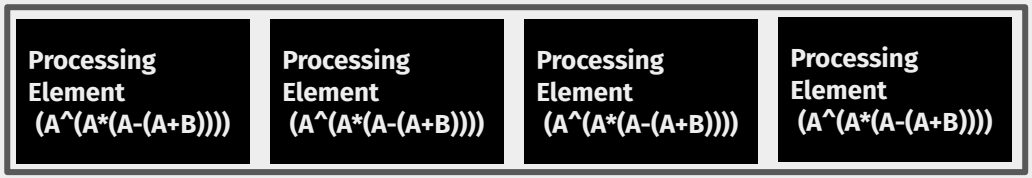
(A^(A*(A-(A+B))))

Deploy this on the simple 4-way array

```
#map = affine_map<(d0) -> (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) -> tensor<16xi64> {
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.add ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) ->
tensor<16xi64>
   %2 = linalg.sub ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) -> tensor<16xi64>
   %3 = linalg.mul ins(%arg0, %2 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) -> tensor<16xi64>
   %4 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]} ins(%arg0, %3 :
tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %5 = arith.xori %in, %in_0 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   return %4 : tensor<16xi64>
 }
}
```

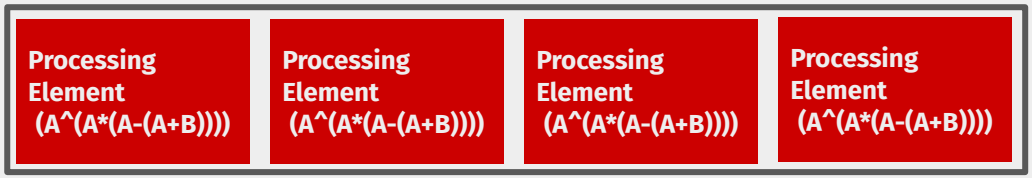
Deploy this on the simple 4-way array

```
#map = affine_map<(d0) -> (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) -> tensor<16xi64> {
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]} ins(%arg0, %arg1 :
tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %2 = arith.addi %in, %in_0 : i64
     %3 = arith.subi %in, %2 : i64
     %4 = arith.muli %in, %3 : i64
     %5 = arith.xori %in, %4 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   return %1 : tensor<16xi64>
 }
}
```



Deploy this on the simple 4-way array

```
#map = affine_map<(d0) -> (d0)>
module {
 func.func public @streamer_add(%arg0: tensor<16xi64>, %arg1: tensor<16xi64>) -> tensor<16xi64> {
   %0 = tensor.empty() : tensor<16xi64>
   %1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]} ins(%arg0, %arg1 :
tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
   ^bb0(%in: i64, %in_0: i64, %out: i64):
     %2 = arith.addi %in, %in_0 : i64
     %3 = arith.subi %in, %2 : i64
     %4 = arith.muli %in, %3 : i64
     %5 = arith.xori %in, %4 : i64
     linalg.yield %5 : i64
   } -> tensor<16xi64>
   return %1 : tensor<16xi64>
 }
}
```



Performance

Store data * 1

Load data * 1

Program accelerator * 1

Discussion

Tradeoff - More specific accelerators or more performance?

