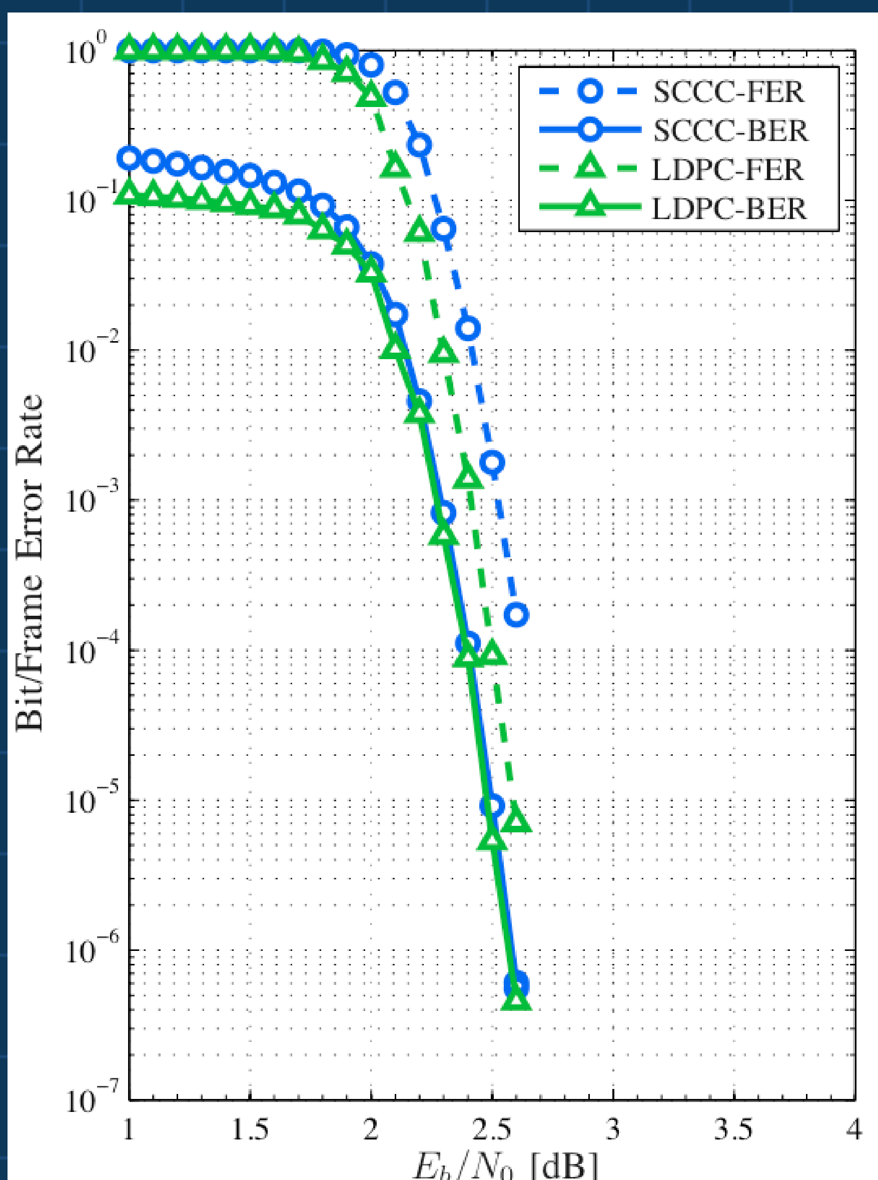
Generating Implementations of Error Correcting Codes using Kansas Lava

Andy Gill, Tristan Bull, Andrew Farmer, Garrin Kimmel, Ed Komp, Erik Perrins
University of Kansas

THE UNIVERSITY OF KANSAS

Information and Telecommunication Technology Center (ITTC)



Center collaboration based round focus areas or labs

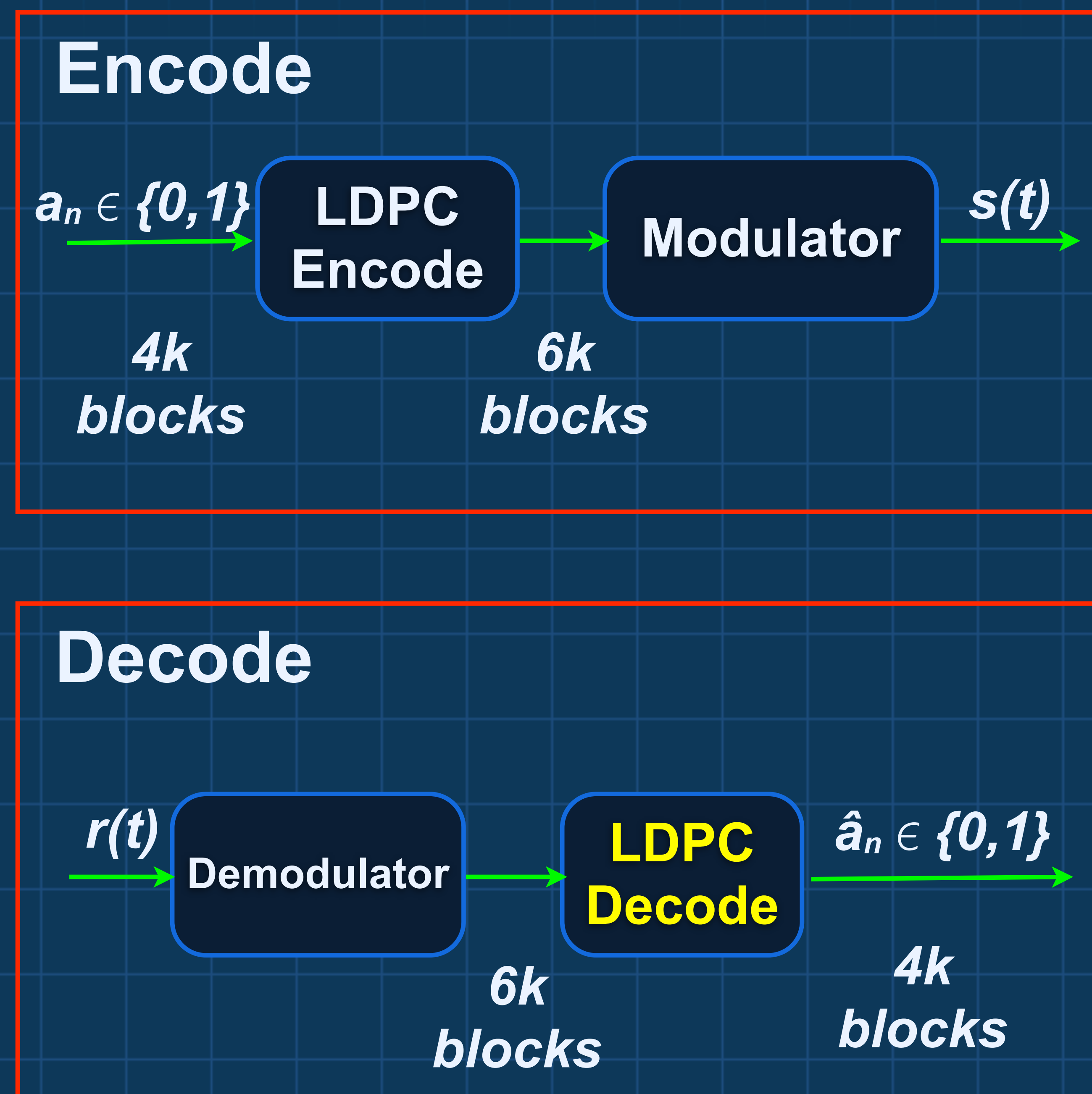
- Faculty are associated with one or more labs.
- Labs for Bioinformatics, Communications and Networks, Computer Systems, e-Learning, Intelligent Systems, Information Assurance, Radar and Remote Sensing.
- This project is a collaboration between three labs.

THE UNIVERSITY OF KANSAS

HFEC Project

- Forward Error Correction (FEC) codes are part of the migration path in future aeronautical telemetry standards for DoD/NASA test ranges
- Two candidate FEC codes have been selected
  - A serially concatenated convolutional code (SCCC) developed at KU
  - A low-density parity check (LDPC) code developed at NASA's Jet Propulsion Laboratory (JPL)
- Both codes have an information block size of 4096 bits and a rate of 2/3
- Hardware prototypes of these systems are needed as the next step in the evolution of the standard

Block Diagram of Prototype LDPC Implementation



THE UNIVERSITY OF KANSAS

HFEC Game Plan

We want to generate circuits for implementing LDPC!

- Interesting, practical problem.
- Based on well understood math.
- Real world constraints and requirements.

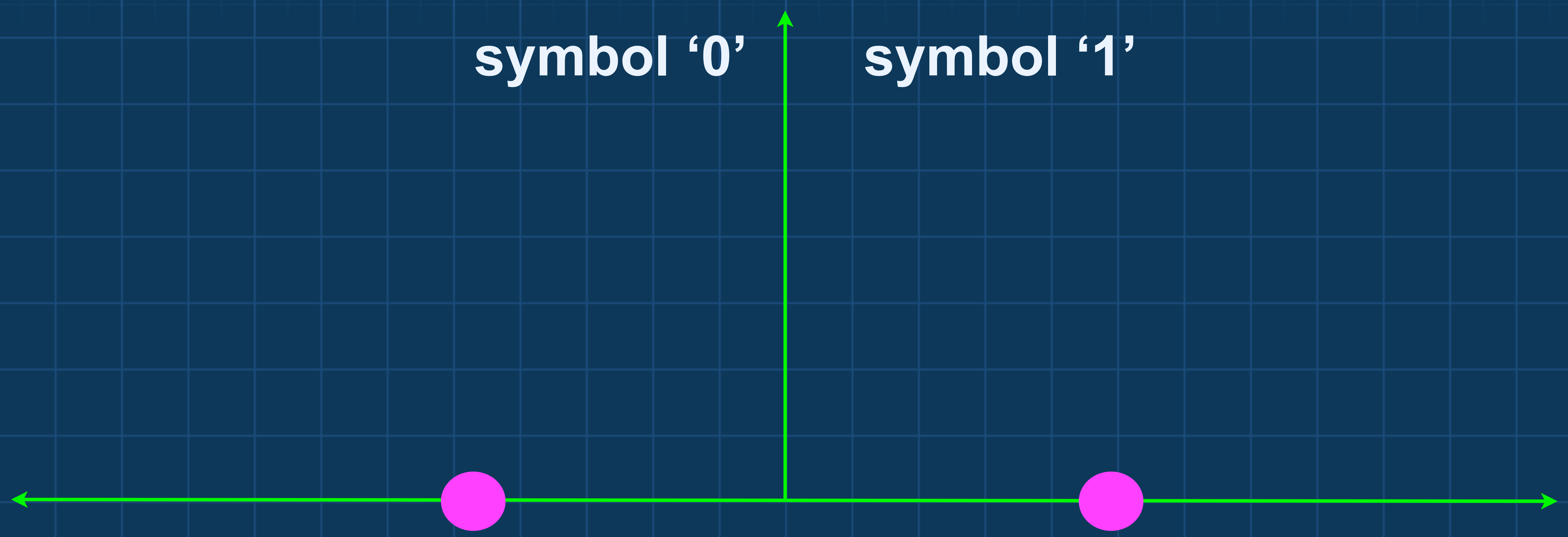
Current workflow is

- Implement prototype of transmit / receive in MATLAB,
- then re-implement in VHDL,
- then re-re-implement in VHDL (once requirements are better understood).

