Domain Specific Optimizations

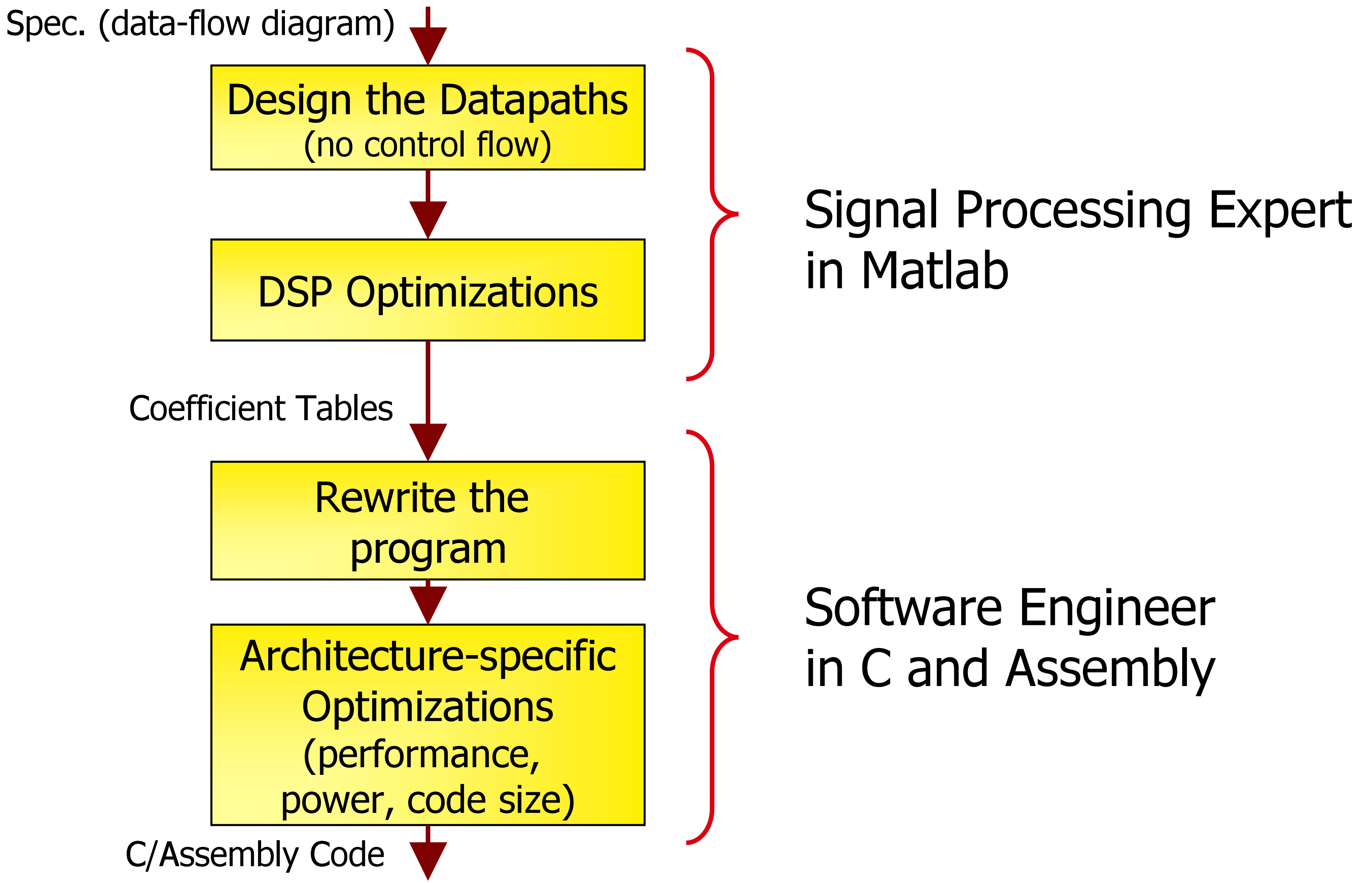
Saman Amarasinghe and William Thies
Massachusetts Institute of Technology

PACT 2003
September 27, 2003

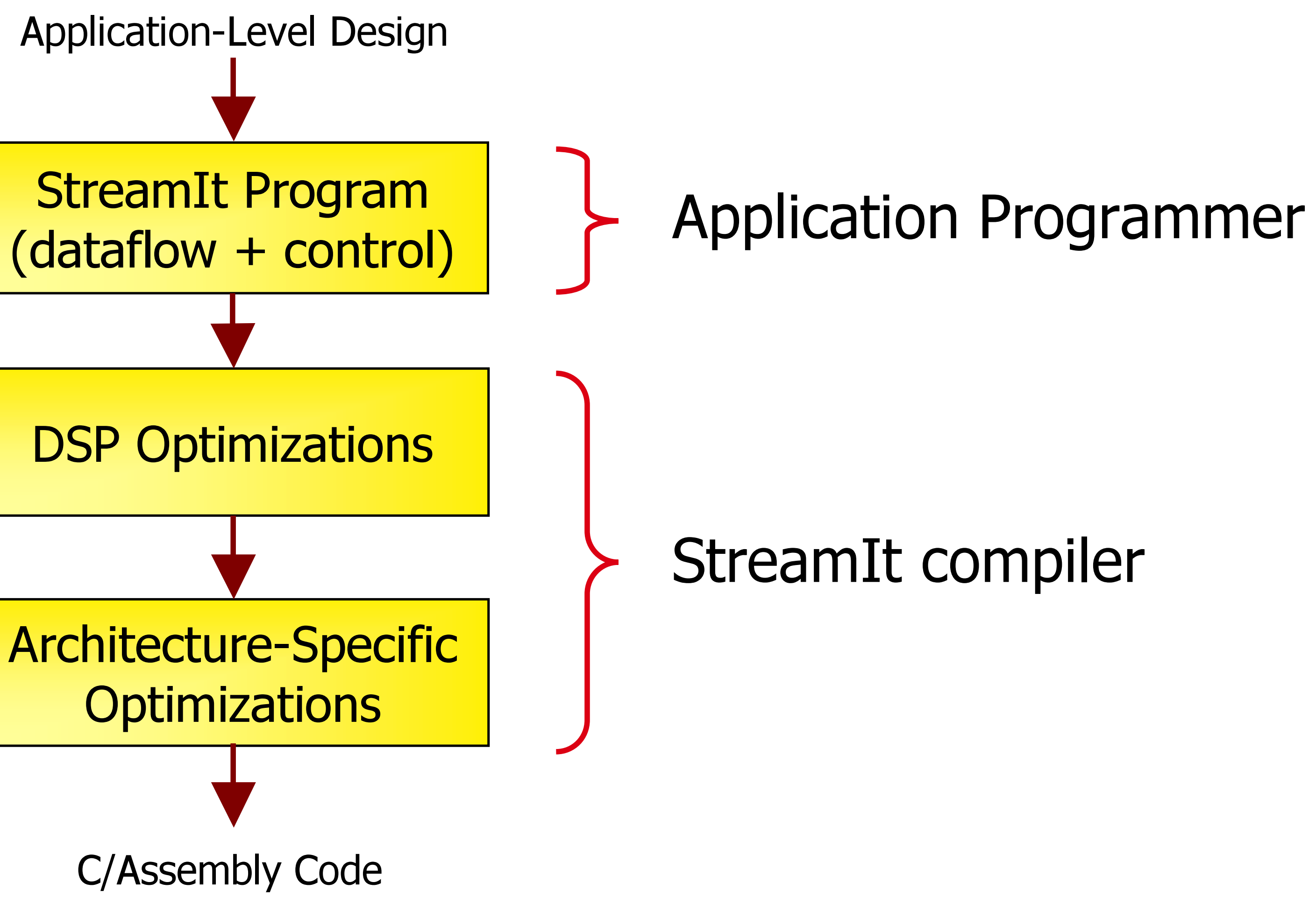
Schedule

1:30-1:40	Overview (Saman)
1:40-2:20	Stream Architectures (Saman)
2:20-3:00	Stream Languages (Bill)
3:00-3:30	Break
3:30-3:55	Stream Compilers (Saman)
3:55-4:20	Domain-specific Optimizations (Saman)
4:20-5:00	Scheduling Algorithms (Bill)

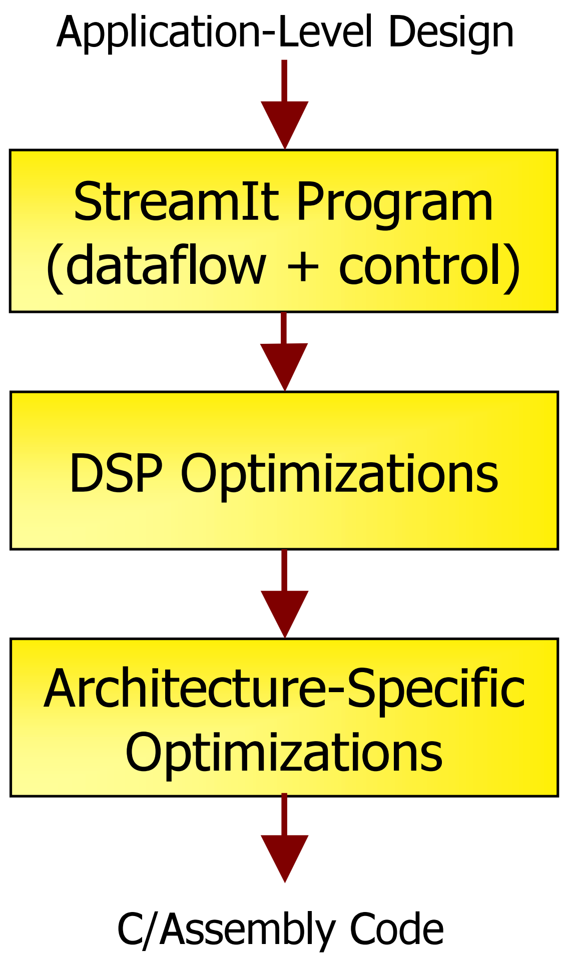
Conventional DSP Design Flow



Design Flow with StreamIt



Design Flow with StreamIt



- Benefits of programming in a single, high-level abstraction
  - Modular
  - Composable
  - Portable
  - Malleable
- The Challenge: Maintaining Performance
  - Replacing Expert DSP Engineer
  - Replacing Expert Assembly Hacker

Our Focus: Linear Filters

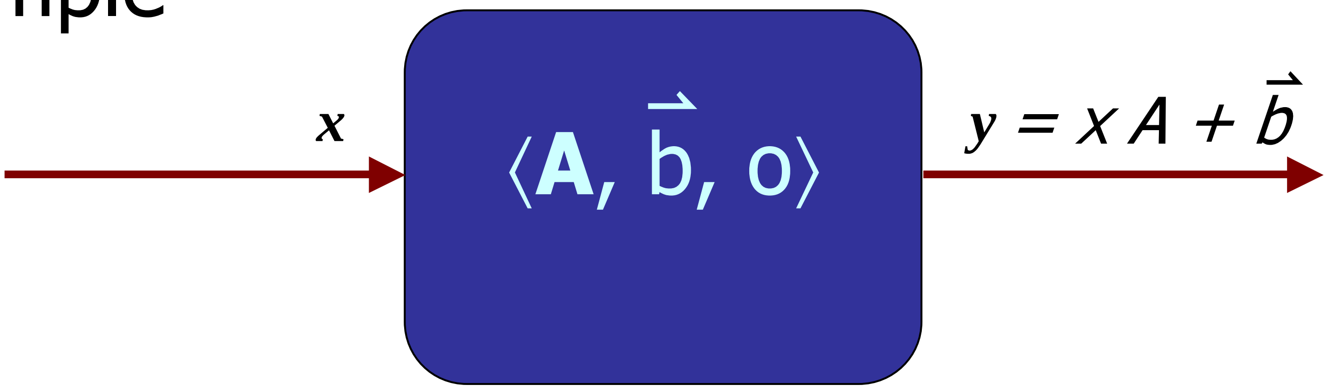
- Most common target of DSP optimizations
  - FIR filters
  - Compressors
  - Expanders
  - DFT/DCT

Output is weighted sum of inputs

- Example optimizations:
  - Combining Adjacent Nodes
  - Translating to Frequency Domain

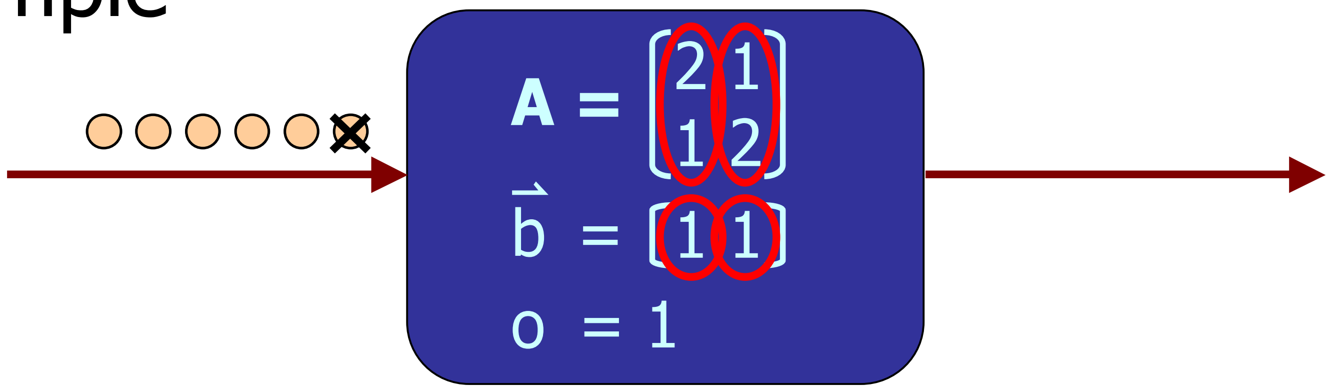
Representing Linear Filters

- A linear filter is a tuple $\langle \mathbf{A}, \vec{b}, o \rangle$
  - **A**: matrix of coefficients
  - $\vec{b}$: vector of constants
  - o: number of items popped
- Example

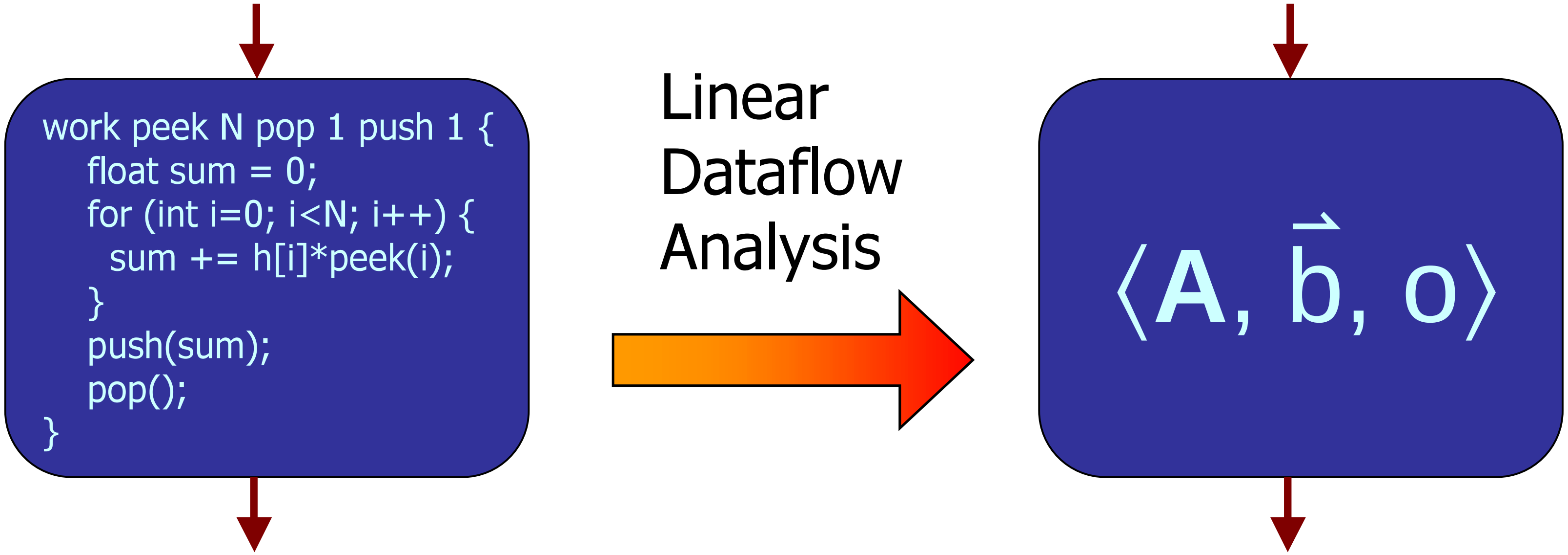


Representing Linear Filters

- A linear filter is a tuple $\langle \mathbf{A}, \vec{b}, o \rangle$
  - $\mathbf{A}$: matrix of coefficients
  - $\vec{b}$: vector of constants
  - o: number of items popped
- Example