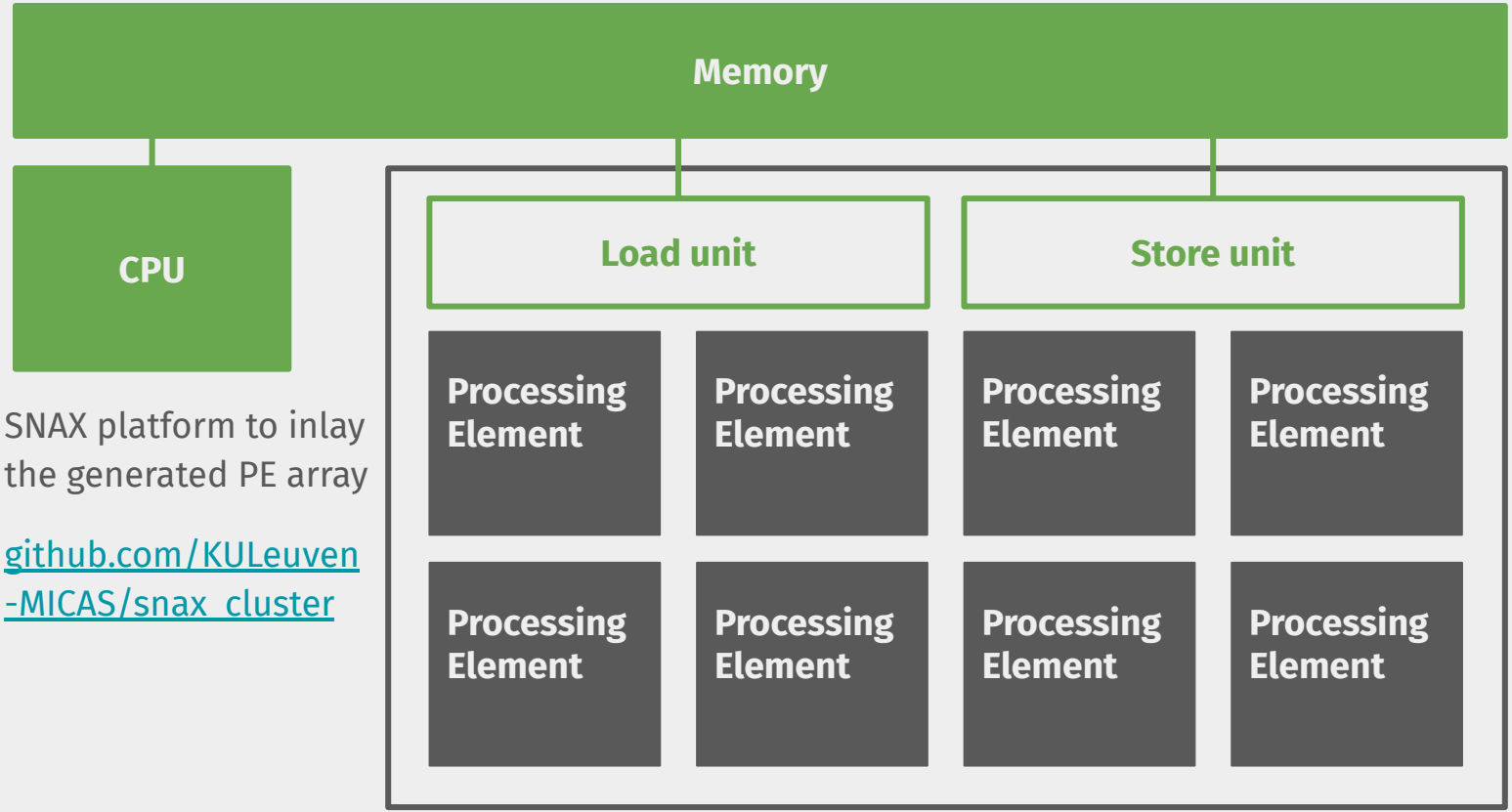
How did we create this?

Shortcut no. 1 - xDSL for quick prototyping

xDSL, an open-source MLIR, CIRCT compatible framework in Python, to quickly prototype new MLIR dialects and combinations of compiler passes.

github.com/xdslproject/xdsl

Shortcut no. 2 - PE array + SNAX platform = NPU



SNAX platform to inlay the generated PE array

github.com/KULeuven-MICAS/snax_cluster

Shortcut no. 3 - SNAX-MLIR sw compilation toolchain

- Compatible with RISC-V control core (`rv32imc`)

- `linalg.generic` → loads/stores + program accelerators with dynamic backends

github.com/KULeuven-MICAS/snax-mlir

In the future

- Overlay functionality on parts of the PE
- Support for multiple dataflows
- Multi-accelerator support
- Support for multi-cycle hardware (ready/valid)
- Floating-point support (not because I like FP, rather because I don’t like quantization)