Research Questions

- Can we use use functional programming to complement and support the developments being made in MATLAB?
- Can we build a functional program that allow the tradeoffs which require re-implementation to be avoided?
- Can we gain a stronger assurance of the relationship between the specification and implementation?

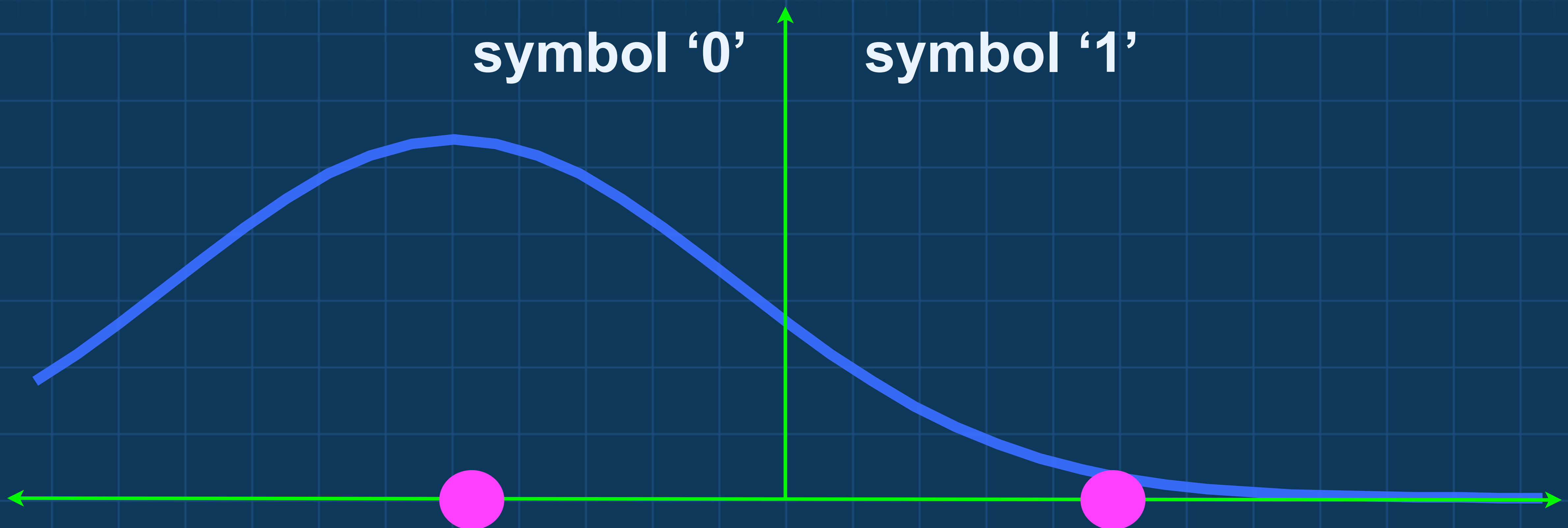
Binary phase-shift keying (BPSK)



- BPSK encoding picks two phases for the binary symbols 0 and 1

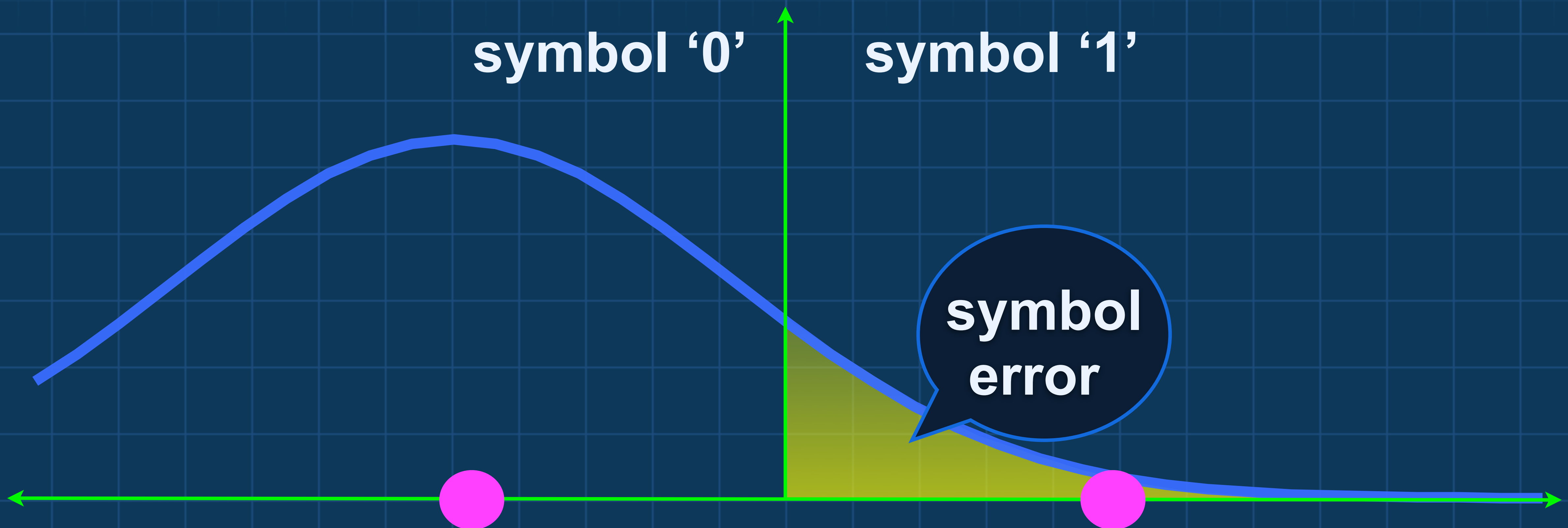
THE UNIVERSITY OF KANSAS

Binary phase-shift keying (BPSK)



- BPSK encoding picks two phases for the binary symbols 0 and 1
- **Always** a possibility of the received symbol being wrong
- Probability density function changes with different signal to noise ratios

Binary phase-shift keying (BPSK)



- BPSK encoding picks two phases for the binary symbols 0 and 1
- **Always** a possibility of the received symbol being wrong
- Probability density function changes with different signal to noise ratios

LDPC encoding a codeword

1
0
1
0
1

1	0	0	0	0	1	1	0	0	1
0	1	0	0	0	0	1	1	1	0
0	0	1	0	0	1	0	1	1	0
0	0	0	1	0	1	0	1	0	1
0	0	0	0	1	0	1	0	1	1

LDPC encoding a codeword

1
0
1
0
1

1	0	0	0	0	1	1	0	0	1
0	1	0	0	0	0	1	1	1	0
0	0	1	0	0	1	0	1	1	0
0	0	0	1	0	1	0	1	0	1
0	0	0	0	1	0	1	0	1	1

1 0 1 0 1 0 0 1 0 0

LDPC encoding a codeword

1
0
1
0
1

Identity Matrix

1	0	0	0	0	1	1	0	0	1
0	1	0	0	0	0	1	1	1	0
0	0	1	0	0	1	0	1	1	0
0	0	0	1	0	1	0	1	0	1
0	0	0	0	1	0	1	0	1	1

1 0 1 0 1 0 0 1 0 0

THE UNIVERSITY OF KANSAS

LDPC encoding a codeword

Message: 1 0 1 0 1

Parity Generation

1	0	0	0	0	1	1	0	0	1
0	1	0	0	0	0	1	1	1	0
0	0	1	0	0	1	0	1	1	0
0	0	0	1	0	1	0	1	0	1
0	0	0	0	1	0	1	0	1	1