Extracting Linear Representation



- Resembles constant propagation
- Maintains linear form $\langle \vec{v}, \vec{b} \rangle$ for each variable
  - Peek expression: generate fresh $\vec{v}$
  - Push expression: copy $\vec{v}$ into **A**
  - Pop expression: increment o

Optimizations using Linear Analysis

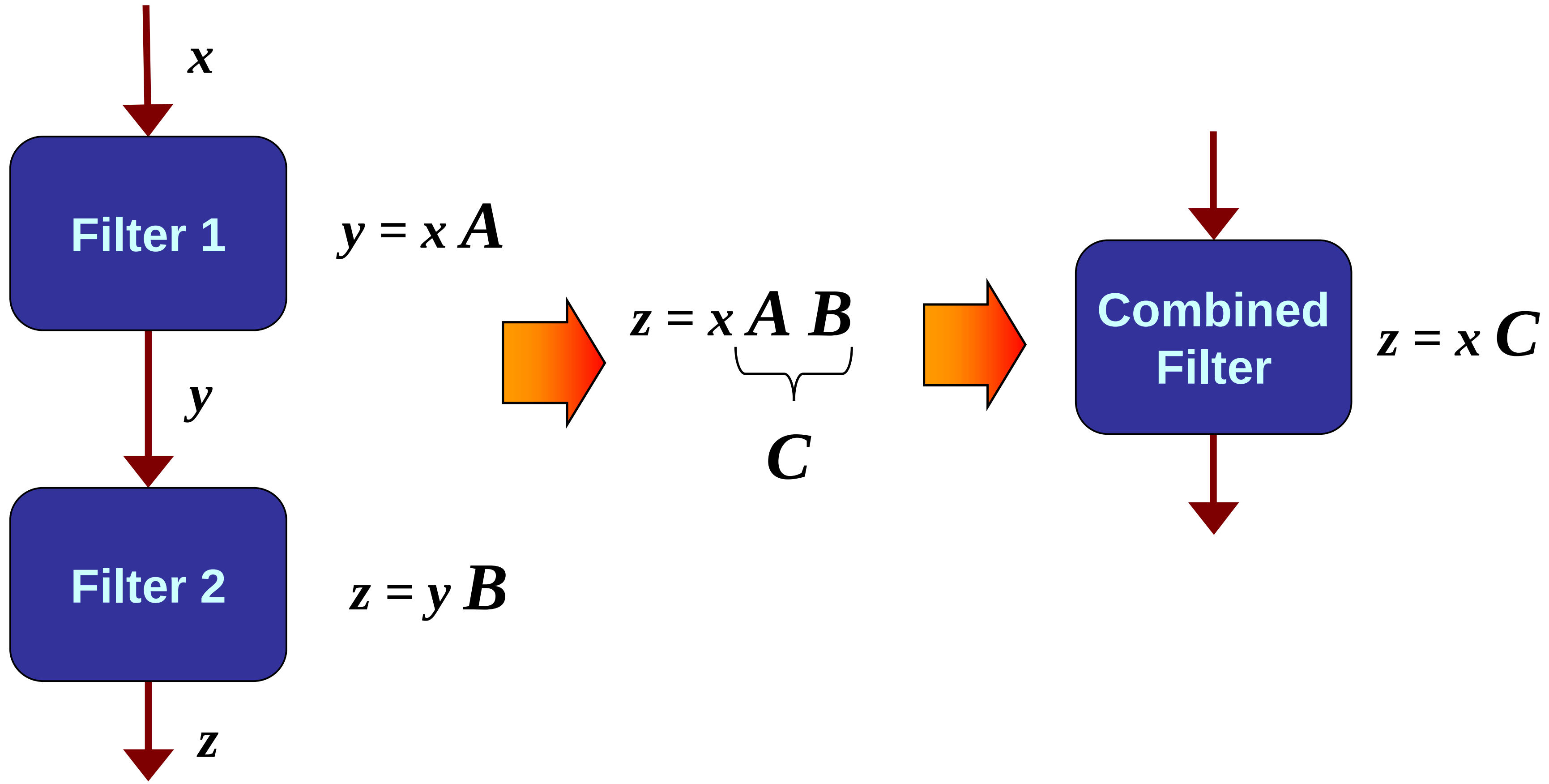
1) Combining adjacent linear structures

2) Shifting from time to the frequency domain

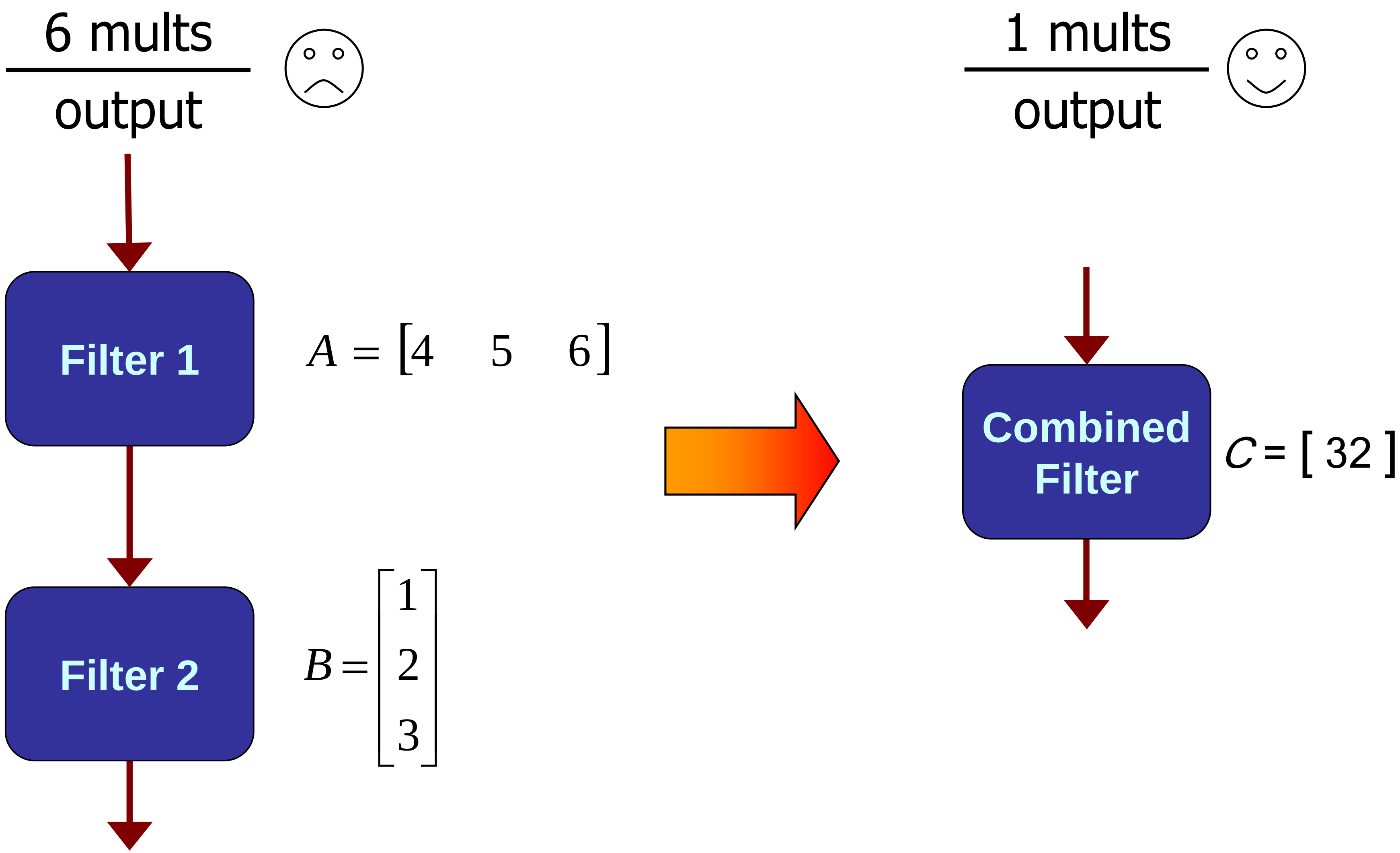
3) Selection of 'optimal' set of transformations

1) Combining Linear Filters

- Pipelines and splitjoins can be collapsed
- Example: pipeline

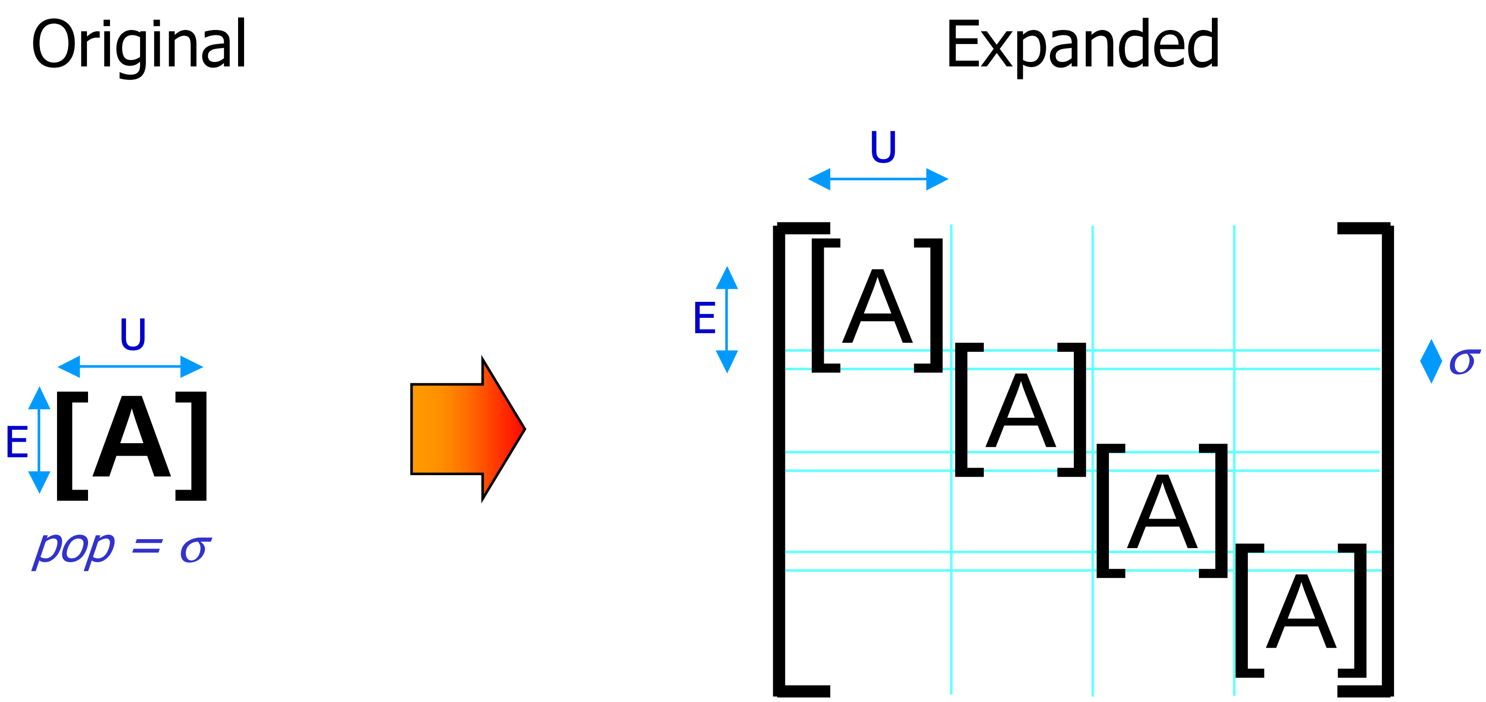


Combination Example



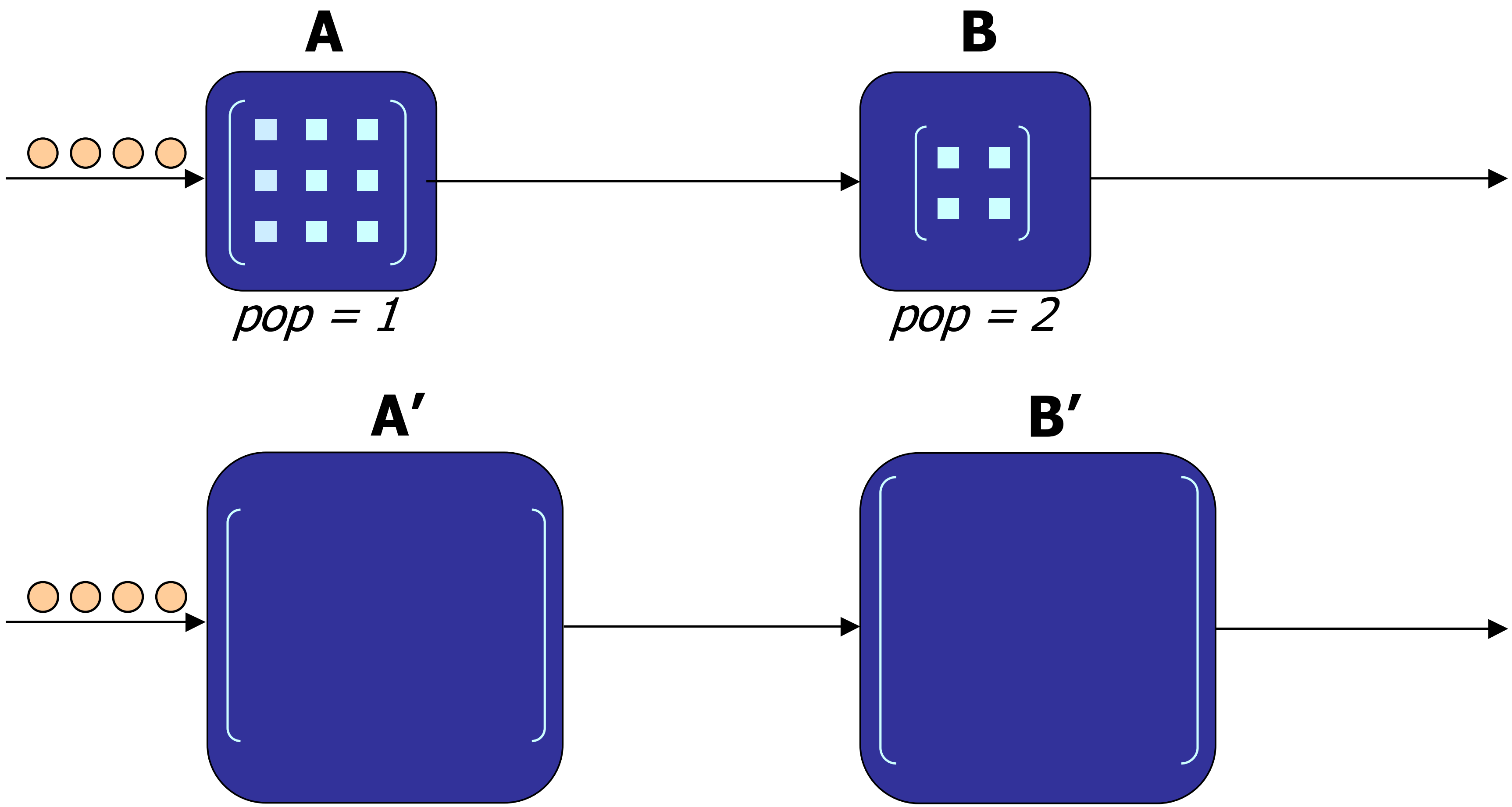
AB for any A and B??