Thank you!
Questions?

josse:matrix.org

linkedin.com/in/jossevandelm/

github.com/JosseVanDelm

vandelm.com

Takeaways of this talk

- ***NPUs** are dedicated processors to accelerate **dense linear algebra***
- *Properly using and and designing NPUS them requires an **overhaul** of the **design process***
- *We leverage MLIR, CIRCT, xDSL and SNAX to quickly create new NPUs and compilers together through **programmable hardware synthesis***

References:

[1] “TPU v4” by Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson licensed under Creative Commons Attribution 4.0 International license

[2] “Apple M1” by Henriok licensed under the Public Domain

[3] “AMD BC-160” by Мой Компьютер licensed under Creative Commons Attribution 3.0 Unported license

[4] “NVIDIA H100” by Geekerwan licensed under Creative Commons Attribution 3.0 Unported

[5] “Microprocessor Trend Data” by “Karl Rupp” licensed under Creative Commons Attribution 4.0 International Public License

[6] Jouppi, Norman P., et al. "In-datacenter performance analysis of a tensor processing unit." Proceedings of the 44th annual international symposium on computer architecture. 2017.

Creating the hardware with the PHS dialect

A new dialect to create **programmable*** hardware

`phs.pe`:
An operation that represents what a PE of the NPU can do

`phs.choose`:
A *named* operation that represents a choice of operation in different MLIR operations

`phs.mux`:
“multiplexer” - an *unnamed* operation operation that represents a choice of dataflow for the PE

`phs.yield`:
a terminator operation similar to `scf.yield` and `linalg.yield`.

To go from linalg to a programmable PE

1. Convert each **`linalg`** to a **`linalg.generic`**
2. Convert each **`linalg.generic`** to a **`phs.pe`** op
3. Aggregate all **`phs.pe`** ops through their named **`phs.choose_ops`** and insert **`phs.mux_ops`** where necessary
4. Generate an array of **`phs.pe`** elements
5. Convert the aggregated **`phs.pe`** to systemverilog with CIRCT

2) Convert each **linalg.generic** to a **phs.pe** op

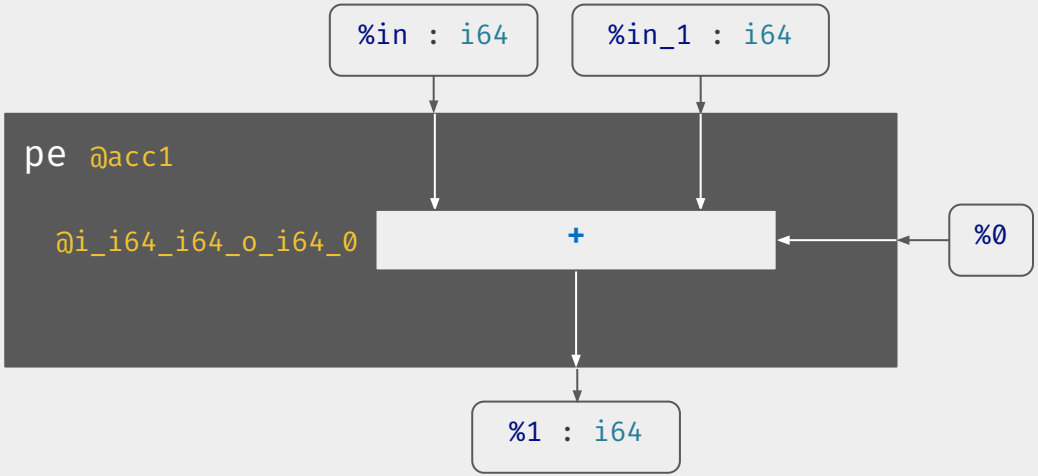
```
%1 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
ins(%arg0, %arg1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
^bb0(%in: i64, %in_0: i64, %out: i64):
 %5 = arith.addi %in, %in_0 : i64
 linalg.yield %5 : i64
} → tensor<16xi64>
```



```
phs.pe @acc1 with %0 (%in : i64, %in_1 : i64) {
 %1 = phs.choose @i_i64_i64_o_i64_0 with %0 (%in : i64, %in_1 : i64) → i64
   0) {
     %2 = arith.addi %in, %in_1 : i64
     phs.yield %2 : i64
   }
 phs.yield %1 : i64
```