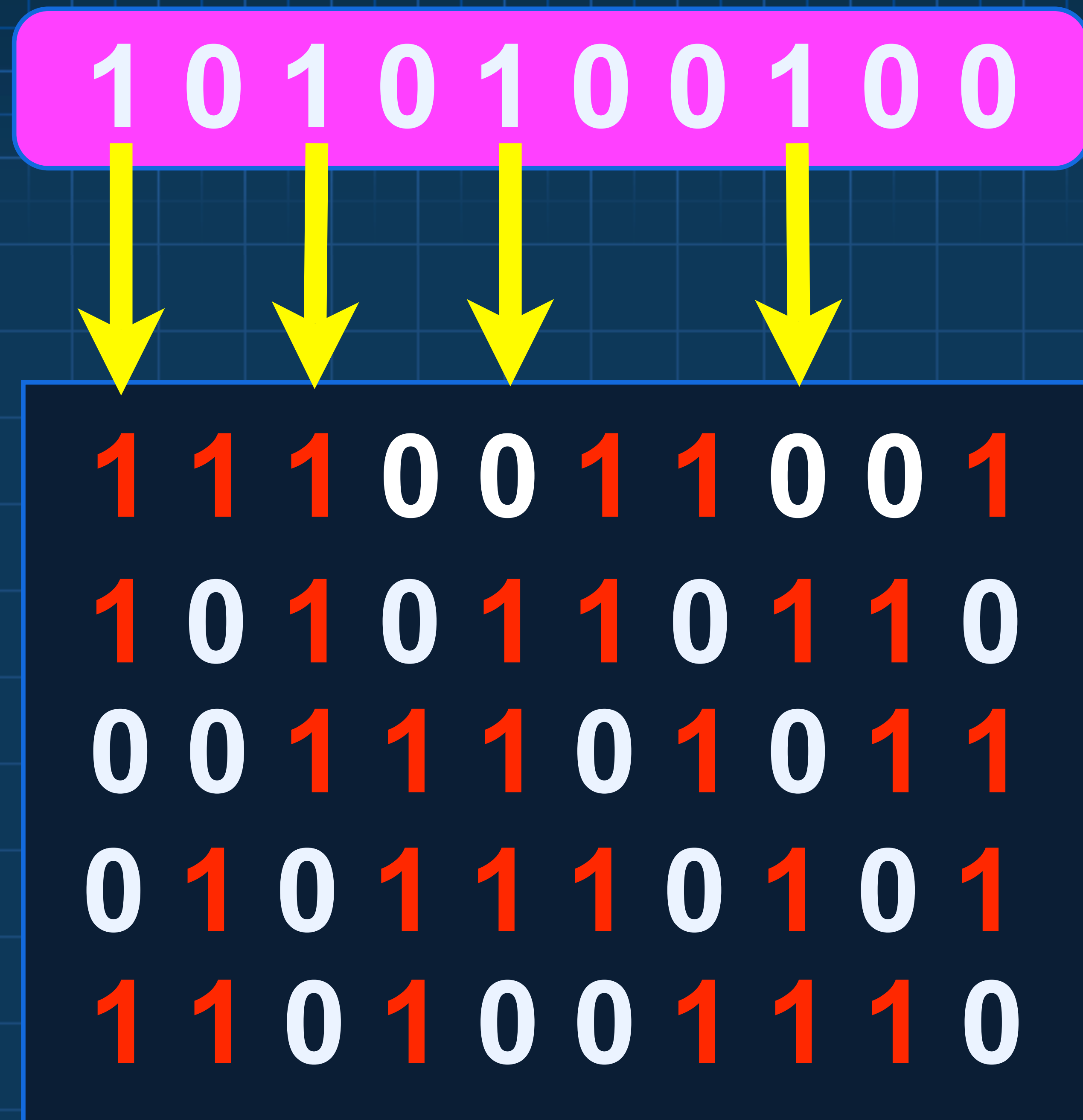
Codeword: 1 0 1 0 1 0 0 1 0 0

LDPC decoding

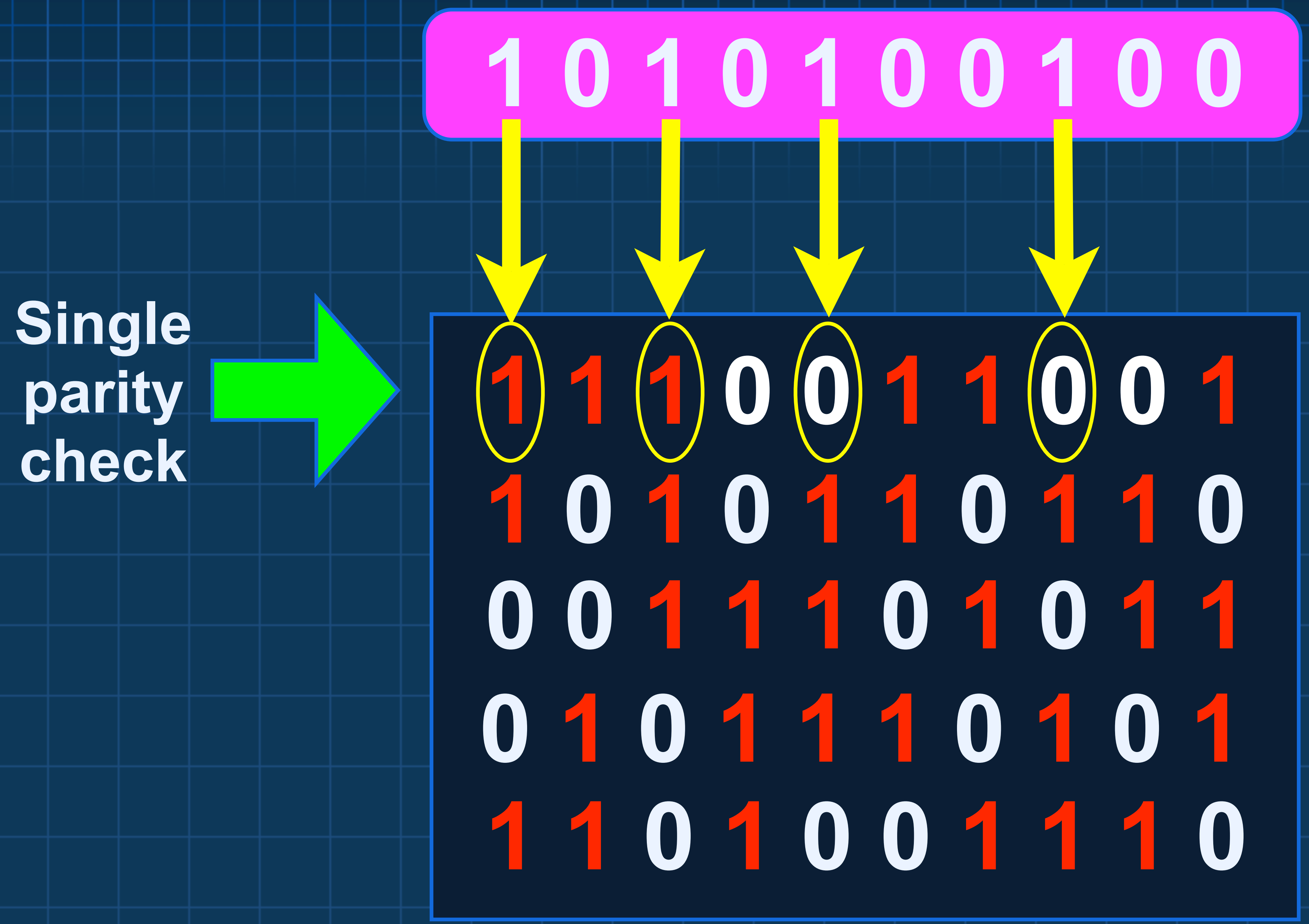
1 0 1 0 1 0 0 1 0 0

Original Code

LDPC decoding



LDPC decoding



THE UNIVERSITY OF KANSAS

LDPC decoding

1 0 1 0 1 0 0 1 0 0

Single parity check

1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

0

THE UNIVERSITY OF KANSAS

LDPC decoding

1 0 1 0 1 0 0 1 0 0

1	1	1	0	0	1	1	0	0	1		0
1	0	1	0	1	1	0	1	1	0		0
0	0	1	1	1	0	1	0	1	1		0
0	1	0	1	1	1	0	1	0	1		0
1	1	0	1	0	0	1	1	1	0		0

LDPC decoding with parity error

1 0 1 1 1 0 0 1 0 0

1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

LDPC decoding with parity error

1 0 1 1 1 0 0 1 0 0

1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

LDPC decoding with parity error

1 0 1 1 1 0 0 1 0 0

Single parity check →

1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

0

LDPC decoding with parity error

1 0 1 1 1 0 0 1 0 0

1	1	1	0	0	1	1	0	0	1		0
1	0	1	0	1	1	0	1	1	0		0
0	0	1	1	1	0	1	0	1	1		0
0	1	0	1	1	1	0	1	0	1		1
1	1	0	1	0	0	1	1	1	0		1

LDPC belief propagation

What does a point need for a successful parity check?

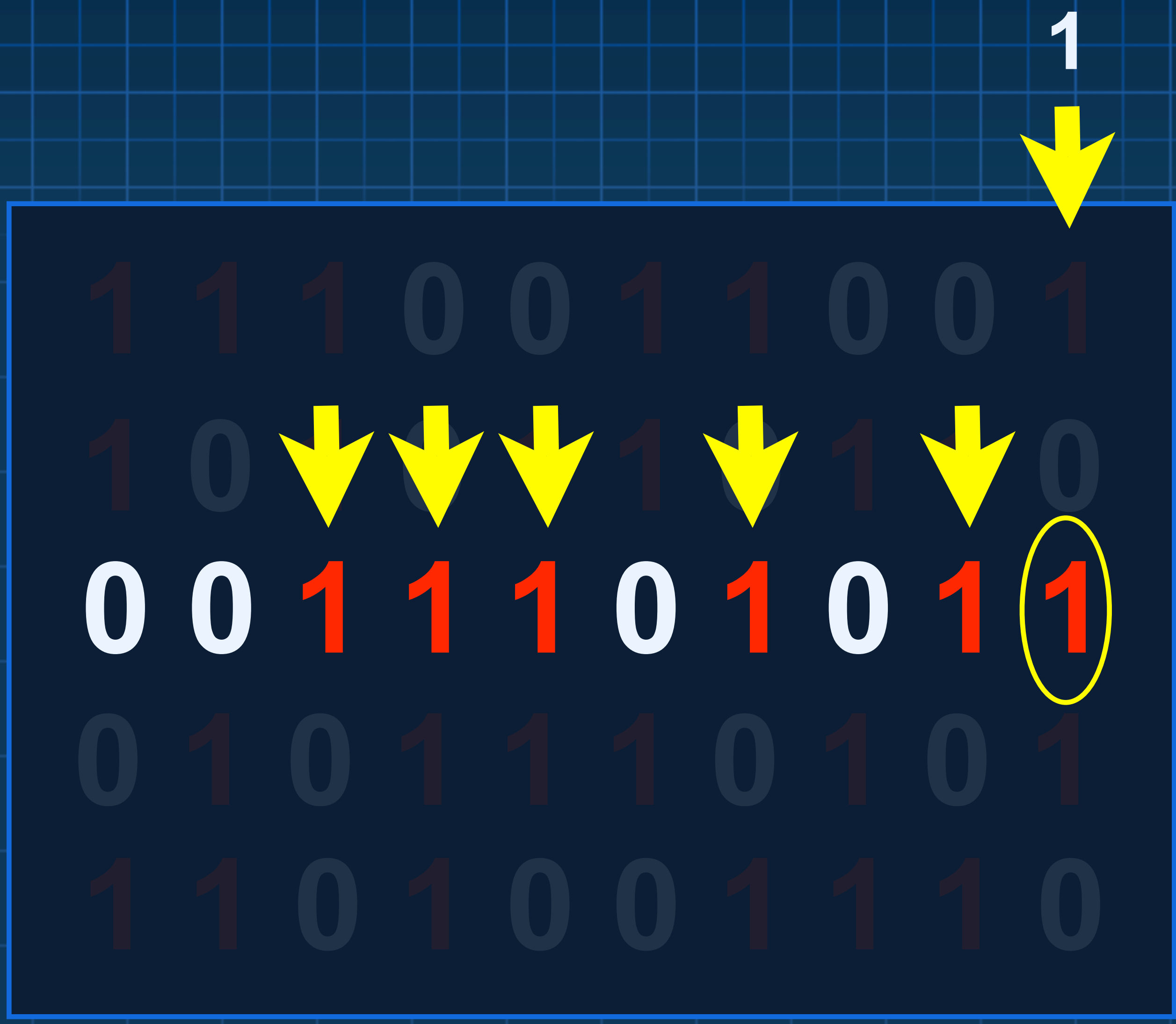
1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

LDPC belief propagation

What does a point need for a successful parity check?