- Linear Expansion



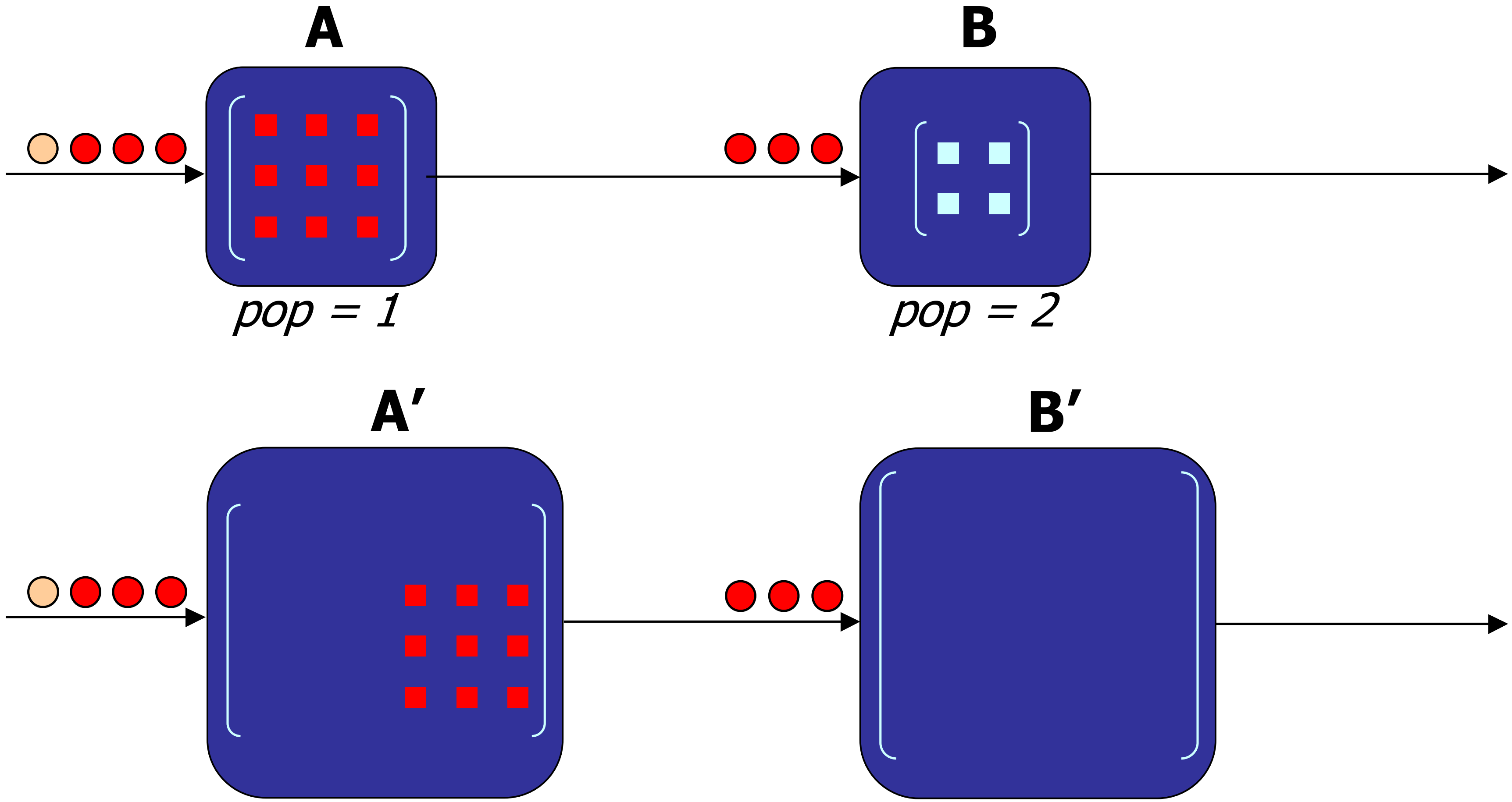
AB for any A and B??

Need to "expand" matrices to a steady-state cycle:



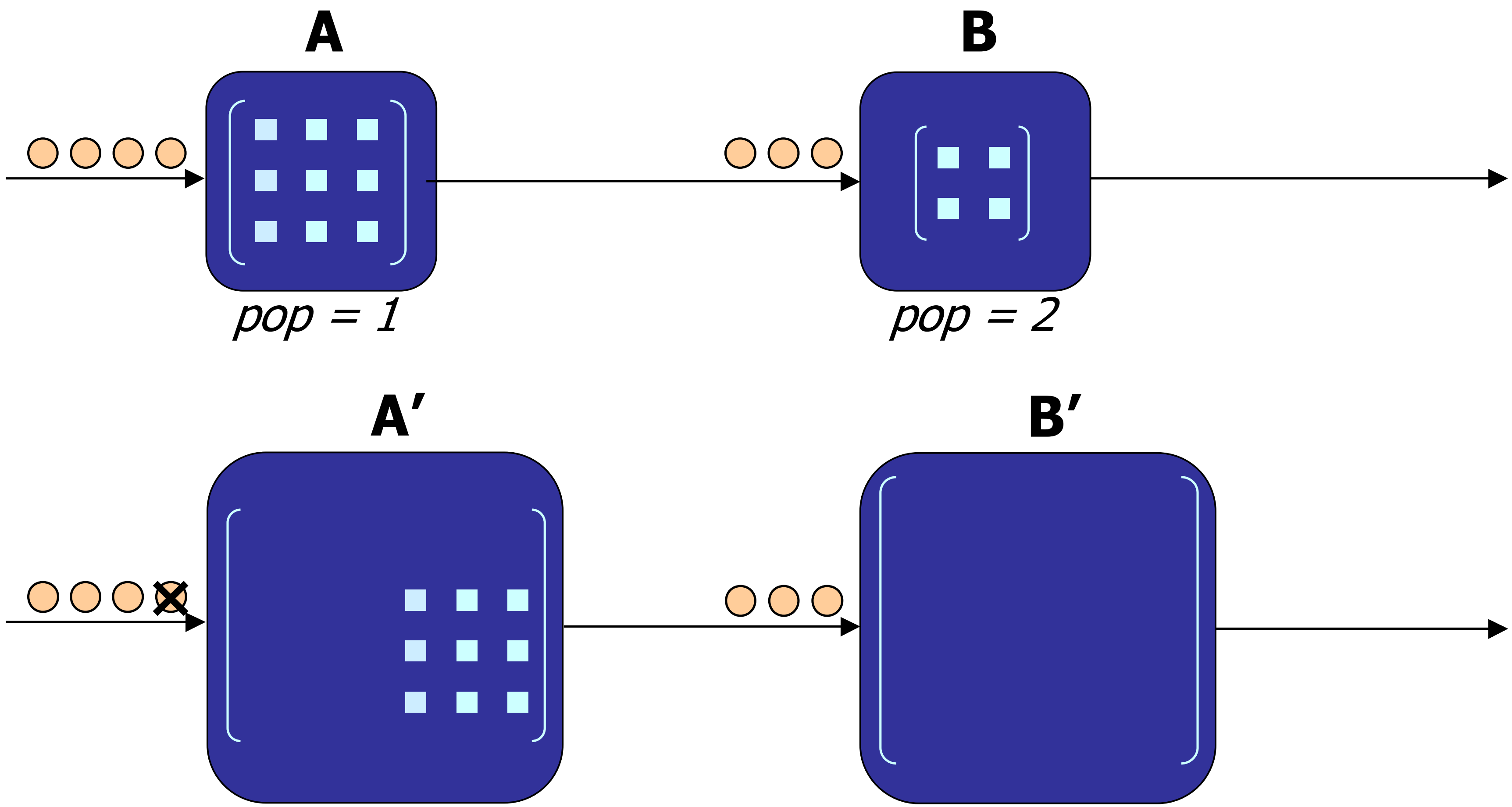
AB for any A and B??

Need to "expand" matrices to a steady-state cycle:



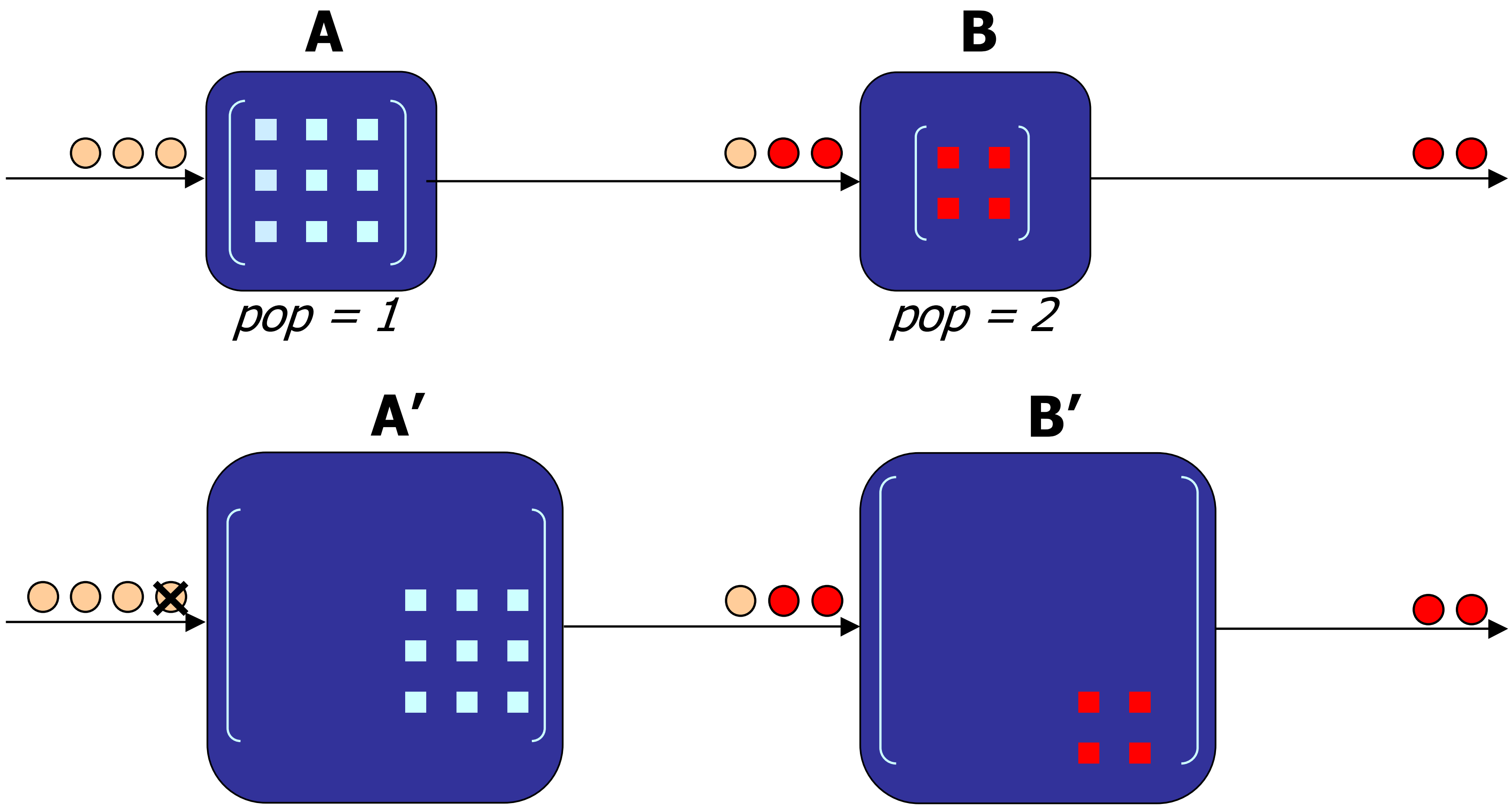
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:



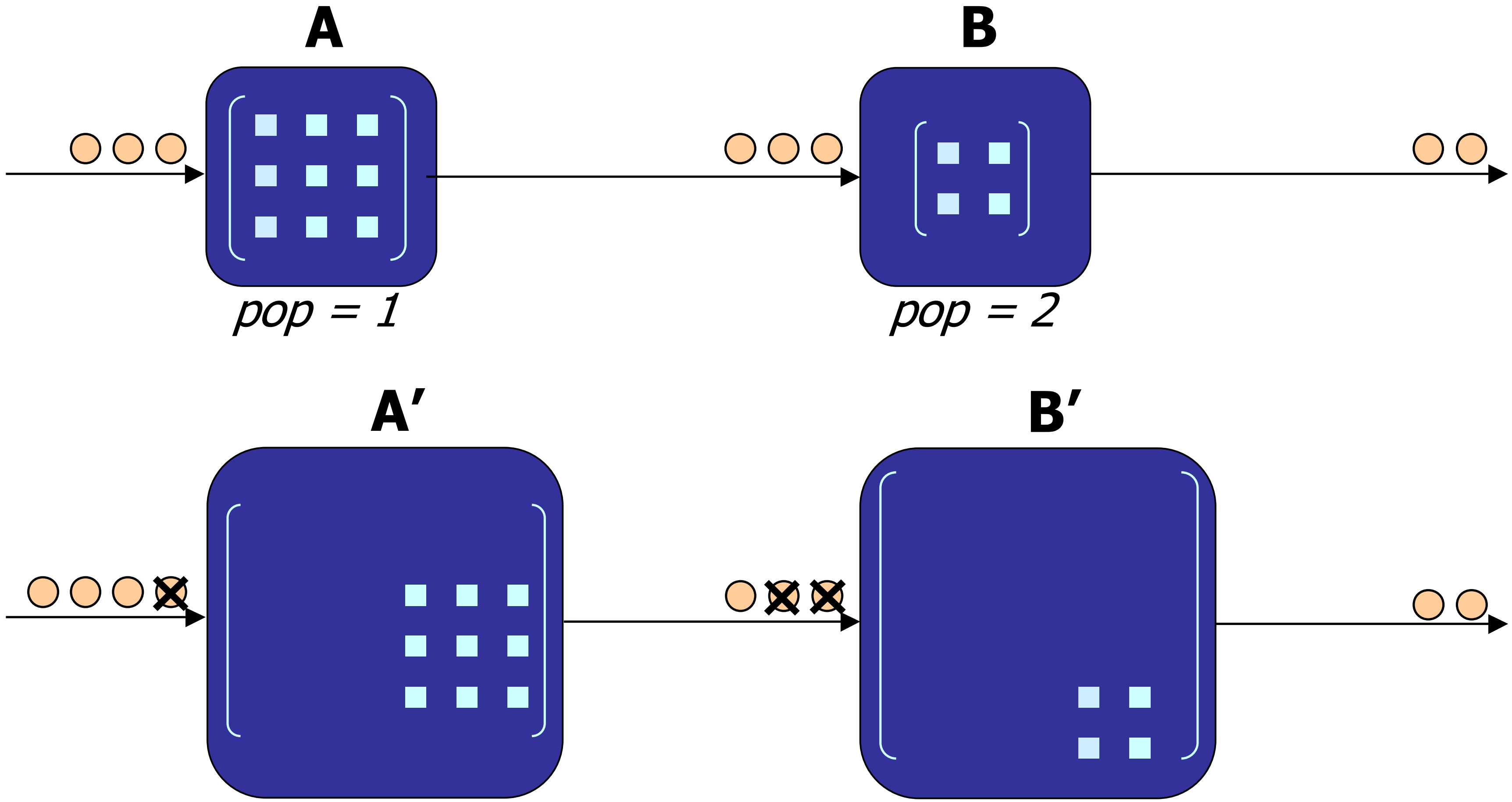
AB for any A and B??