2) Convert each **linalg.generic** to a **phs.pe** op

```
phs.pe @acc1 with %0 (%in : i64, %in_1 : i64) {
 %1 = phs.choose @i_i64_i64_o_i64_0 with %0 (%in : i64, %in_1 : i64) → i64
   0) {
     %2 = arith.addi %in, %in_1 : i64
     phs.yield %2 : i64
   }
 phs.yield %1 : i64
```



2) Convert each **linalg.generic** to a **phs.pe** op

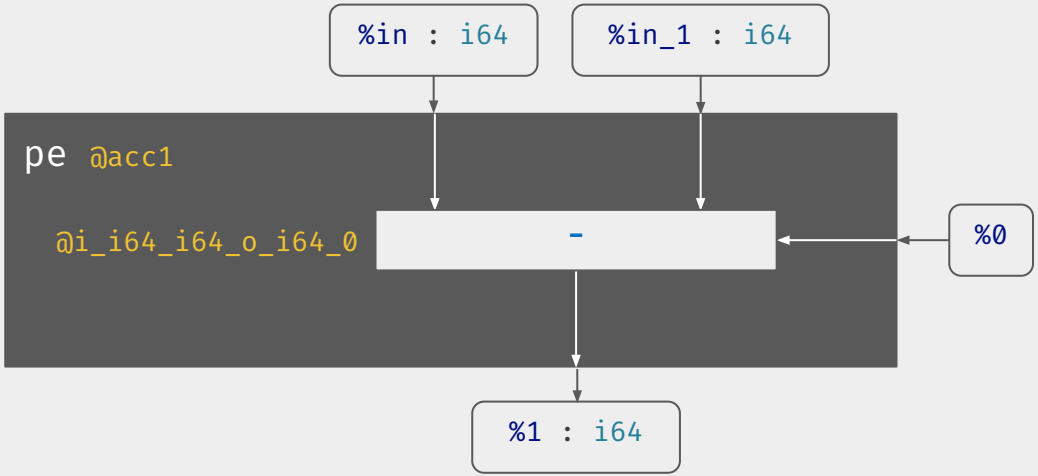
```
%2 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
ins(%arg0, %1 : tensor<16xi64>, tensor<16xi64>) outs(%0 : tensor<16xi64>) {
^bb0(%in: i64, %in_0: i64, %out: i64):
 %5 = arith.subi %in, %in_0 : i64
 linalg.yield %5 : i64
} → tensor<16xi64>
```



```
phs.pe @acc1 with %0 (%in : i64, %in_1 : i64) {
 %1 = phs.choose @i_i64_i64_o_i64_0 with %0 (%in : i64, %in_1 : i64) → i64
   0) {
     %2 = arith.subi %in, %in_1 : i64
     phs.yield %2 : i64
   }
 phs.yield %1 : i64
```

2) Convert each **linalg.generic** to a **phs.pe** op

```
phs.pe @acc1 with %0 (%in : i64, %in_1 : i64) {
 %1 = phs.choose @i_i64_i64_o_i64_0 with %0 (%in : i64, %in_1 : i64) → i64
   0) {
     %2 = arith.subi %in, %in_1 : i64
     phs.yield %2 : i64
   }
 phs.yield %1 : i64
```



3) Aggregate all **phs.pe** ops through their named **phs.choose_ops** and insert **phs.mux_ops** where necessary