1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

LDPC belief propagation

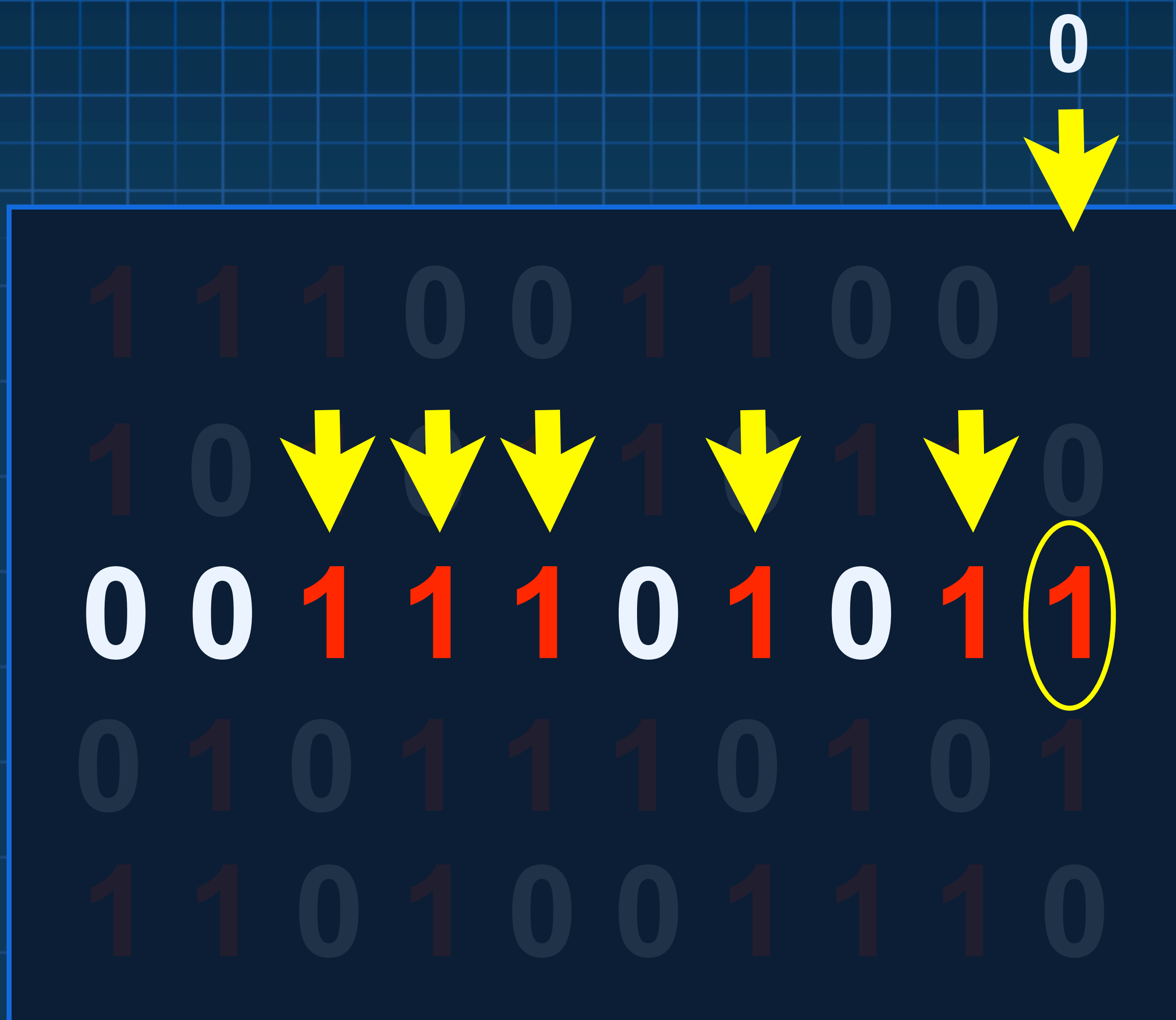


What does a point need for a successful parity check?

If codeword is 1
the parity of activated
vertical siblings
should sum to 1

THE UNIVERSITY OF KANSAS

LDPC belief propagation



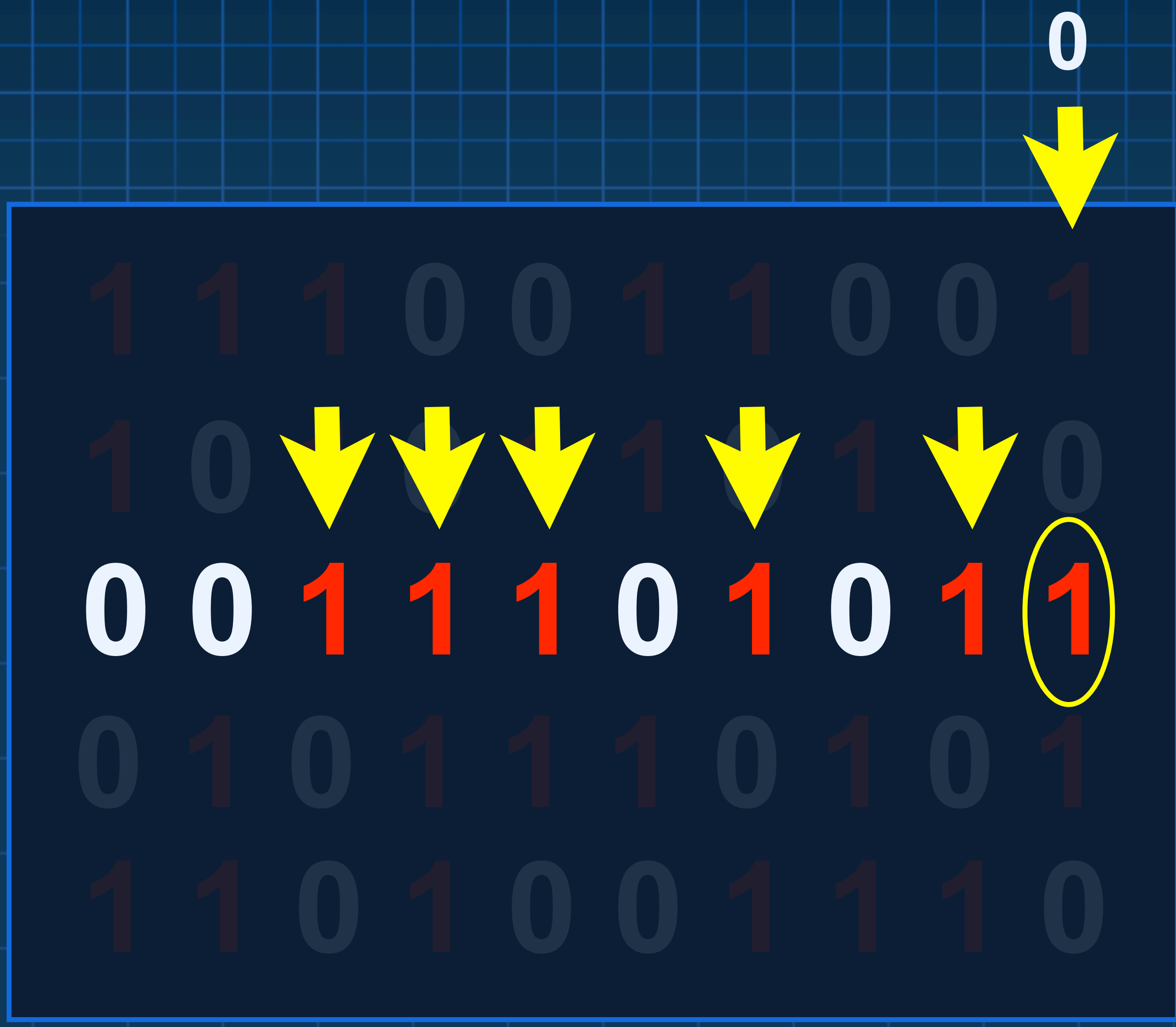
What does a point need for a successful parity check?

**If codeword is 1
the parity of activated
vertical siblings
should sum to 1**

**If codeword is 0
the parity of activated
vertical siblings
should sum to 0**

KU THE UNIVERSITY OF KANSAS

LDPC belief propagation



What does a point need for a successful parity check?

If codeword is 1 the parity of activated vertical siblings should sum to 1

If codeword is 0 the parity of activated vertical siblings should sum to 0

Depending on how certain you are about your input, you can make suggestions to your siblings.