Need to "expand" matrices to a steady-state cycle:



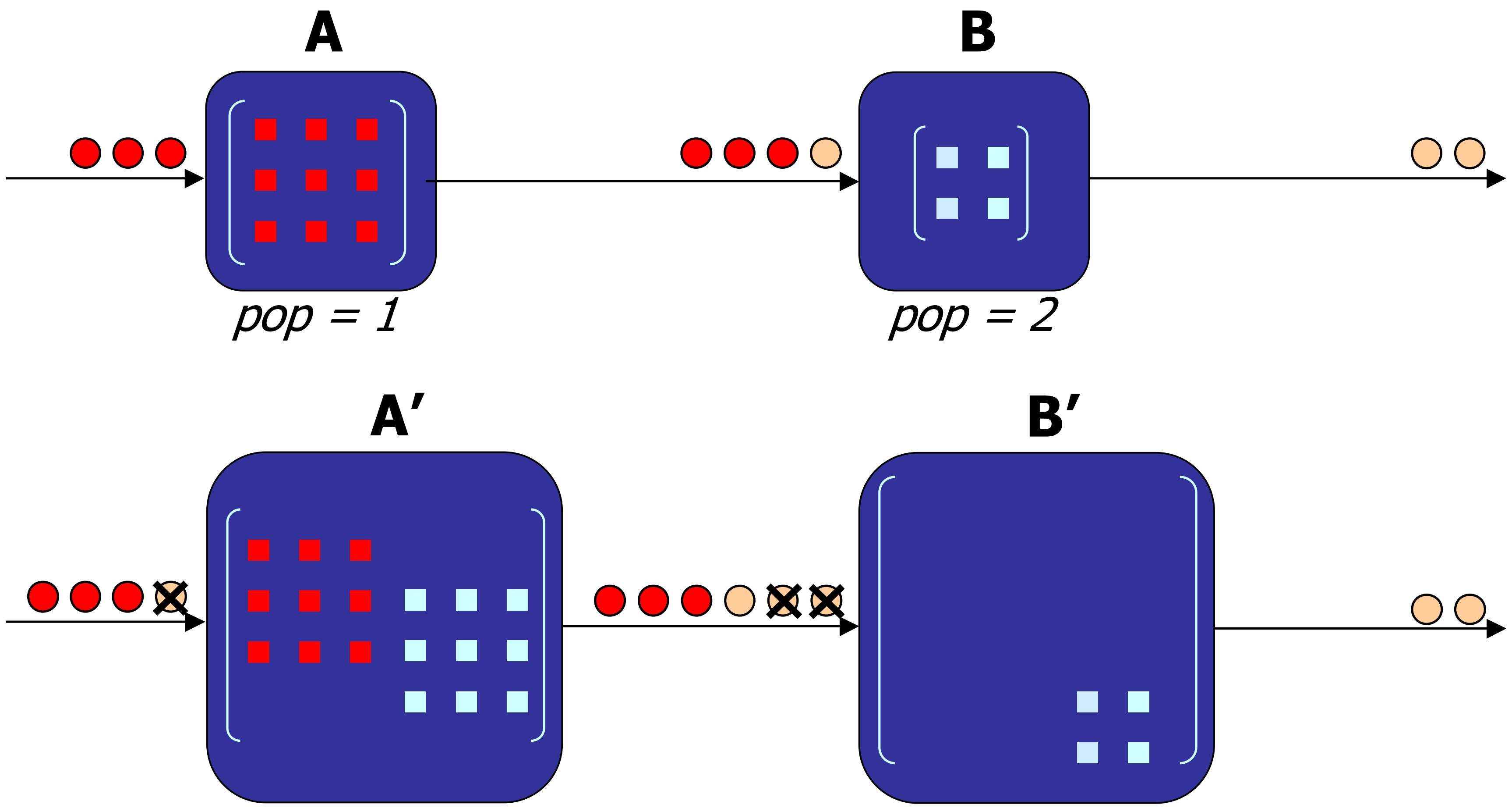
AB for any A and B??

Need to "expand" matrices to a steady-state cycle:



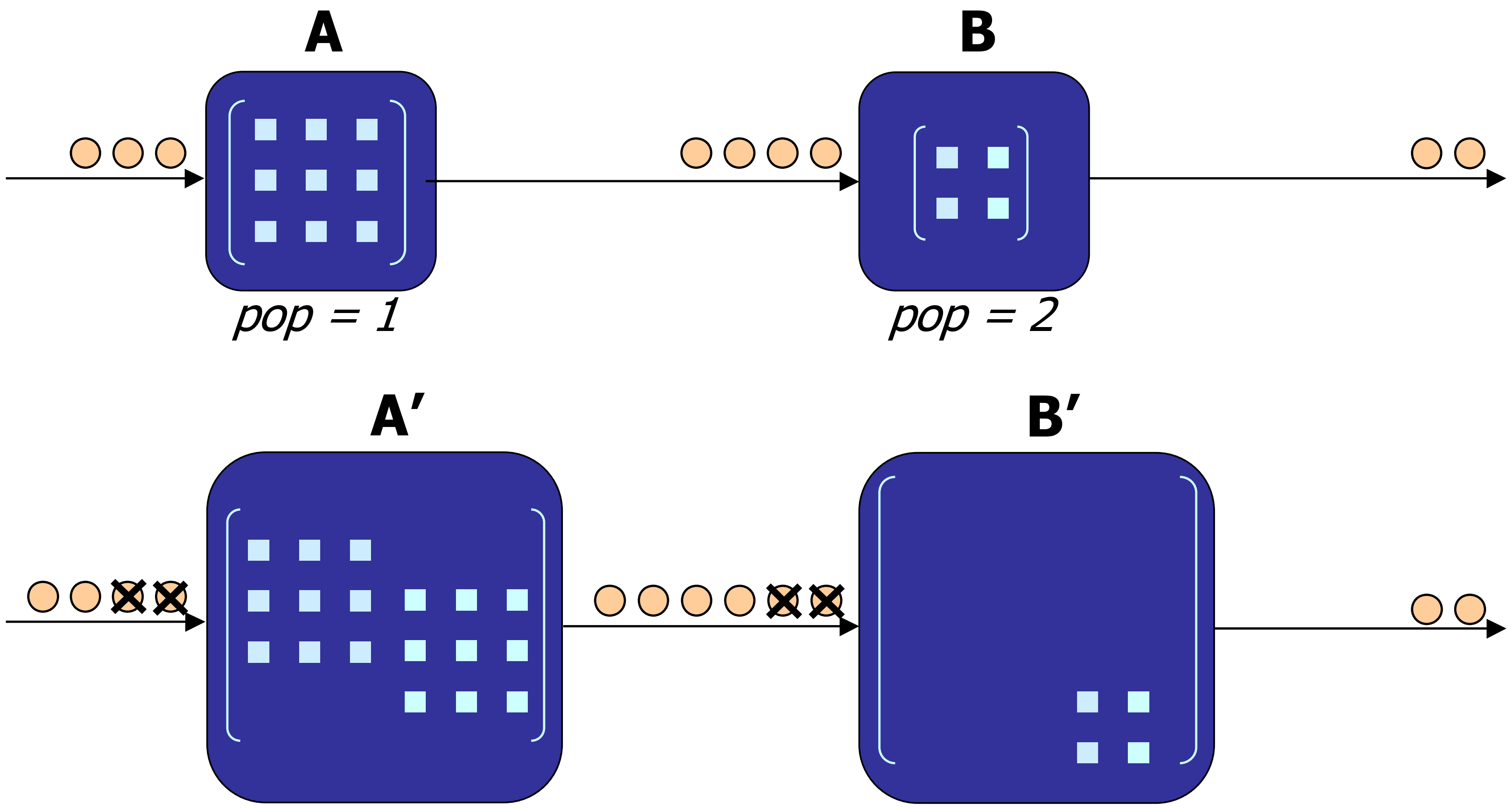
AB for any A and B??

Need to "expand" matrices to a steady-state cycle:



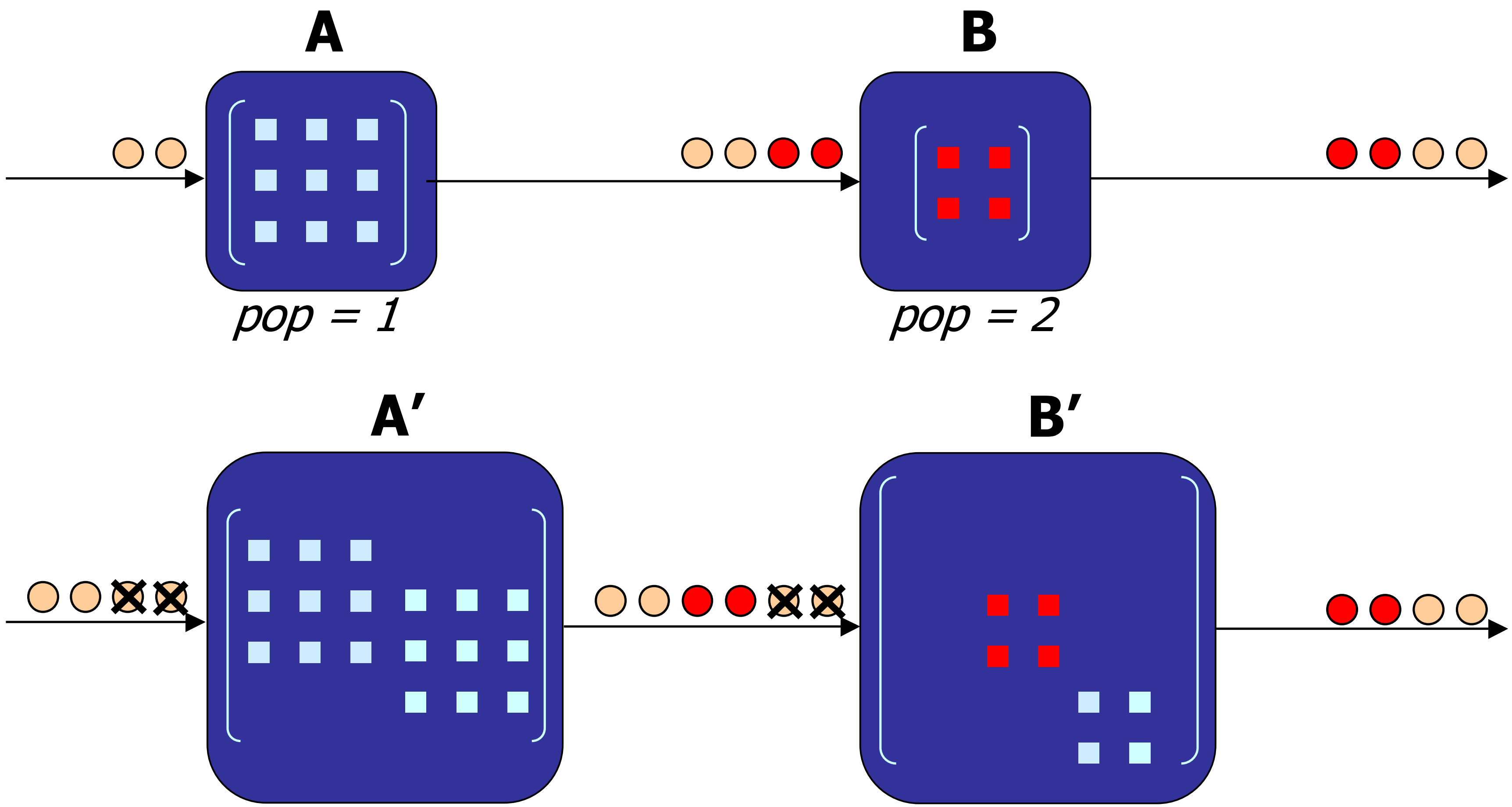
AB for any A and B??

Need to "expand" matrices to a steady-state cycle:



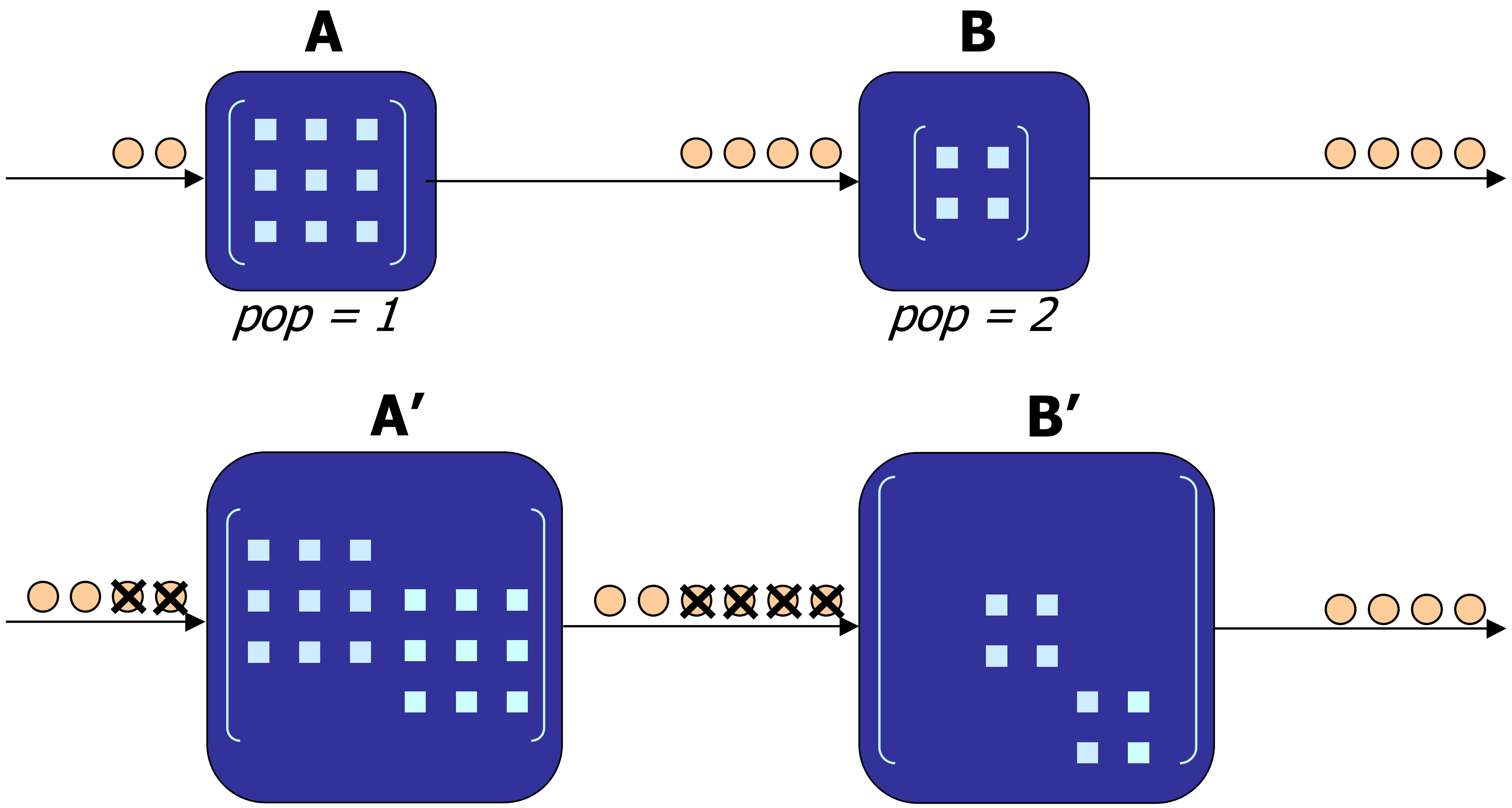
AB for any A and B??

Need to "expand" matrices to a steady-state cycle:



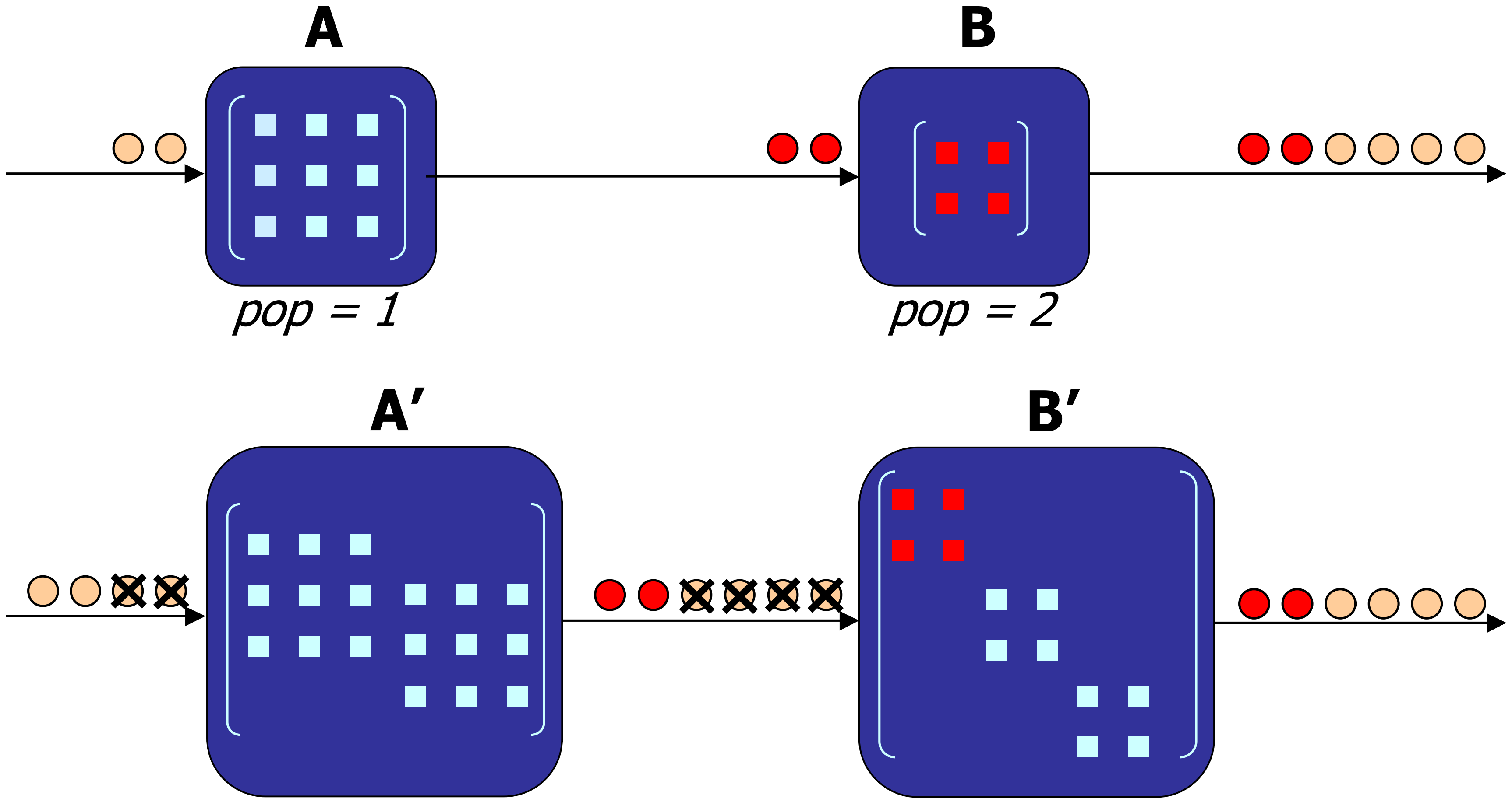
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:



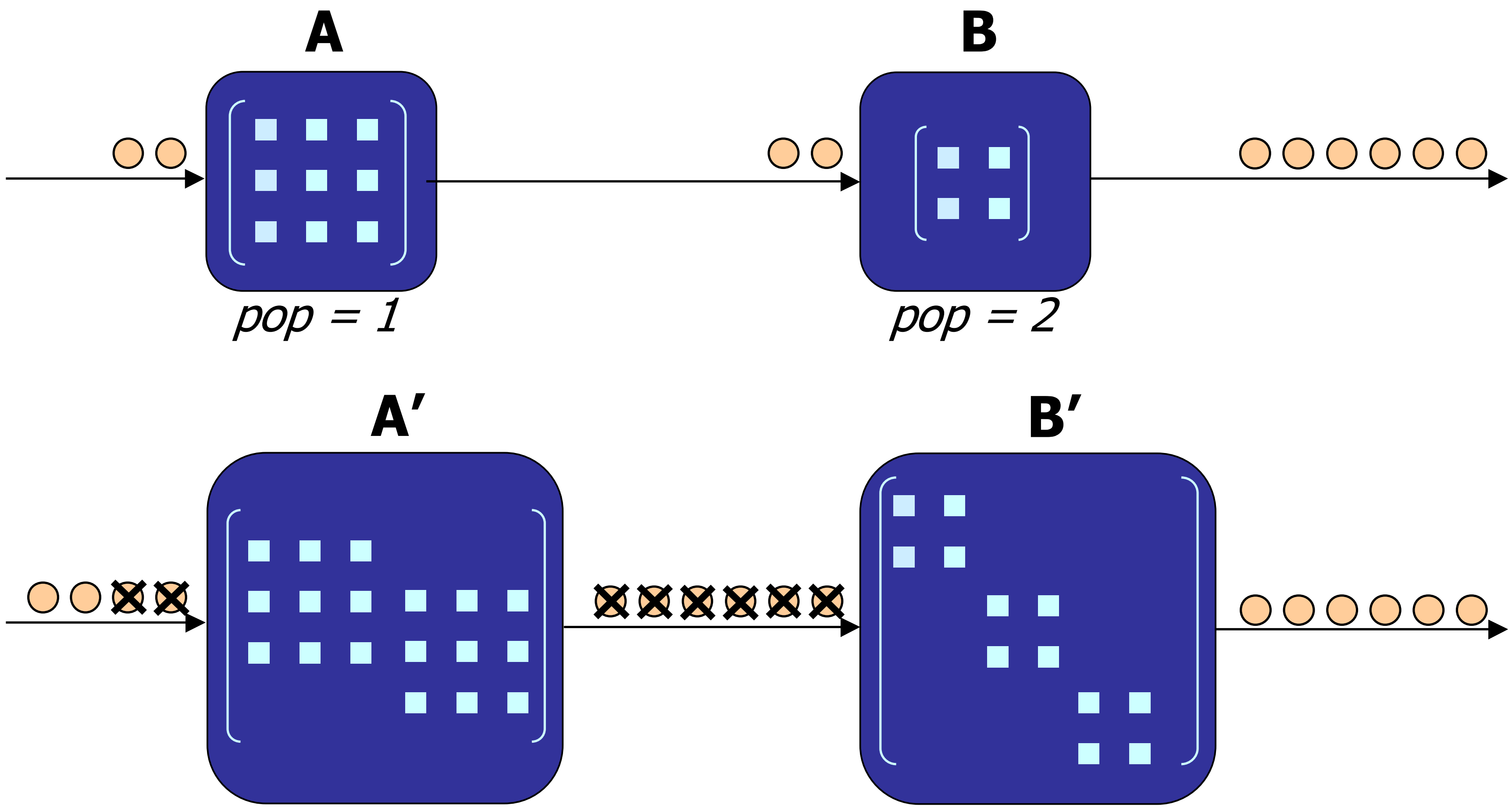
AB for any A and B??

Need to "expand" matrices to a steady-state cycle:



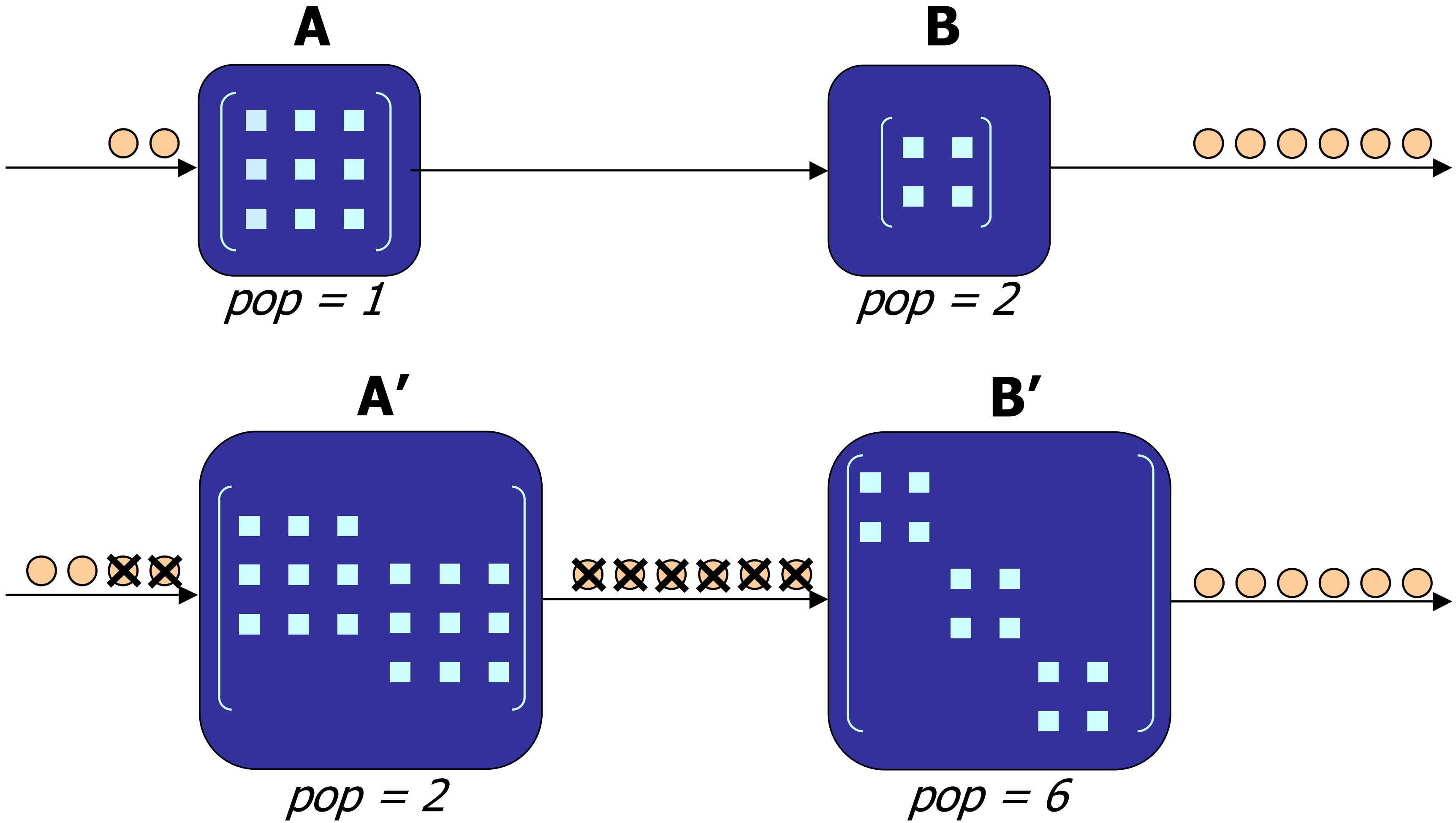
AB for any A and B??

Need to "expand" matrices to a steady-state cycle:



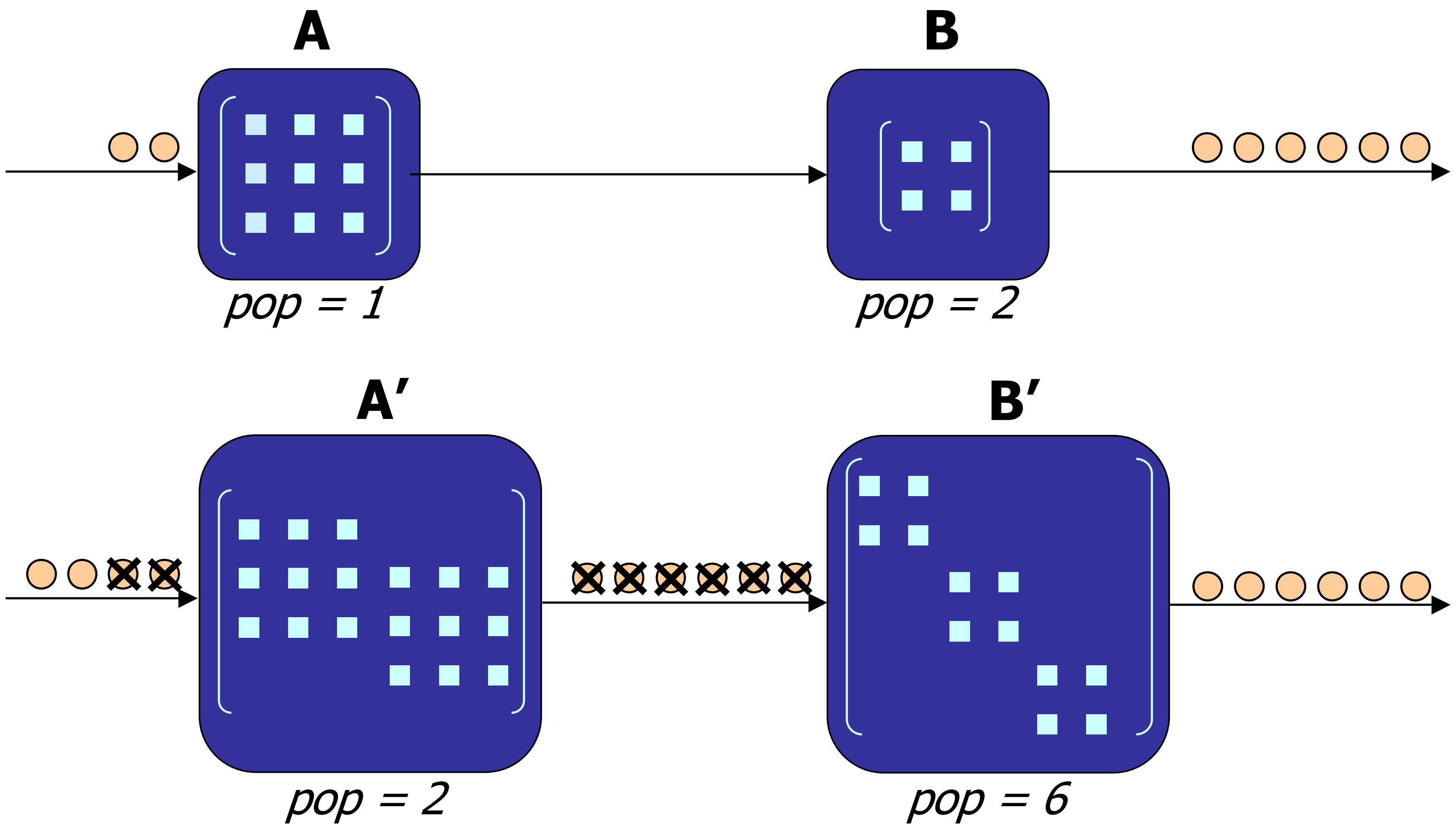
AB for any A and B??

Need to “expand” matrices to a steady-state cycle:

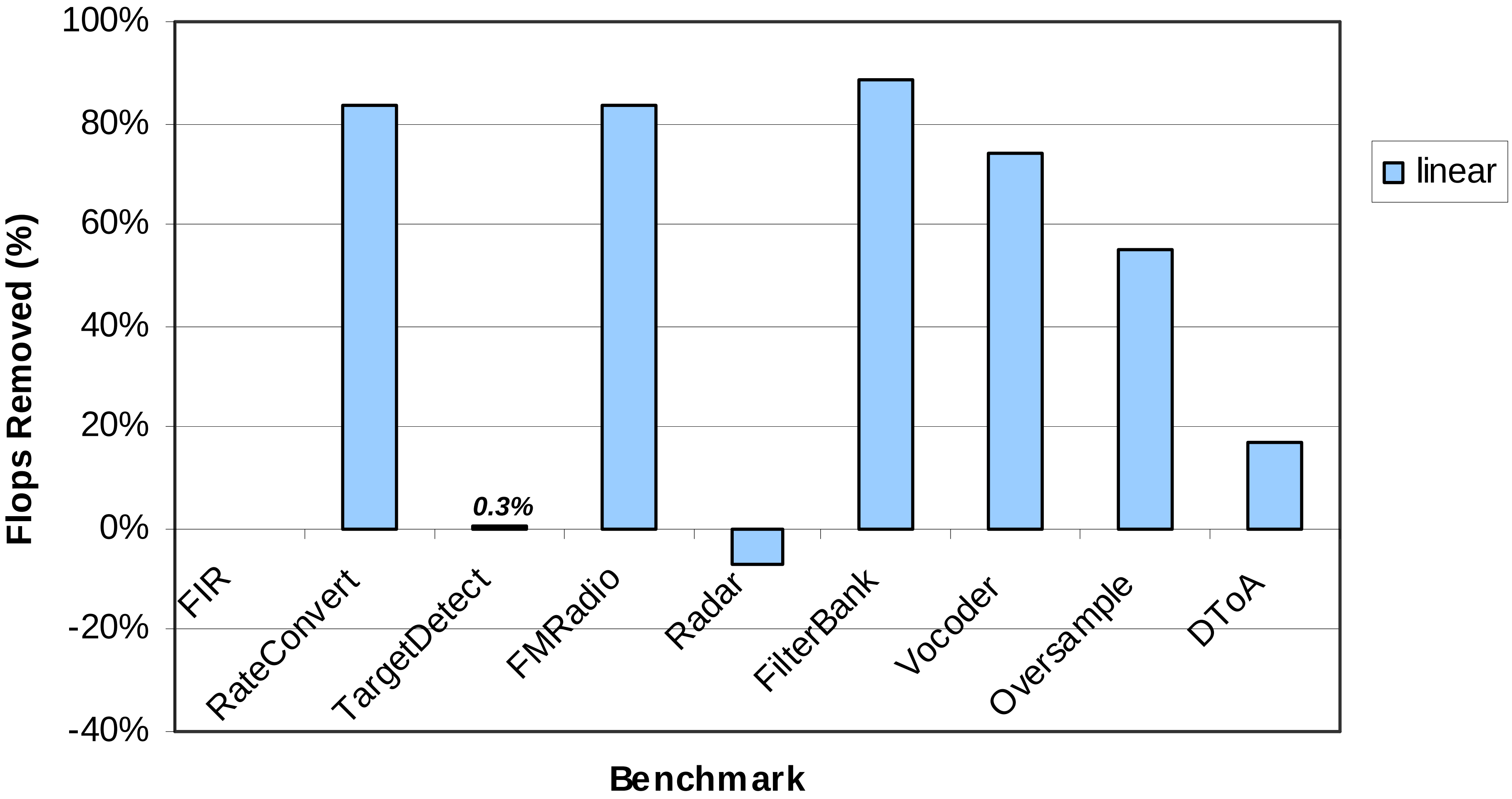


AB for any A and B??