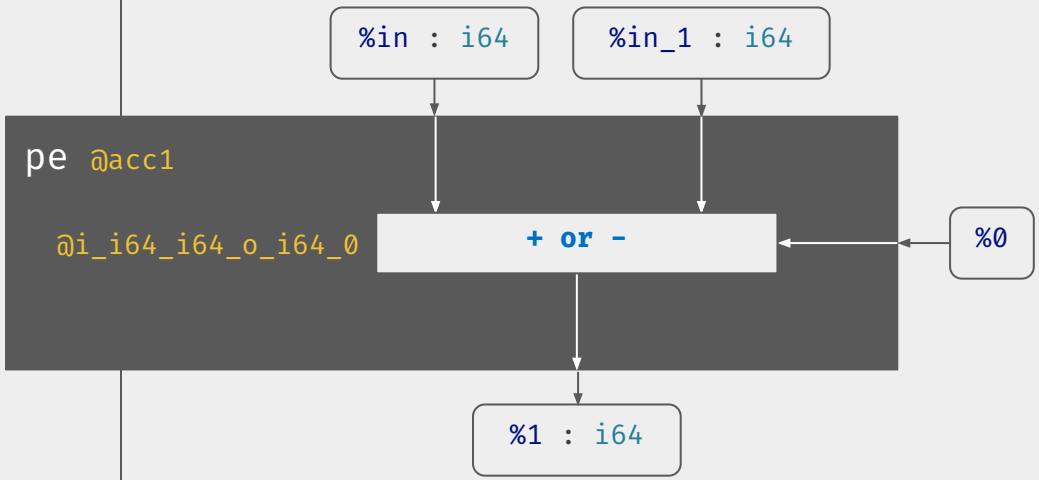
```
phs.pe @acc1 with %0 (%in : i64, %in_1 : i64) {
 %1 = phs.choose @i_i64_i64_o_i64_0 with %0
      (%in : i64, %in_1 : i64) → i64
   0) {
     %2 = arith.addi %in, %in_1 : i64
     phs.yield %2 : i64
   }
 phs.yield %1 : i64
```

```
phs.pe @acc1 with %0 (%in : i64, %in_1 : i64) {
 %1 = phs.choose @i_i64_i64_o_i64_0 with %0
 (%in : i64, %in_1 : i64) → i64
   0) {
     %2 = arith.subi %in, %in_1 : i64
     phs.yield %2 : i64
   }
 phs.yield %1 : i64
```

```
phs.pe @acc1 with %0 (%in : i64, %in_1 : i64) {
 %1 = phs.choose @i_i64_i64_o_i64_0 with %0
(%in : i64, %in_1 : i64) → i64
   0) {
     %2 = arith.addi %in, %in_1 : i64
     phs.yield %2 : i64
   }
   1) {
     %3 = arith.subi %in, %in_1 : i64
     phs.yield %3 : i64
   }
 phs.yield %1 : i64
}
```