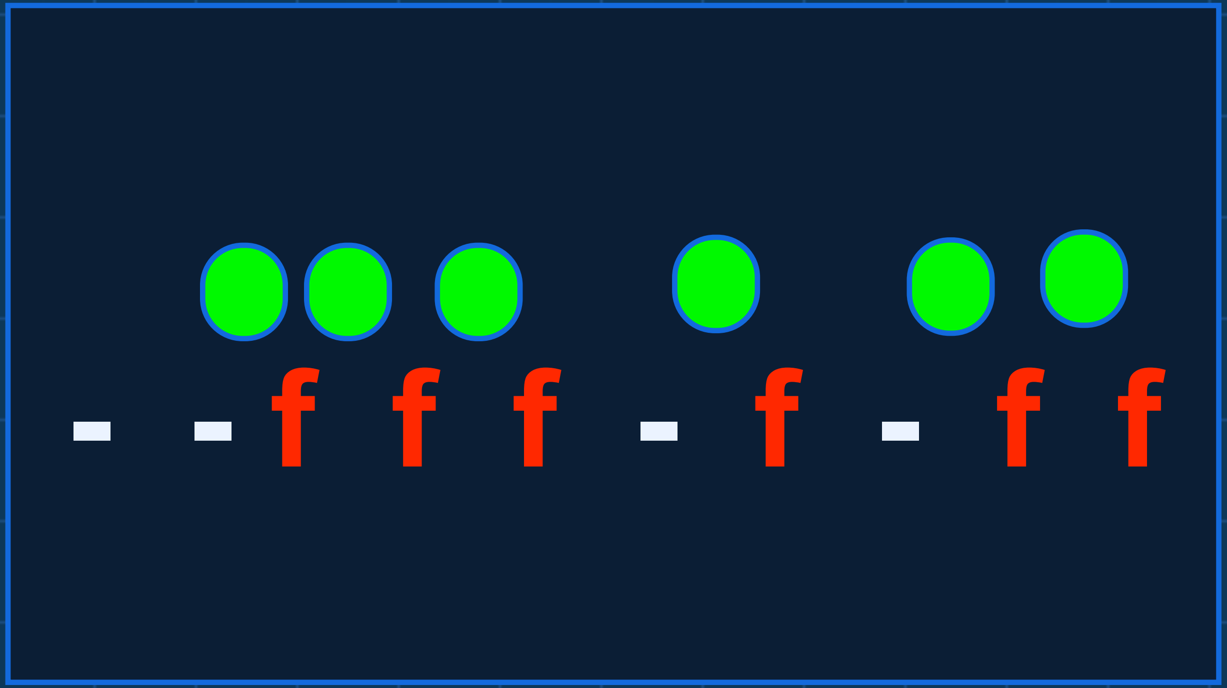
THE UNIVERSITY OF KANSAS

LDPC check nodes

Use **soft** code guesses

1 -1 1 0.5 1 -1 -1 1 -1 -1

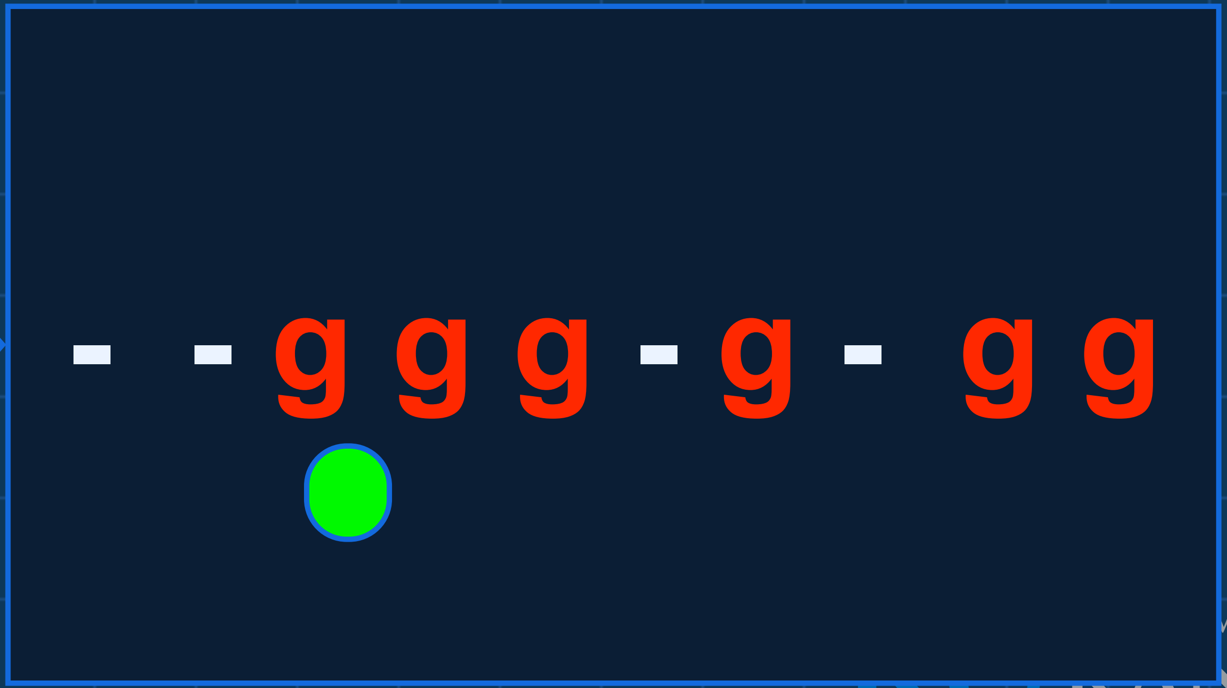
Single Line



$f_j = \tanh(-0.5 * (lam_j - chk_j))$

$g_j = -2 * \tanh^{-1}(\prod f(...))$

Same Single Line



LDPC check nodes

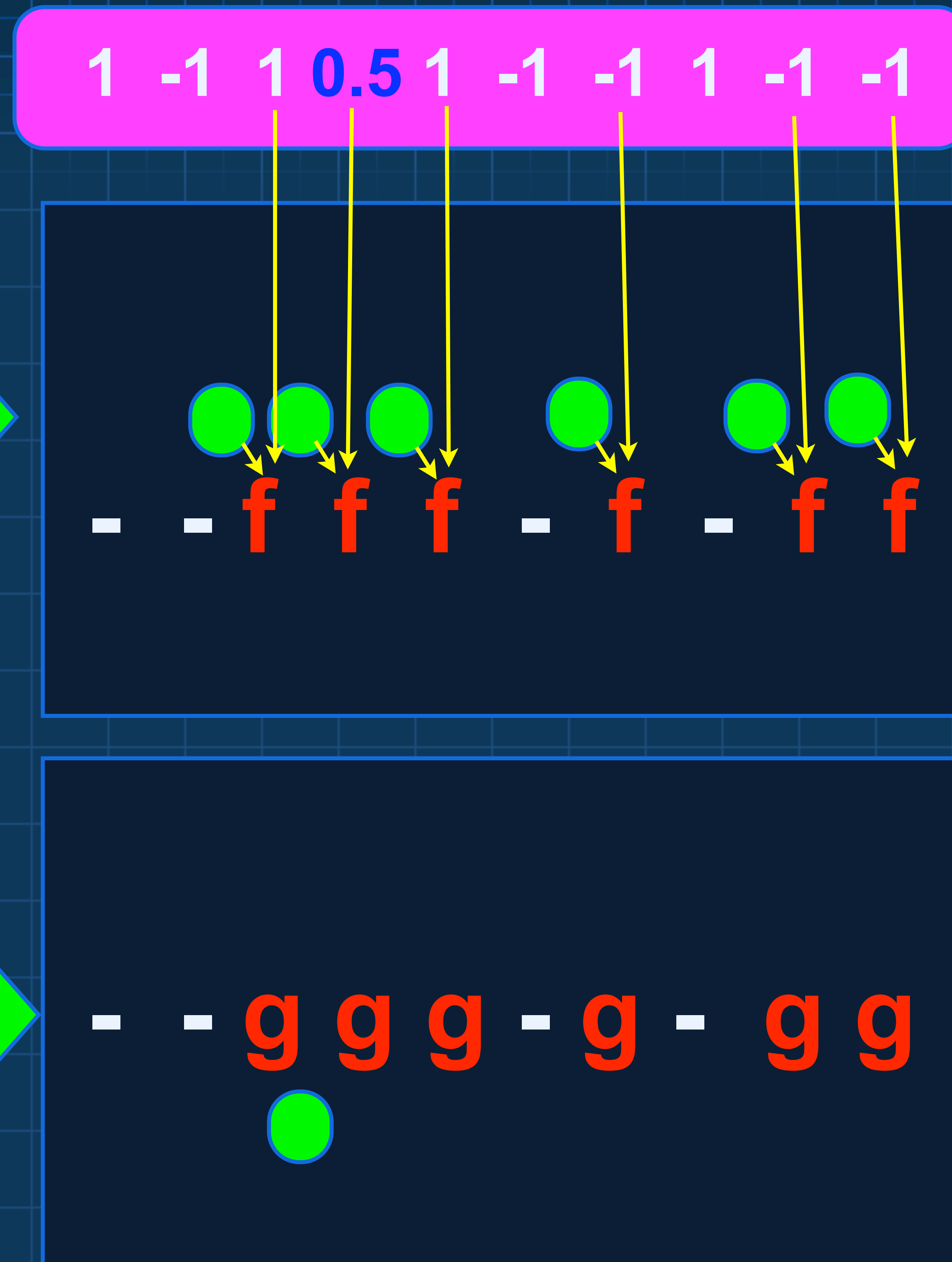
Use soft code guesses

Single Line

$f_j = \tanh(-0.5 * (lam_j - chk_j))$

$g_j = -2 * \tanh^{-1}(\Pi\ f(...))$

Same Single Line



LDPC check nodes

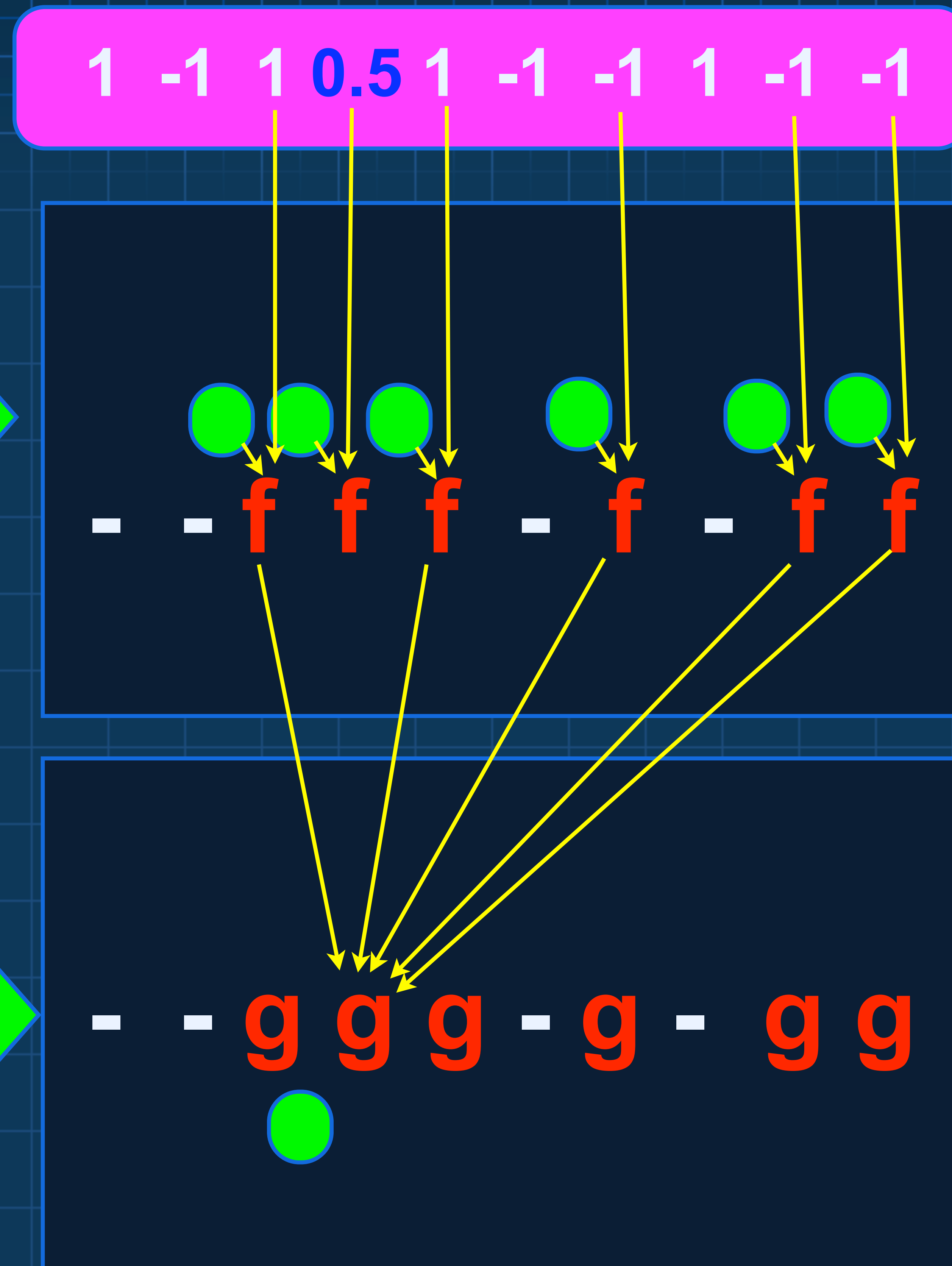
Use **soft** code guesses

Single Line

$f_j = \tanh(-0.5 * (lam_j - chk_j))$

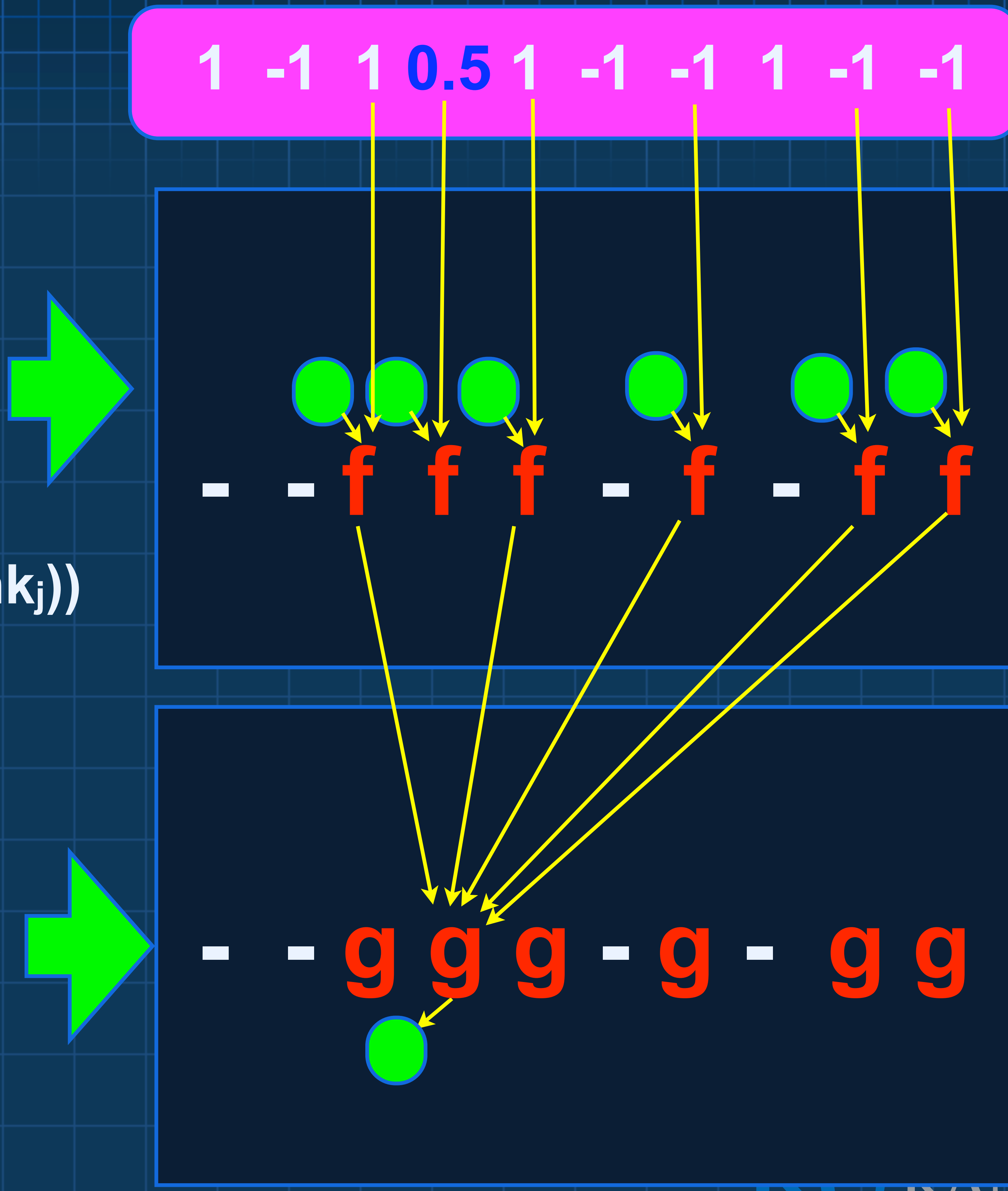
$g_j = -2 * \tanh^{-1}(\Pi\ f(\ldots))$

Same Single Line



LDPC check nodes

Use **soft** code guesses



Single Line

$f_j = \tanh(-0.5 * (lam_j - chk_j))$

$g_j = -2 * \tanh^{-1}(\prod f(...))$

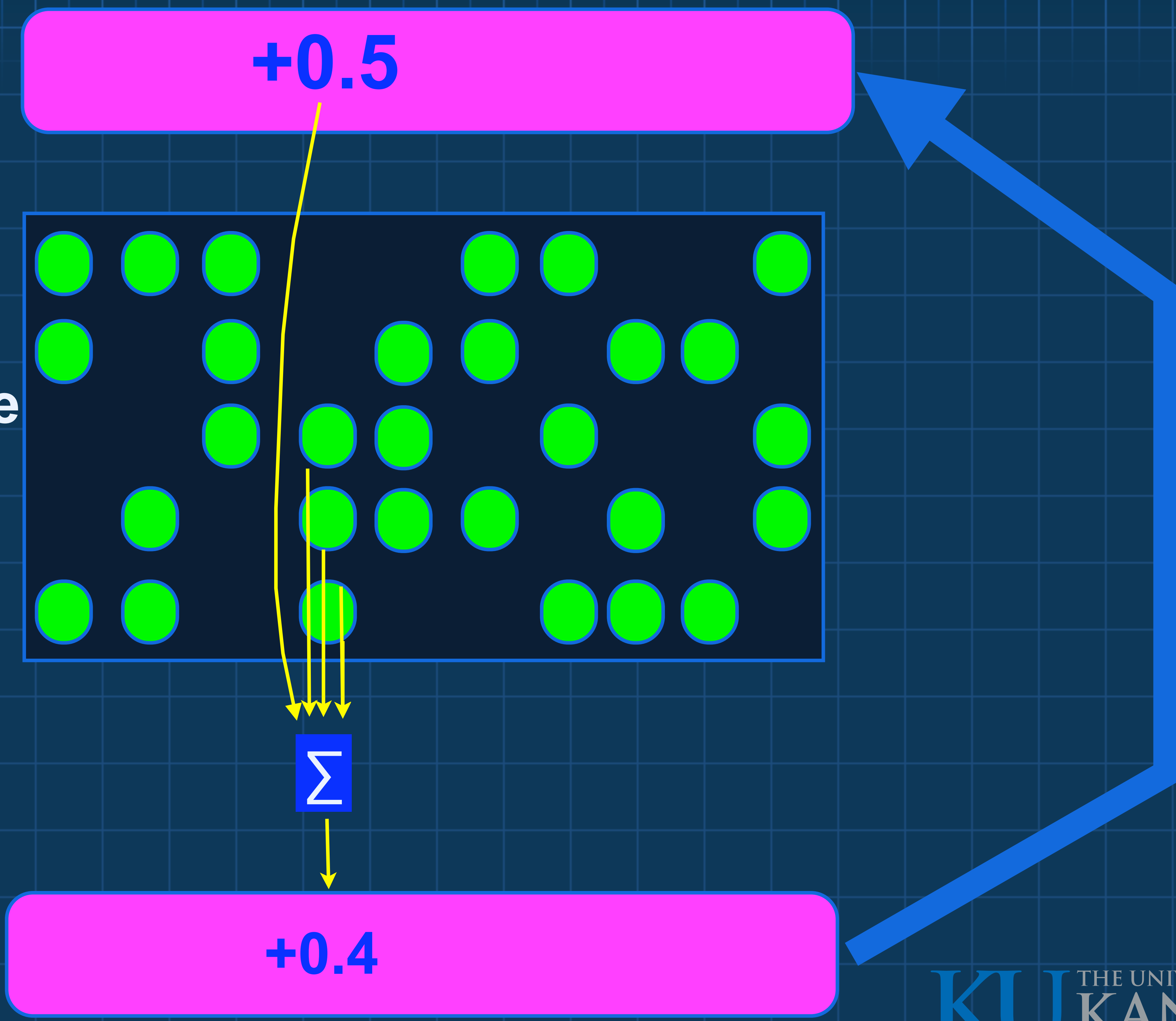
Same Single Line

LDPC iterates over the codeword

Soft codeword

Each '1' in original decode matrix has a check node

Iterate using this new code guess



LDPC Specification

Algorithm 15.2 Iterative Log Likelihood Decoding Algorithm for Binary LDPC Codes

---

Input: A, the received vector $\mathbf{r}$, the maximum # of iterations L, and the channel reliability L_c.
Initialization: Set $\eta_{m,n}^{[0]} = 0$ for all (m, n) with $A(m, n) = 1$.
Set $\lambda_n^{[0]} = L_c r_n$
Set the loop counter $l = 1$.
Check node update: For each (m, n) with $A(m, n) = 1$: Compute

$$\eta_{m,n}^{[l]} = -2 \tanh^{-1} \left(\prod_{j \in \mathcal{N}_{m,n}} \tanh \left(-\frac{\lambda_j^{[l-1]} - \eta_{m,j}^{[l-1]}}{2} \right) \right) \quad (15.33)$$

Bit node update: For $n = 1, 2, \ldots, N$: Compute

$$\lambda_n^{[l]} = L_c r_n + \sum_{m \in \mathcal{M}_n} \eta_{m,n}^{[l]} \quad (15.34)$$

Make a tentative decision: Set $\hat{c}_n = 1$ if $\lambda_n[l] > 0$, else set $\hat{c}_n = 0$.
If $A\hat{\mathbf{c}} = 0$, then **Stop**. Otherwise, if #iterations $< L$, loop to **Check node update**