Expanded dimensions match

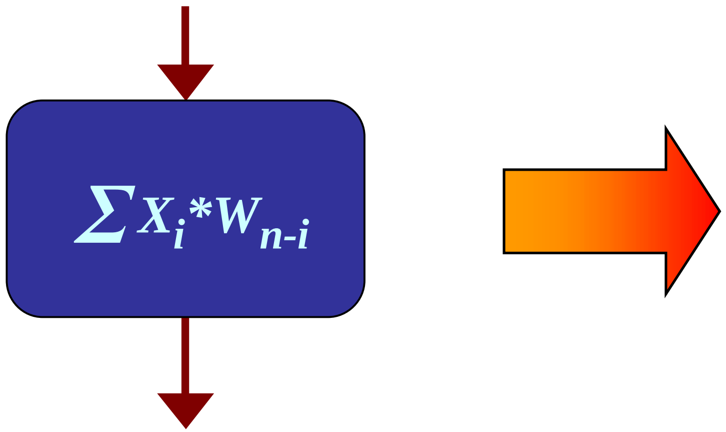


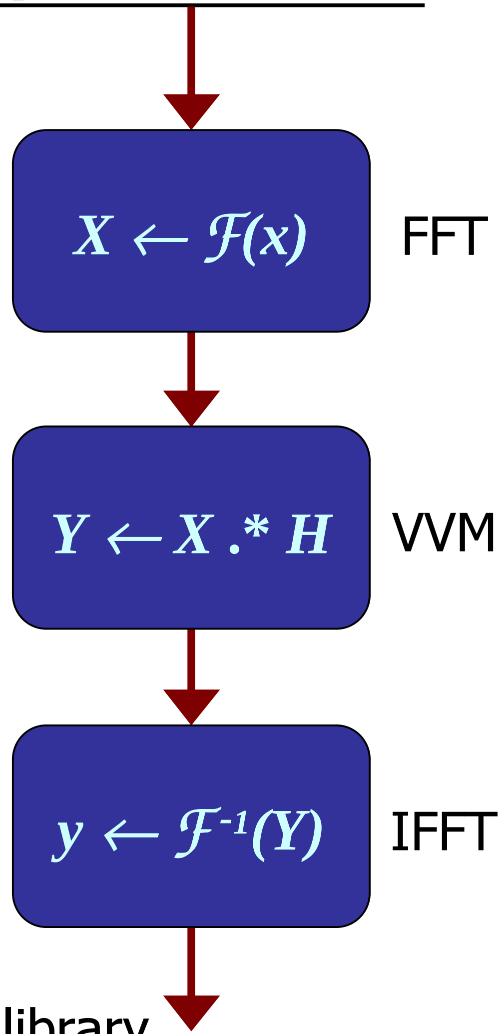
Floating-Point Operations Reduction



2) From Time to Frequency Domain

- Convolutions can be done cheaply in the Frequency Domain





- Painful to do by hand
  - Blocking
  - Coefficient calculations
  - Startup
  - Multiple outputs
  - Interfacing with FFT library

Generic Freq. Implementation

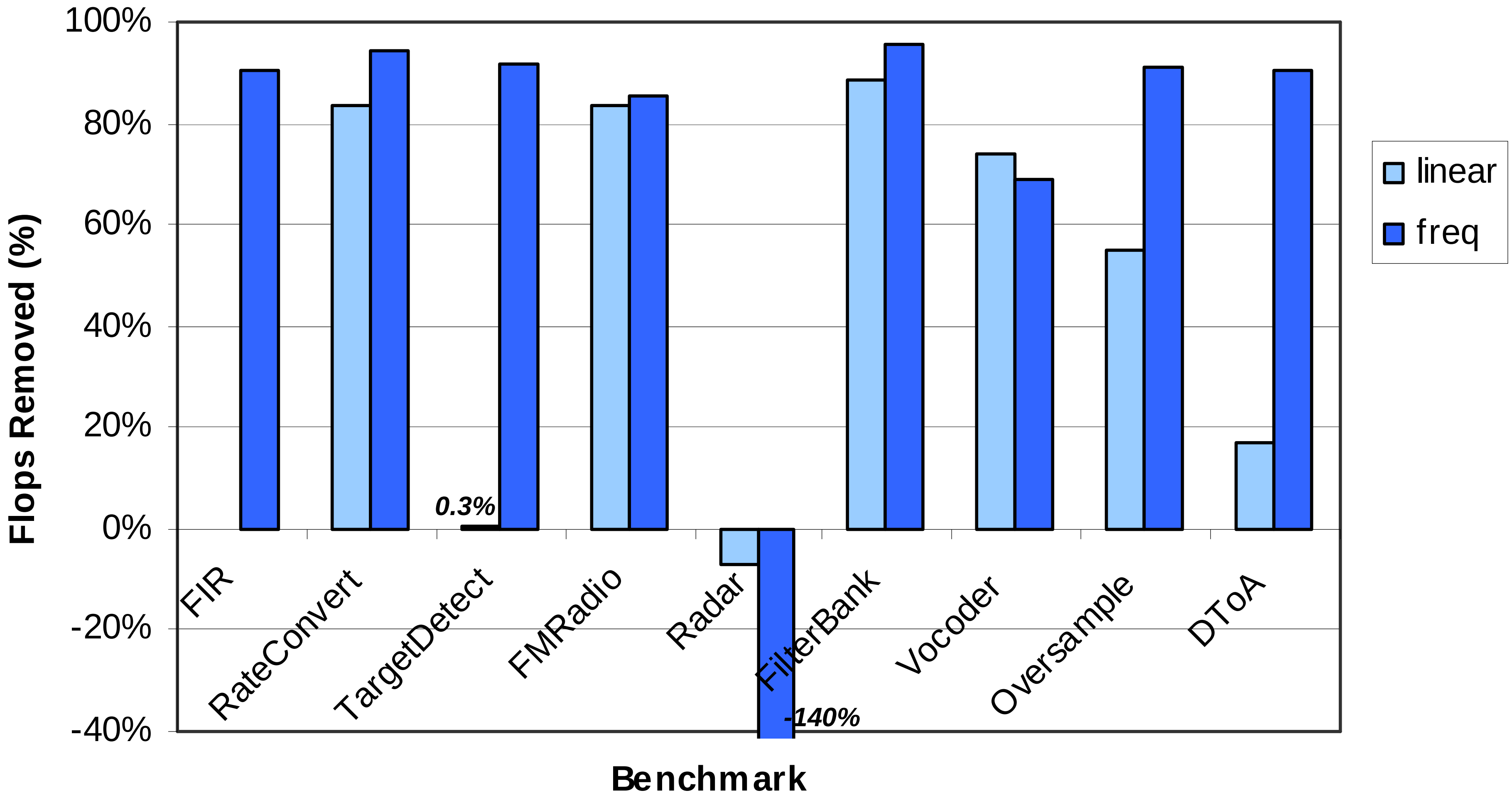
```
float → float pipeline optimizedFreq (A, b⃗, e, o, u) {
  add float → float filter {
    N ← 2^⌈lg(2e)⌉
    m ← N − 2e + 1
    partials⃗ ← new array[0 ... e − 2, 0 ... u − 1]
    r ← m + e − 1

    init {
      for j = 0 to u − 1
        H⃗[*, j] ← FFT(N, A[*, u − 1 − j])
    }

    prework peek r pop r push u * m {
      x⃗ ← pop(0 ... m + e − 2)
      X⃗ ← FFT(N, x⃗)
      for j = 0 to u − 1 {
        Y⃗[*, j] ← X⃗ .* H⃗[*, j]
        y⃗[*, j] ← IFFT(N, Y⃗[*, j])
        partials⃗[*, j] ← y⃗[m + e − 1 ... m + 2e − 3, j]
      }
      for i = 0 to m − 1
        for j = 0 to u − 1
          push(y⃗[i + e − 1, j] + b⃗[j])
    }

    work peek r pop r push u * r {
      x⃗ ← pop(0 ... m + e − 2)
      X⃗ ← FFT(N, x⃗)
      for j = 0 to u − 1 {
        Y⃗[*, j] ← X⃗ .* H⃗[*, j]
        y⃗[*, j] ← IFFT(N, Y⃗[*, j])
      }
      for i = 0 to e − 1
        for j = 0 to u − 1 {
          push(y⃗[i, j] + partials⃗[i, j])
          partials⃗[i, j] ← y⃗[m + e − 1 + i, j]
        }
      for i = 0 to m − 1
        for j = 0 to u − 1
          push(y⃗[i + e − 1, j] + b⃗[j])
    }
  }
  add Decimator(o, u)
}
```

Floating-Point Operations Reduction



3) Transformation Selection

- When to apply what transformations?
  - Linear filter combination can increase the computation cost
  - Shifting to the Frequency domain is expensive for filters with pop > 1
    - Compute all outputs, then decimate by pop rate
  - Some expensive transformations may later enable other transformations, reducing the overall cost

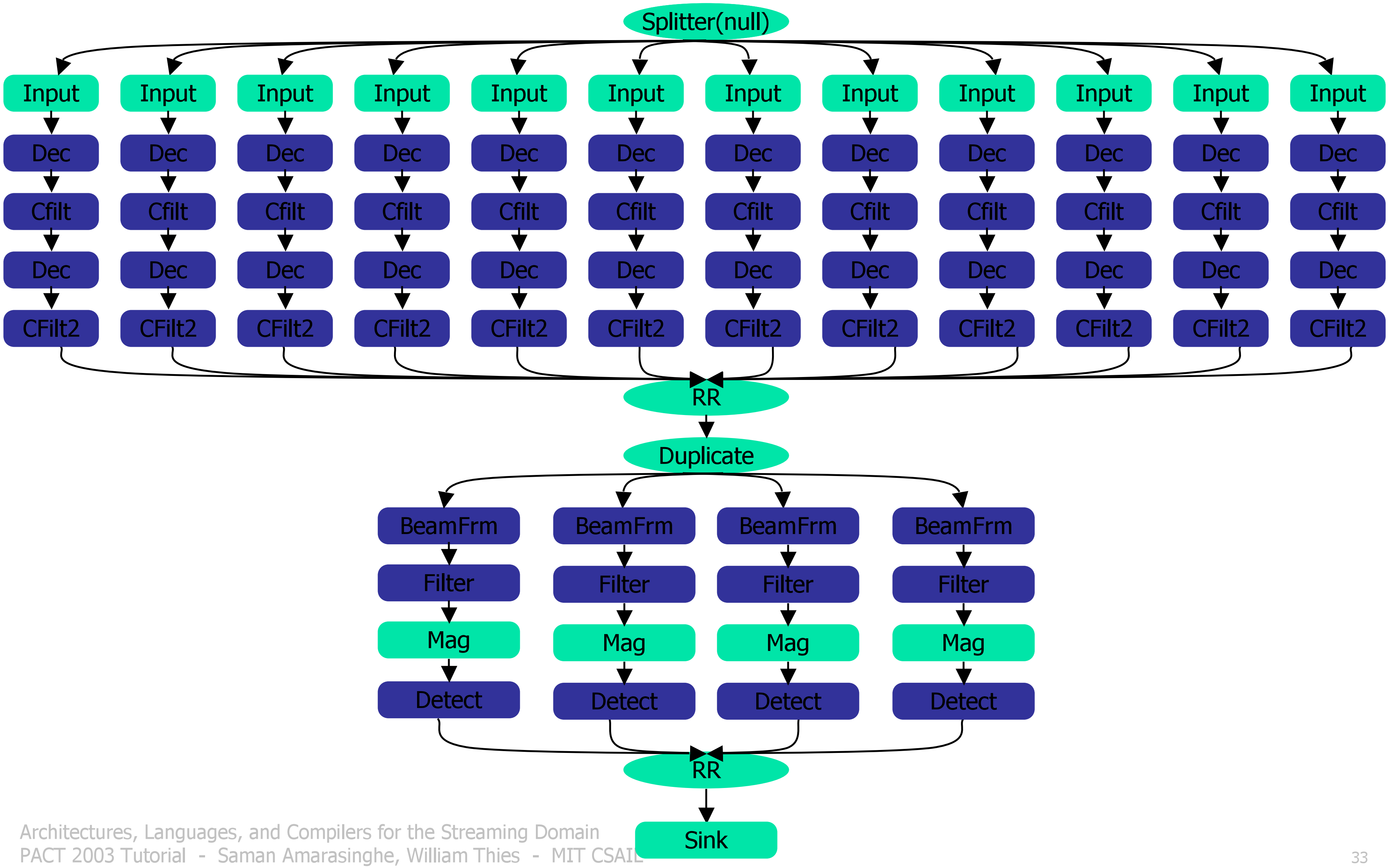
Selection Algorithm

- Estimate minimal cost for each structure:
  - Linear combination
  - Frequency translation
  - No transformation
    - If hierarchical, consider all possible groupings of children

Cost function based on profiler feedback

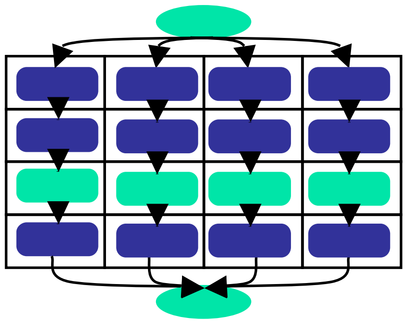
- Overlapping sub-problems allows efficient dynamic programming search

Radar (Transformation Selection)



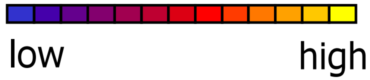
Radar (Transformation Selection)

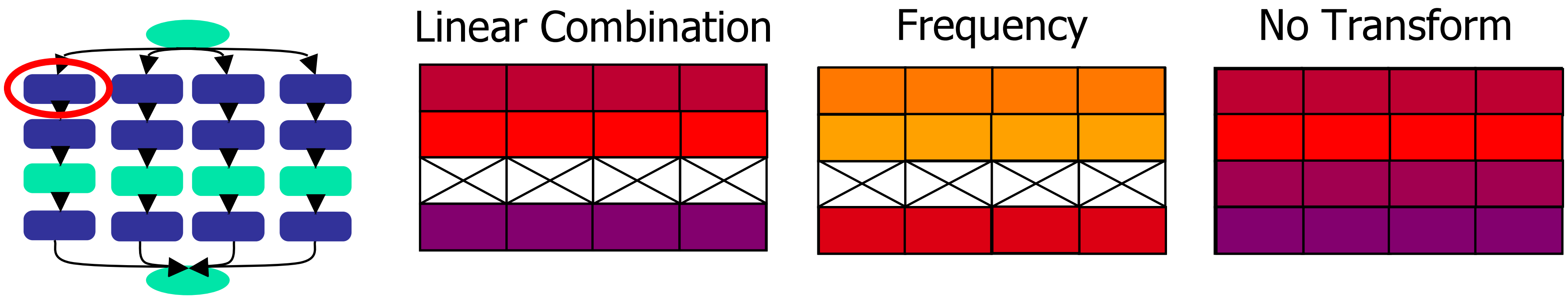
First compute cost of individual filters:



Radar (Transformation Selection)

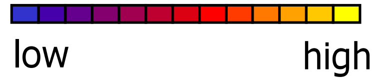
First compute cost of individual filters:

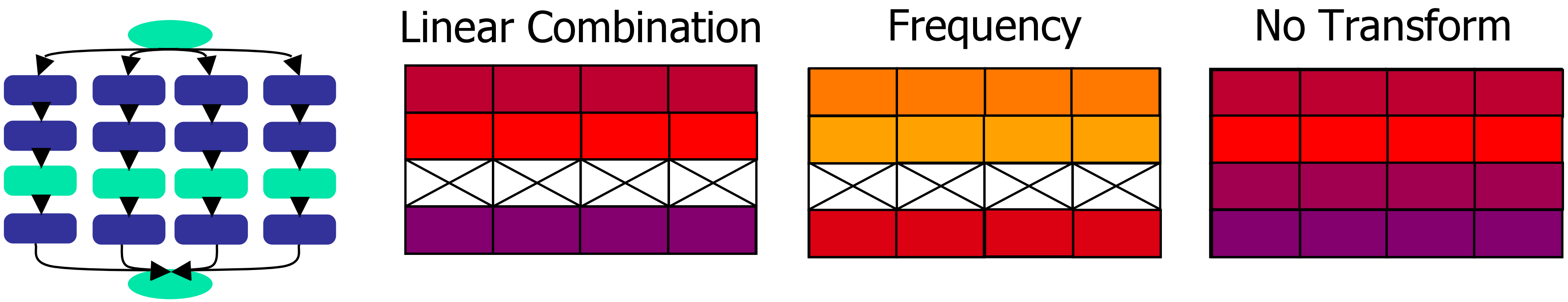




Radar (Transformation Selection)

First compute cost of individual filters:

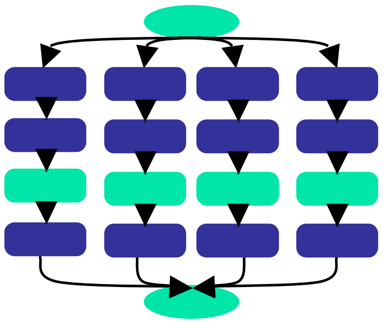




1x1

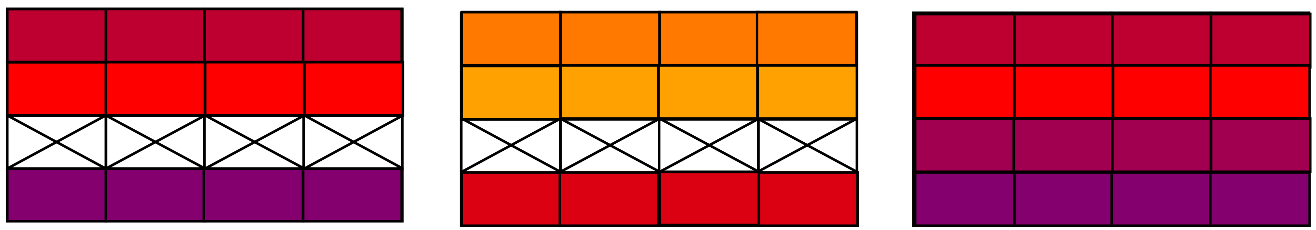
Radar (Transformation Selection)

Then, compute cost of 1x2 nodes:



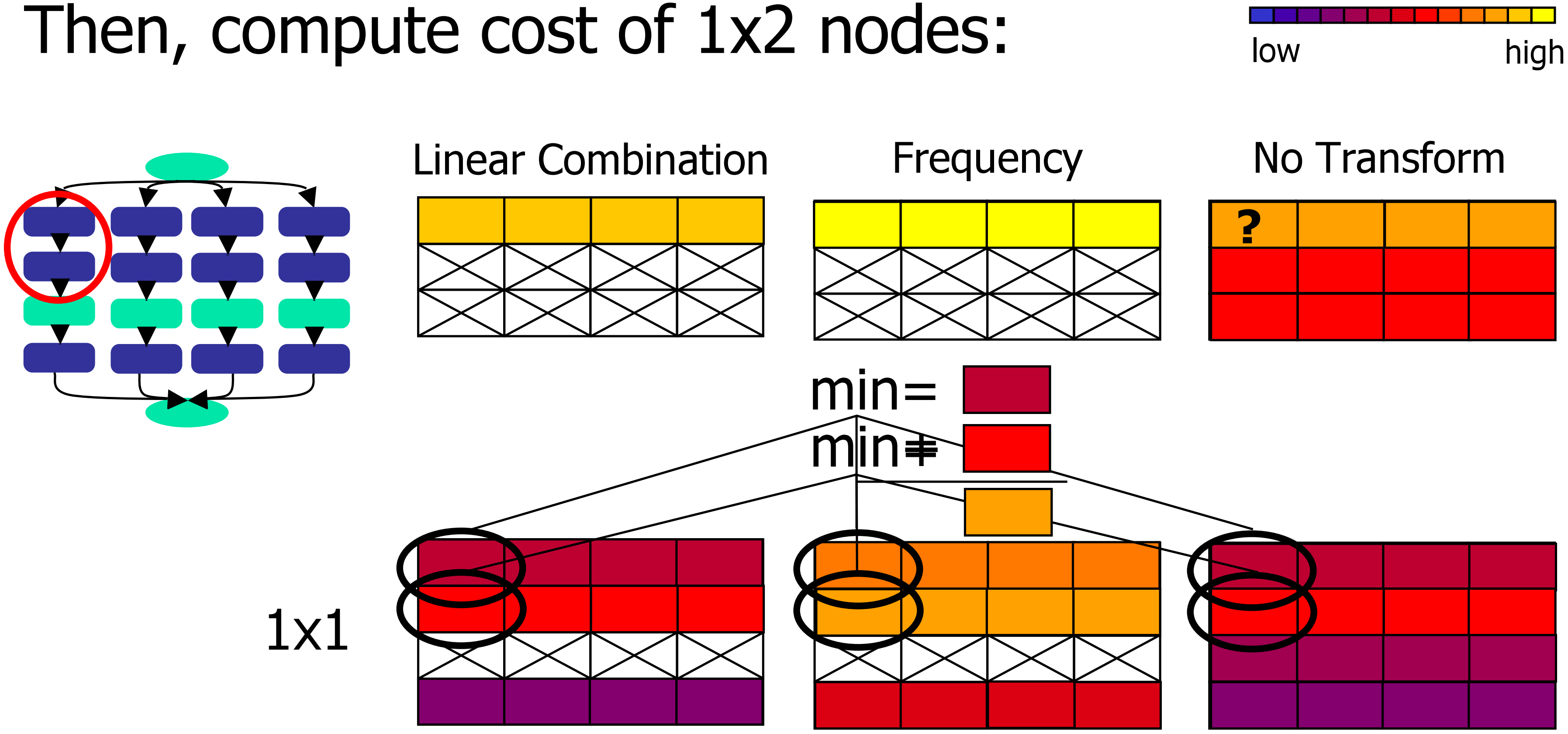
Linear Combination Frequency No Transform

1x1



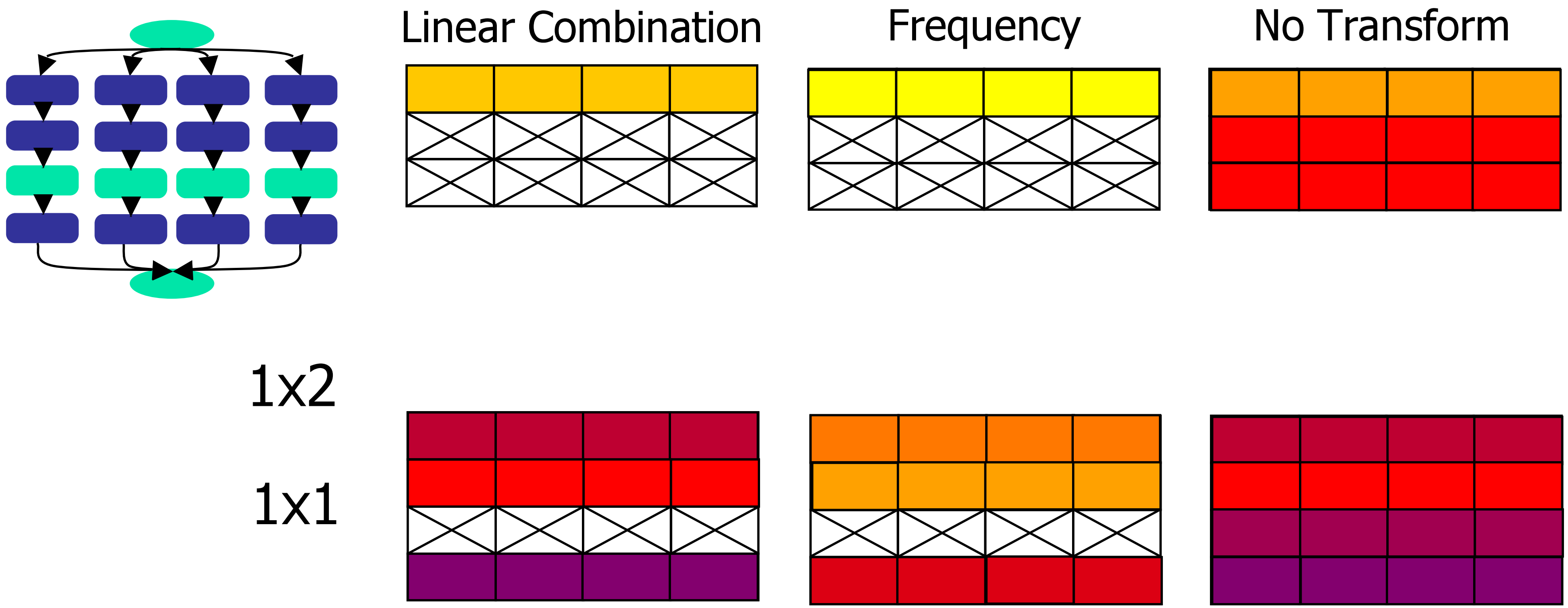
Radar (Transformation Selection)

Then, compute cost of 1x2 nodes:



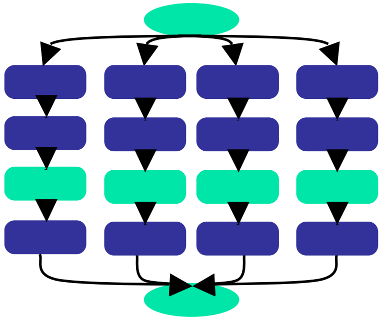
Radar (Transformation Selection)

Then, compute cost of 1x2 nodes:

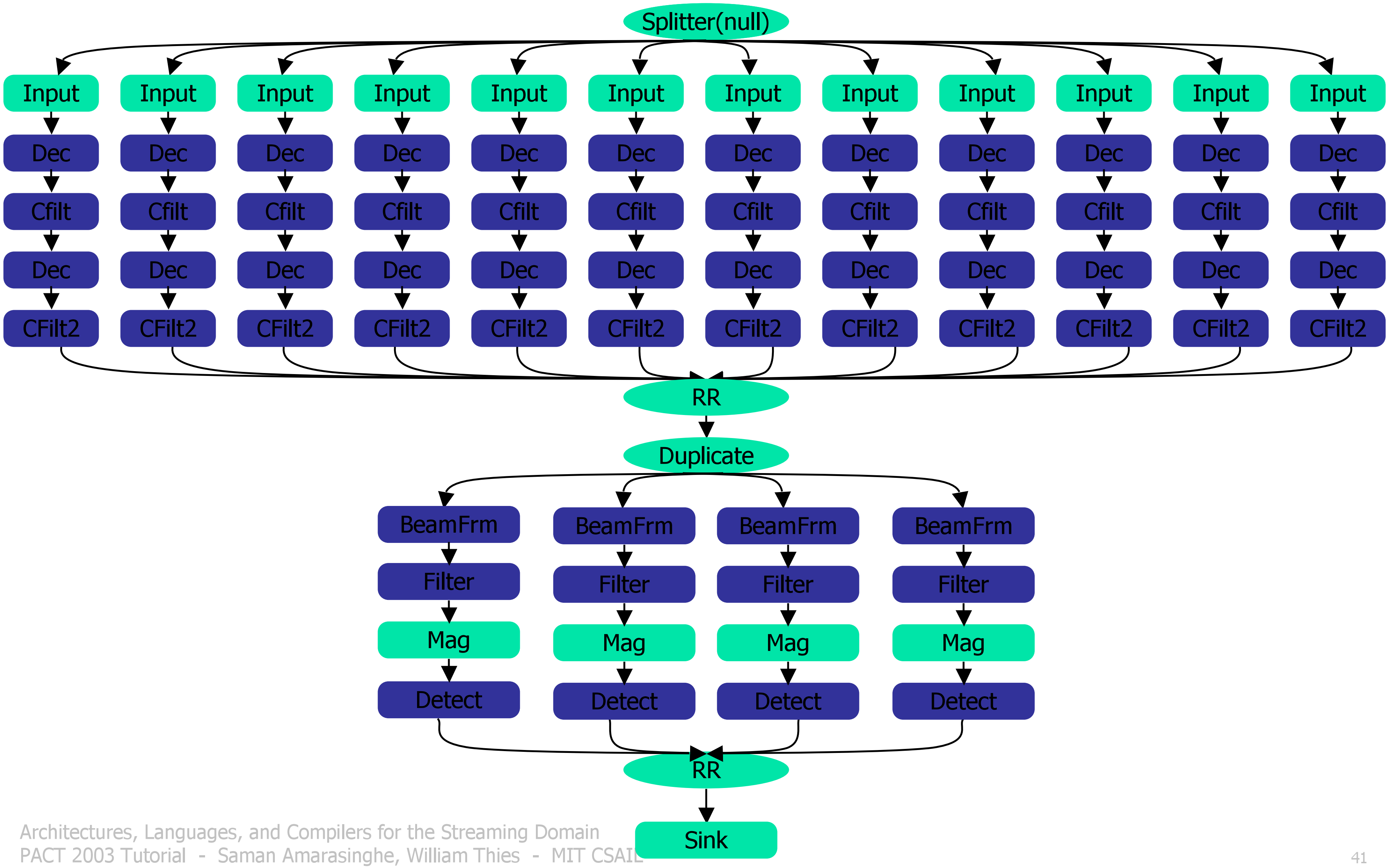


Radar (Transformation Selection)

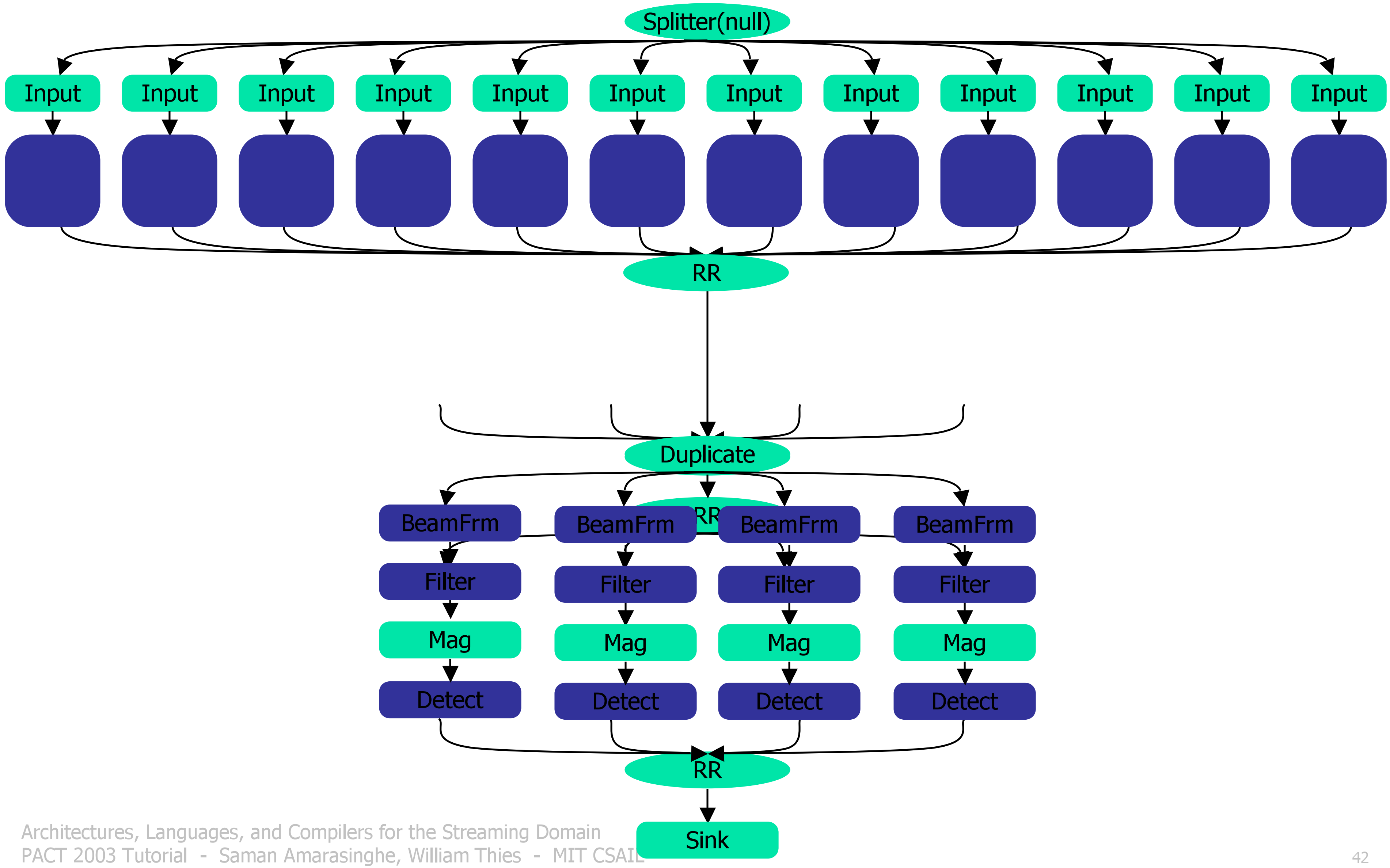
Continue with	1x3	2x1	3x1	4x1
	1x4	2x2	3x2	4x2
		2x3	3x3	4x3
		2x4	3x4	4x4 ⇨ Overall solution