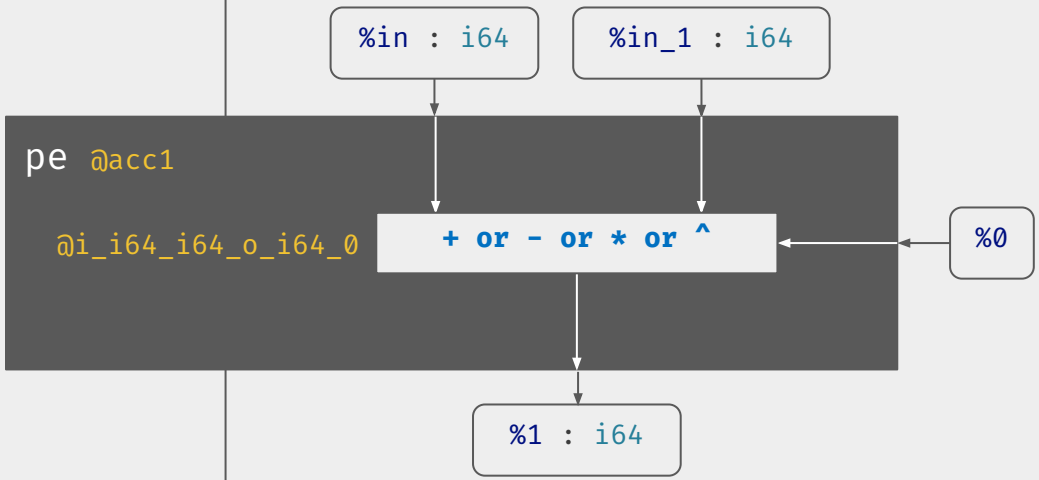
3) Aggregate all **phs.pe** ops through their named **phs.choose_ops** and insert **phs.mux_ops** where necessary

```
phs.pe @acc1 with %0 (%in : i64, %in_1 : i64) {
 %1 = phs.choose @i_i64_i64_o_i64_0 with %0
(%in : i64, %in_1 : i64) → i64
   0) {
     %2 = arith.addi %in, %in_1 : i64
     phs.yield %2 : i64
   }
   1) {
     %3 = arith.subi %in, %in_1 : i64
     phs.yield %3 : i64
   }
 phs.yield %1 : i64
}
```



3) Aggregate all **phs.pe** ops through their named **phs.choose_ops** and insert **phs.mux_ops** where necessary

```
phs.pe @acc1 with %0 (%in : i64, %in_1 : i64) {
 %1 = phs.choose @i_i64_i64_o_i64_0 with %0 (%in : i64, %in_1 : i64) → i64
   0) {
     %2 = arith.addi %in, %in_1 : i64
     phs.yield %2 : i64
   }
   1) {
     %3 = arith.subi %in, %in_1 : i64
     phs.yield %3 : i64
   }
   2) {
     %4 = arith.muli %in, %in_1 : i64
     phs.yield %4 : i64
   }
   3) {
     %5 = arith.xori %in, %in_1 : i64
     phs.yield %5 : i64
   }
 phs.yield %1 : i64
}
```



```
hw.module private @acc1(in %in data_0: i64, in %in_1 data_1: i64,
                        in %0 switch_0: i2, out out_0: i64) {
  %1 = arith.addi %in, %in_1 : i64
  %2 = arith.subi %in, %in_1 : i64
  %3 = arith.muli %in, %in_1 : i64
  %4 = arith.xori %in, %in_1 : i64
  %5 = hw.array_create %4, %3, %2, %1 : i64
  %6 = hw.array_get %5[%0] : !hw.array<4xi64>, i2
  hw.output %6 : i64
}
hw.module @acc1_array(in %0 data_0: !hw.array<4xi64>,
                      in %1 data_1: !hw.array<4xi64>,
                      in %2 switch_0: i2, out out_0:
!hw.array<4xi64>) {
  %3 = arith.constant 0 : i2
  %4 = hw.array_get %0[%3] : !hw.array<4xi64>, i2
  %5 = arith.constant 0 : i2
  %6 = hw.array_get %1[%5] : !hw.array<4xi64>, i2
  %7 = hw.instance "acc1_pe_0" @acc1(data_0: %4: i64,
       data_1: %6: i64, switch_0: %2: i2) -> (out_0: i64)
  %8 = arith.constant 1 : i2
  %9 = hw.array_get %0[%8] : !hw.array<4xi64>, i2
  %10 = arith.constant 1 : i2
  %11 = hw.array_get %1[%10] : !hw.array<4xi64>, i2
  %12 = hw.instance "acc1_pe_1" @acc1(data_0: %9: i64,
       data_1: %11: i64, switch_0: %2: i2) -> (out_0: i64)
  …
  %23 = hw.array_create %22, %17, %12, %7 : i64
  hw.output %23 : !hw.array<4xi64>
}
```

4) Generate an array of **phs.pe** elements