Otherwise, declare a decoding failure and **Stop**.

From
Error Correction Coding: Mathematical Methods and Algorithms,
Tood K. Moon, Wiley-Interscience.

LDPC In Haskell

```
loop options lc n a@(A a_rref aRows aCols) ne lam orig_lam
                | BitMatrix.cardinality ans == 0 = return (Just c_hat)
                | n > iterations options         = return Nothing
                | otherwise = loop options lc (succ n) a ne' lam' orig_lam
  where
            c_hat :: Matrix x U1
            c_hat = (\ c -> if c > 0 then 1 else 0) <$> lam

            ans :: BitMatrix (y, X1)
            ans = a_rref `BitMatrix.mm` BitMatrix.fromMatrix (M.unitColumn c_hat)

            ne' :: SM.Matrix (y,x) a
            ne' = SM.fromAssocList 0
                   [ ((m,n),
                      -2 * (atanh (product
                                 [tanh ((-((lam ! j) - (ne SM.! (m,j))))/2)
                                          | j <- BitMatrix.toList (aRows ! m)
                                          , j /= n
                                          ])))
                   | (m,n) <- BitMatrix.toList a_rref ]

            lam' :: Matrix x a
            lam' = forAll $ \ n -> (orig_lam ! n)
                      + sum [ ne' SM.! (m,n)
                            | m <- BitMatrix.toList (aCols ! n) ]
```

Lava

- Lava is an Embedded Domain Specific Language (EDSL) for describing hardware level concerns
- Haskell acts as the host language
  - Lava is a Library in Haskell
  - Lava programs are Haskell programs
  - Haskell programs are not necessarily Lava programs

```haskell
halfAdder
  :: (Signal Bool, Signal Bool)
  -> (Signal Bool, Signal Bool)
halfAdder (a,b) = (carry,sum)
  where carry = and2 (a,b)
        sum   = xor2 (a,b)
```

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.all;

entity half is
  port (i0 : in std_logic;
        i1 : in std_logic;
        o0 : out std_logic;
        o1 : out std_logic);
end entity half;