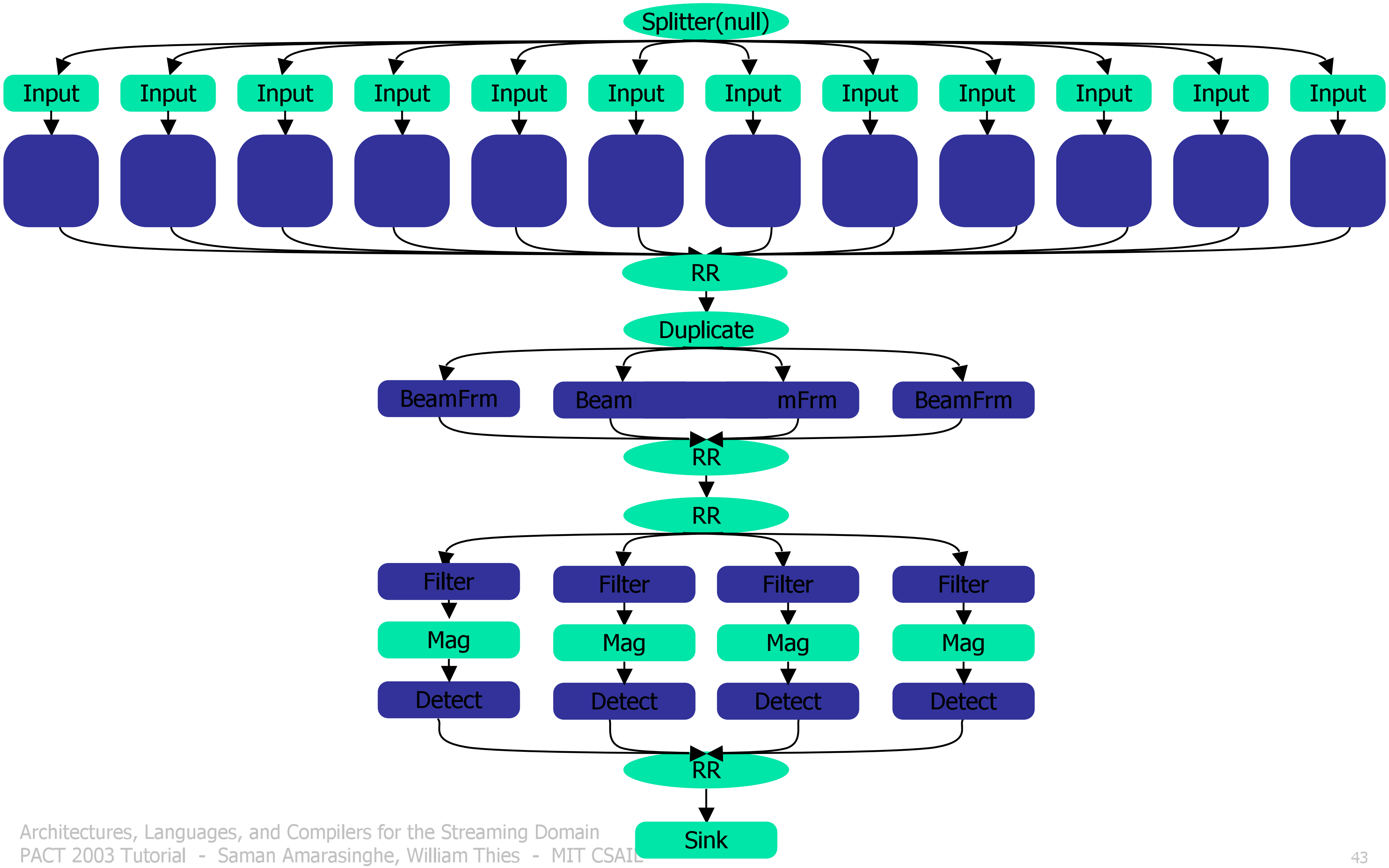
Radar (Transformation Selection)



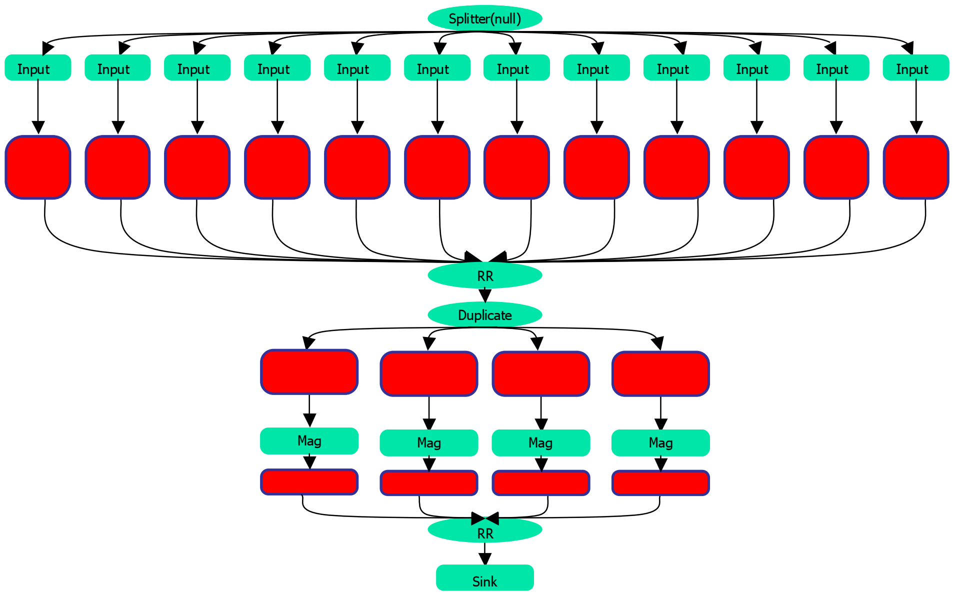
Radar (Transformation Selection)



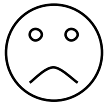
Radar (Transformation Selection)



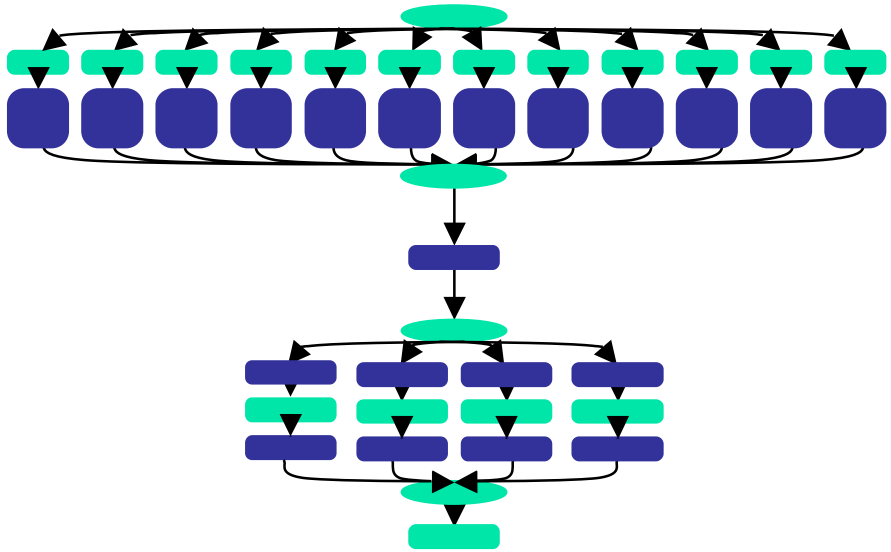
Radar



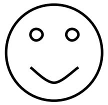
Maximal Combination and Shifting to Frequency Domain



2.4 times as many FLOPS

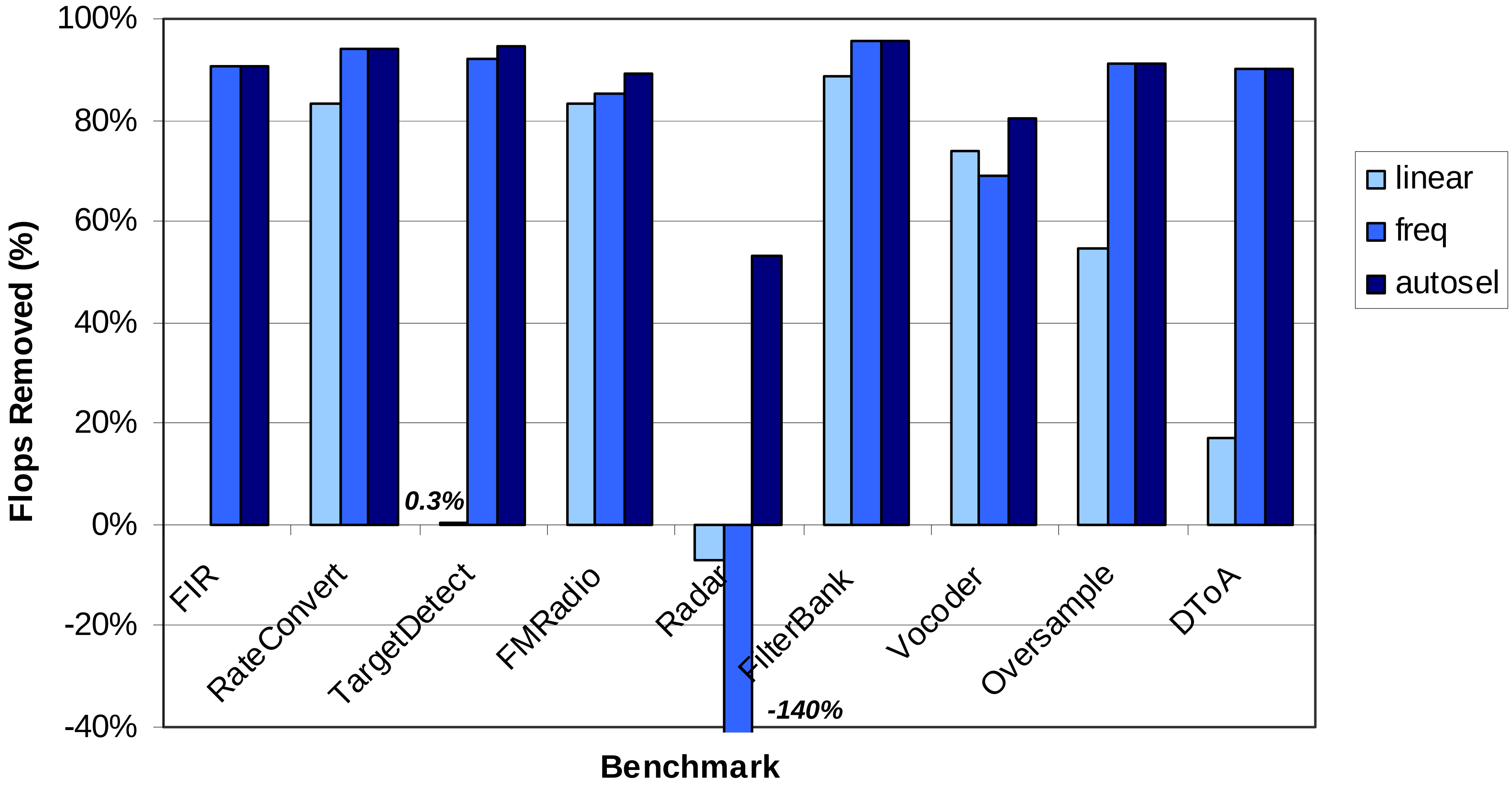


Using Transformation Selection



half as many FLOPS

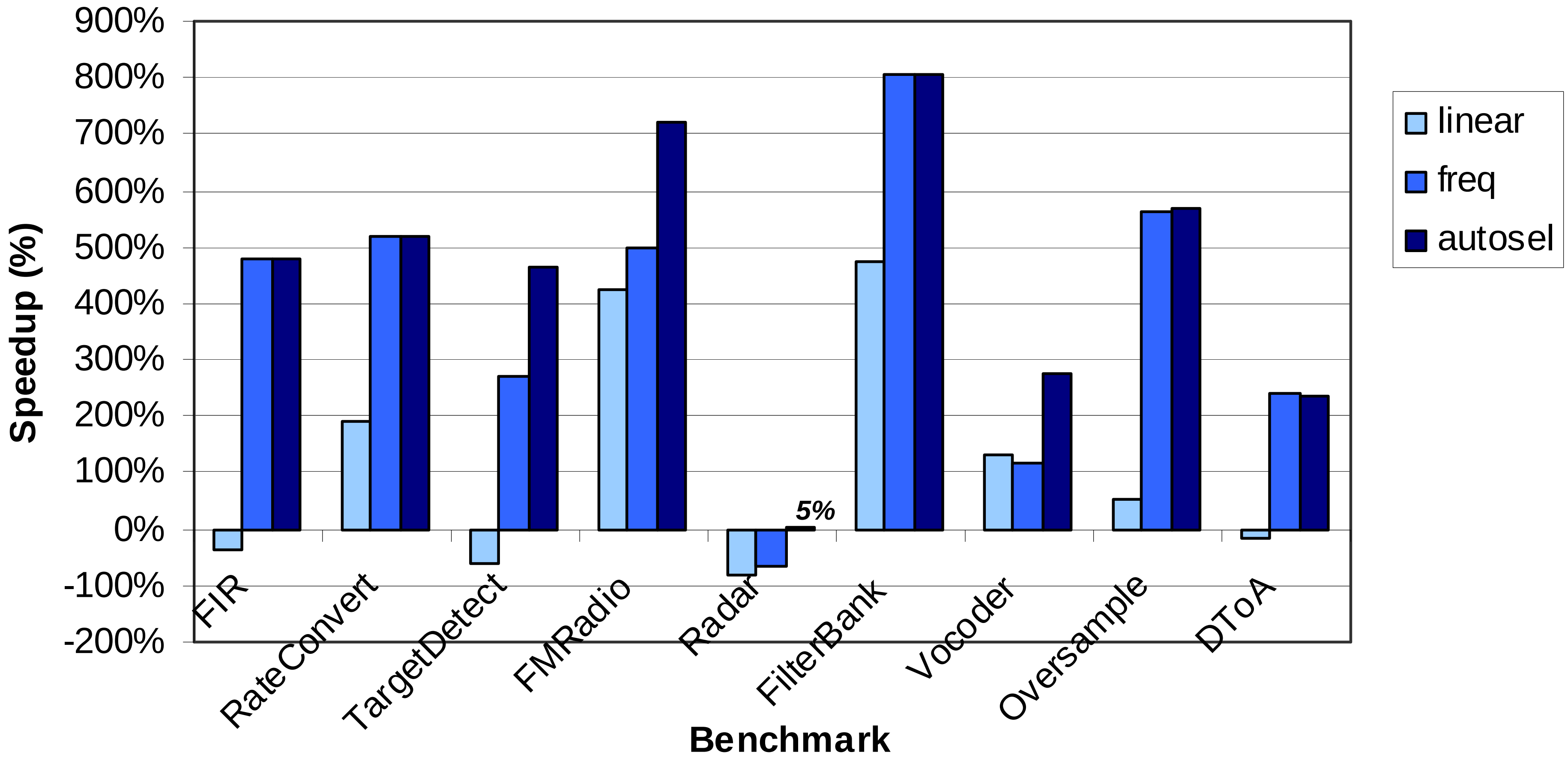
Floating-Point Operations Reduction



Experimental Results

- Fully automatic implementation
  - StreamIt compiler
- StreamIt to C compilation
  - FFTW for shifting to the frequency domain
- Benchmarks all written in StreamIt
- Measurements
  - Dynamic floating-point instruction counting
  - Speedups on a general purpose processor

Execution Speedup



On a Pentium IV

Related Work

- SPIRAL/SPL (Püschel et. al)
  - Automatic derivation of DSP transforms
- FFTW (Friego et. al)
  - Wicked fast FFT
- ADE (Covell, MIT PhD Thesis, 1989)
- Affine Analysis (Karr, Acta Informatica, 1976)
  - *Affine relationships among variables of a program*
- Linear Analysis (Cousot, Halbwatchs, POPL, 1978)
  - *Automatic discovery of linear restraints among variables of a program*

Conclusions

- A DSP Program Representation: *Linear Filters*
  - A dataflow analysis that recognizes linear filters

- Three Optimizations using Linear Information
  - Adjacent Linear Structure Combination
  - Time Domain to Frequency Domain Transformation
  - Automatic Transformation Selection

- First Step in Replacing the DSP Engineer from the Design Flow
  - On the average 90% of the FLOPs eliminated
  - Average performance speedup of 450%

- StreamIt: A Unified High-level Abstraction for DSP Programming
  - Increased abstraction does not have to sacrifice performance

http://cag.lcs.mit.edu/linear/