architecture str of half is
signal sig_o0_4 : std_logic;
signal sig_o0_2 : std_logic;
begin
  sig_o0_4 <= i0 XOR i1;
  sig_o0_2 <= i0 AND i1;
  o0 <= sig_o0_2;
  o1 <= sig_o0_4;
end architecture;
```

THE UNIVERSITY OF KANSAS

Specifications and Implementations

- We want to link together our Haskell executable specification with our Lava implementations
  - Ease of test generation and debugging
  - Stepping stones provide placeholders for assurance arguments
  - Possibility of a future design methodology

Specification ⟵ **Both Haskell** ⟶ **Implementation**

Specification	Implementation
Runs on Haskell RTS	VHDL / ModelSim / FPGAs execution platform
Big step functions	Fine grain execution (in efficient implementations)
No concept of clock cycles	It is all about the cycles
Haskell recursion for control flow	Control logic direct datums
Mutable state is global	Mutable state is local

THE UNIVERSITY OF KANSAS

Kansas Lava

- Started as a teaching tool for FP class
- Used to generate binaries for orbital simulation in summer 2009
- Grown into a primary FP research platform at KU
- We want to address the range of computations that can be mapped over functors

Classical Haskell	Hawk	Kansas Lava	Lava
Execution	Execution		Synthesis & Execution
Value-based computation model	Signal-based computation model		Signal-based computation model
Arbitrary computation over functors	Arbitrary computation over functors		restricted computation over functors

Family of Kansas Lava Functors

Functor f

$$\text{map} :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$

$$\text{map}\ g\ .\ \text{map}\ h\ = \text{map}\ (g\ .\ h)$$

$$\text{map}\ id\ = id$$

Strategy

Refactor computation to use synthesizable and structural functors

- Pipe_a
- Stream
- Mem_a
- Seq
- List
- Enabled
- Comb
- Matrix_x
- Array_x
- Maybe

Key

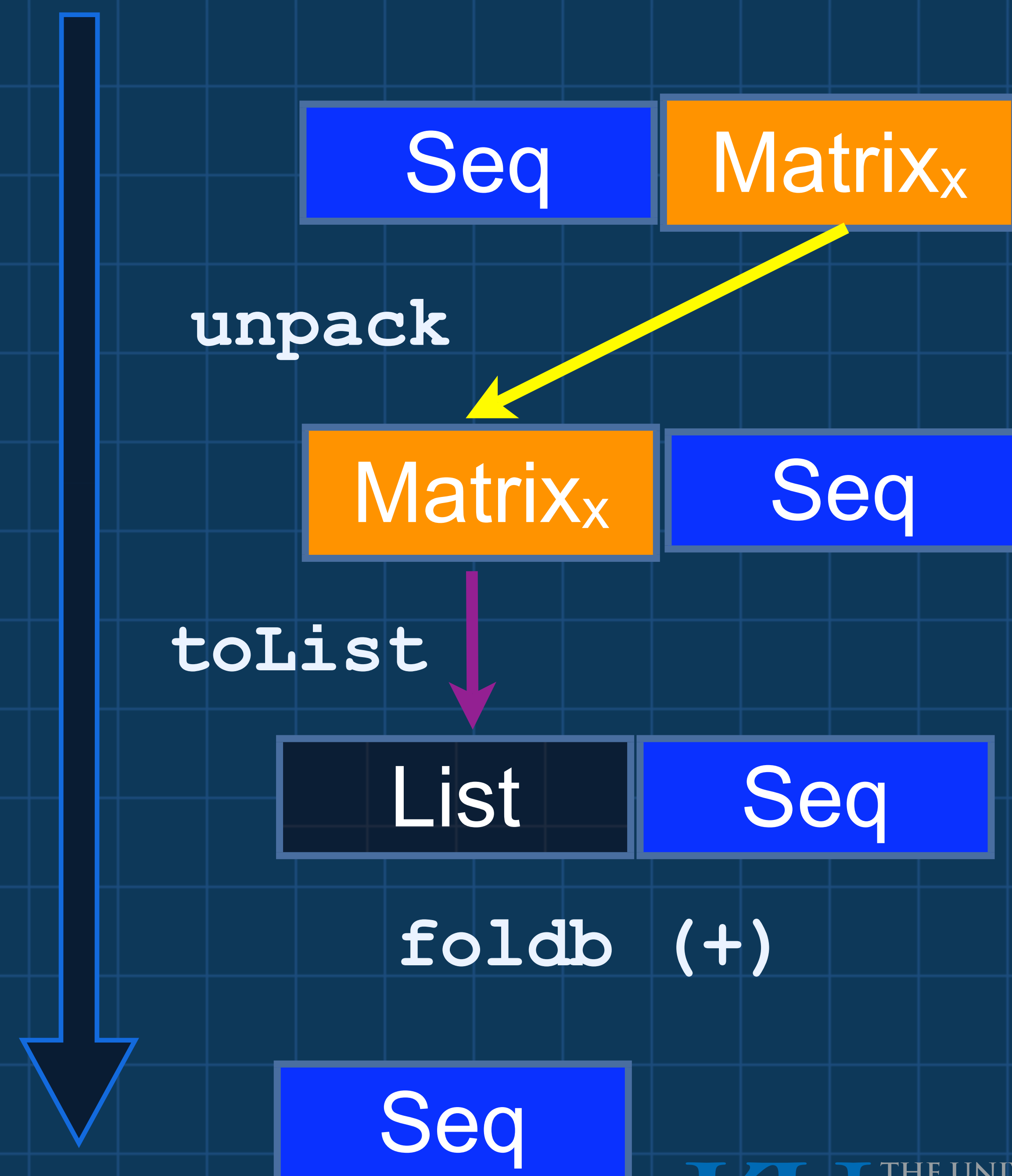
- Synthesizable
- Structural
- Haskell

THE UNIVERSITY OF KANSAS

Kansas Lava Example

```
sumMatrix :: (...) => Seq (Matrix x a) -> Seq a
sumMatrix = foldb (+)
          . M.toList
          . unpack
```

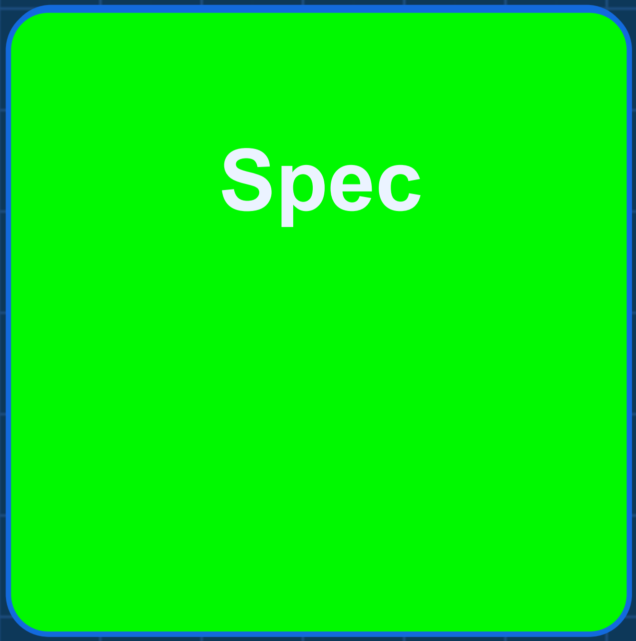
- Primitives like "+" are defined over Seq.
- Kansas Lava programs use functions like unpack to move values so that primitives can act.



THE UNIVERSITY OF KANSAS

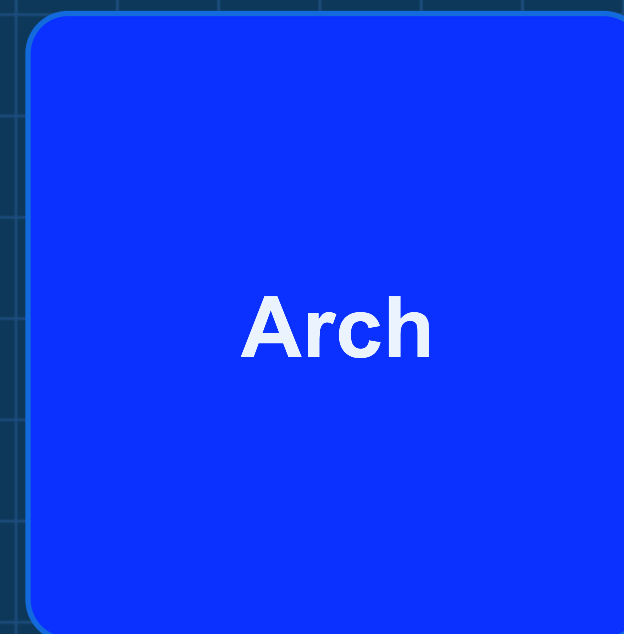
Gameplan

- Refactor specification in Haskell to architecture, reflecting where computation should take place
  - Computation is placed by architecture
  - Sub-components should be synthesizable in hardware
  - This is the push stage
- Refine architecture to use Kansas Lava types
  - sub-components are joined to make larger synthesizable components
  - This is the pull stage
- The result is a synthesizable circuit
  - This circuit reflects the chosen architecture
  - This circuit implements the specification.



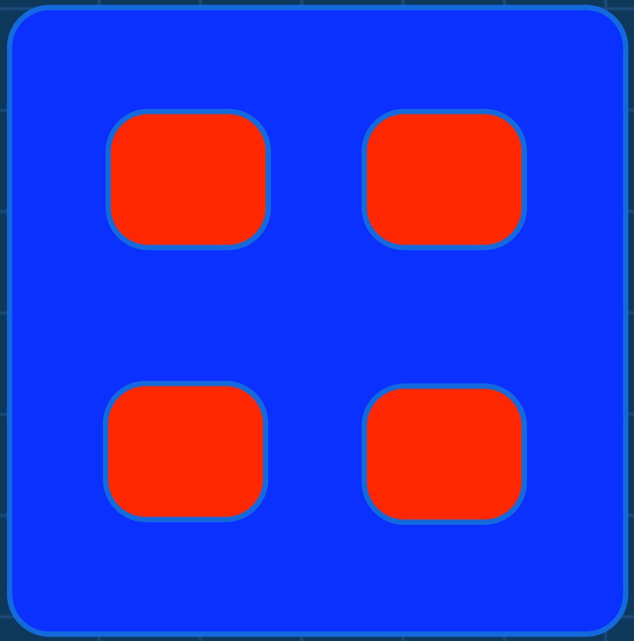
Gameplan

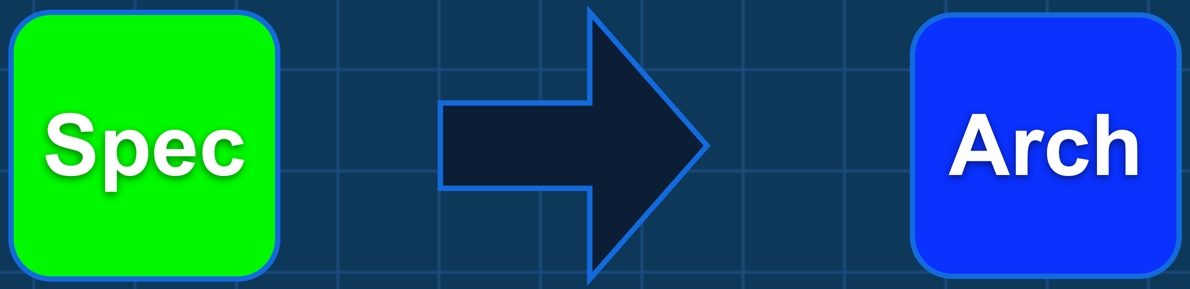
- Refactor specification in Haskell to architecture, reflecting where computation should take place
  - Computation is placed by architecture
  - Sub-components should be synthesizable in hardware
  - This is the push stage
- Refine architecture to use Kansas Lava types
  - sub-components are joined to make larger synthesizable components
  - This is the pull stage
- The result is a synthesizable circuit
  - This circuit reflects the chosen architecture
  - This circuit implements the specification.



Gameplan

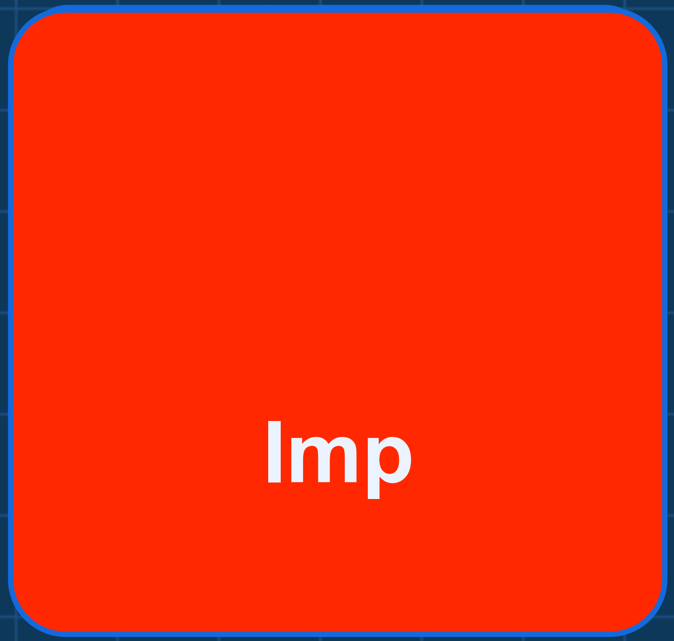
- Refactor specification in Haskell to architecture, reflecting where computation should take place
  - Computation is placed by architecture
  - Sub-components should be synthesizable in hardware
  - This is the push stage
- Refine architecture to use Kansas Lava types
  - sub-components are joined to make larger synthesizable components
  - This is the pull stage
- The result is a synthesizable circuit
  - This circuit reflects the chosen architecture
  - This circuit implements the specification.



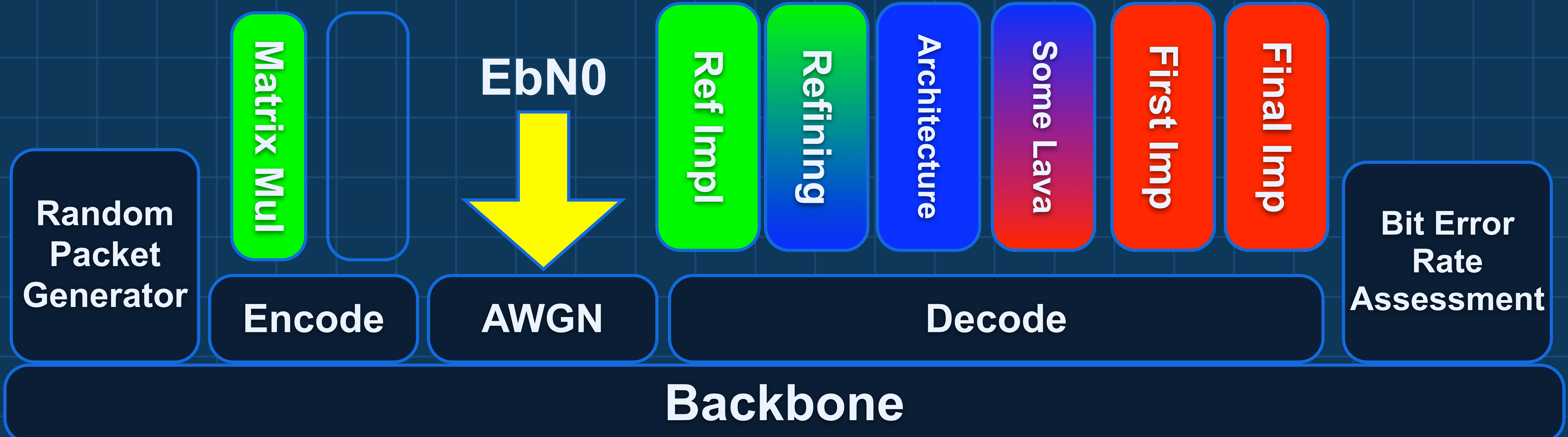


Gameplan

- Refactor specification in Haskell to architecture, reflecting where computation should take place
  - Computation is placed by architecture
  - Sub-components should be synthesizable in hardware
  - This is the push stage
- Refine architecture to use Kansas Lava types
  - sub-components are joined to make larger synthesizable components
  - This is the pull stage
- The result is a synthesizable circuit
  - This circuit reflects the chosen architecture
  - This circuit implements the specification.

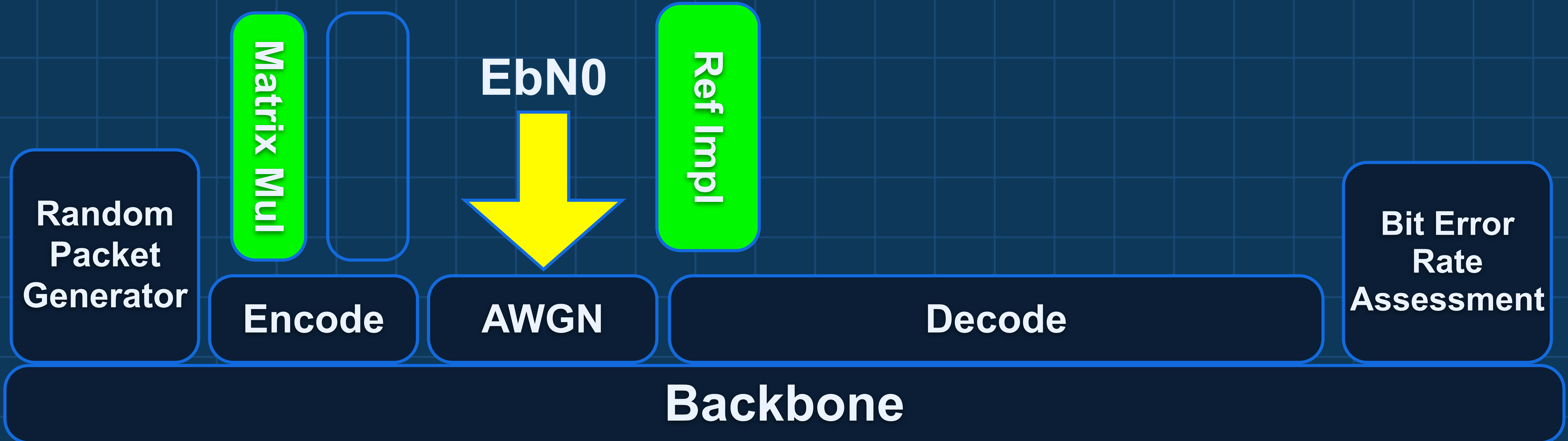


LDPC Backbone for Model, Architecture and Implementation



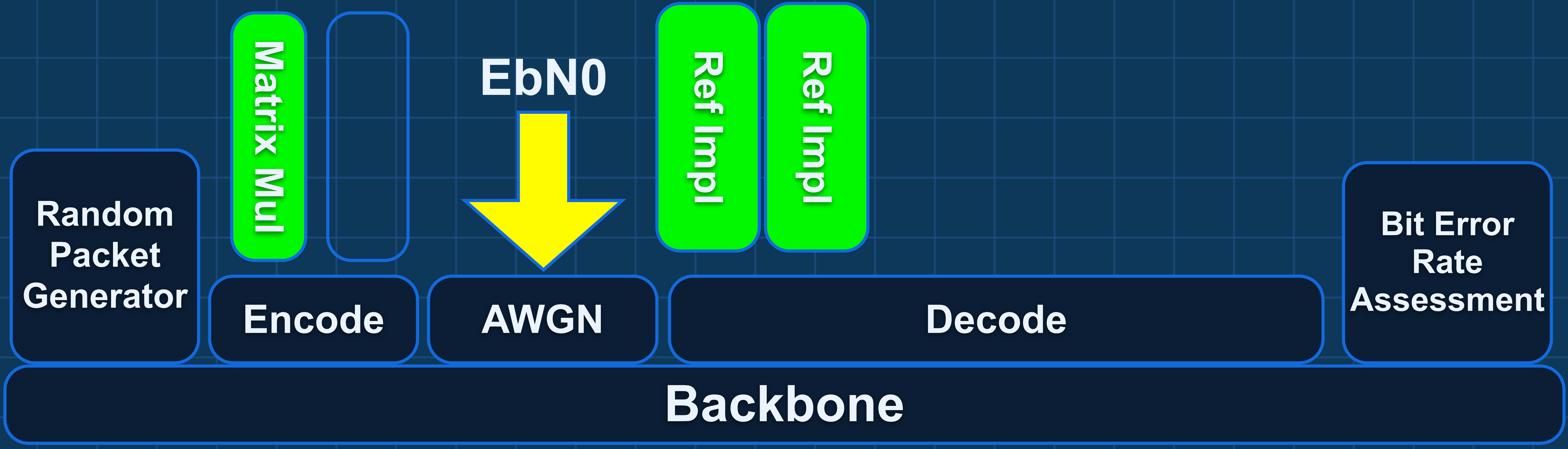
THE UNIVERSITY OF KANSAS

LDPC Backbone for Model, Architecture and Implementation



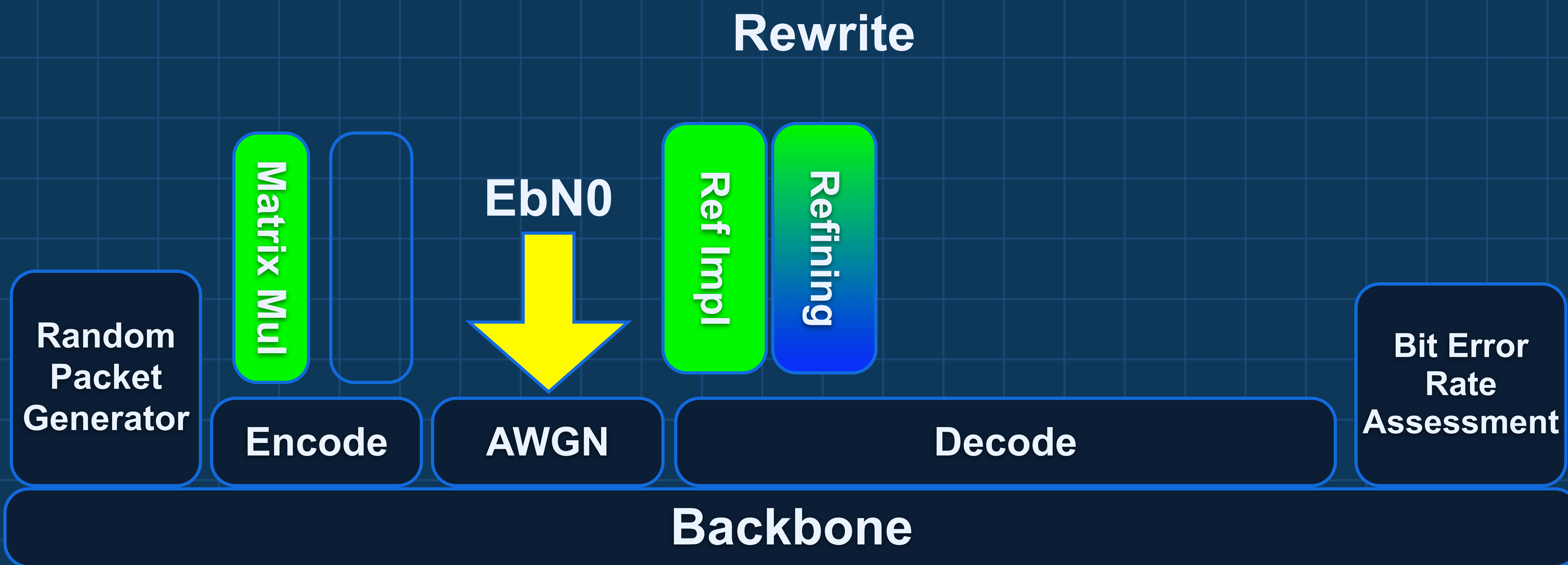
THE UNIVERSITY OF KANSAS

LDPC Backbone for Model, Architecture and Implementation

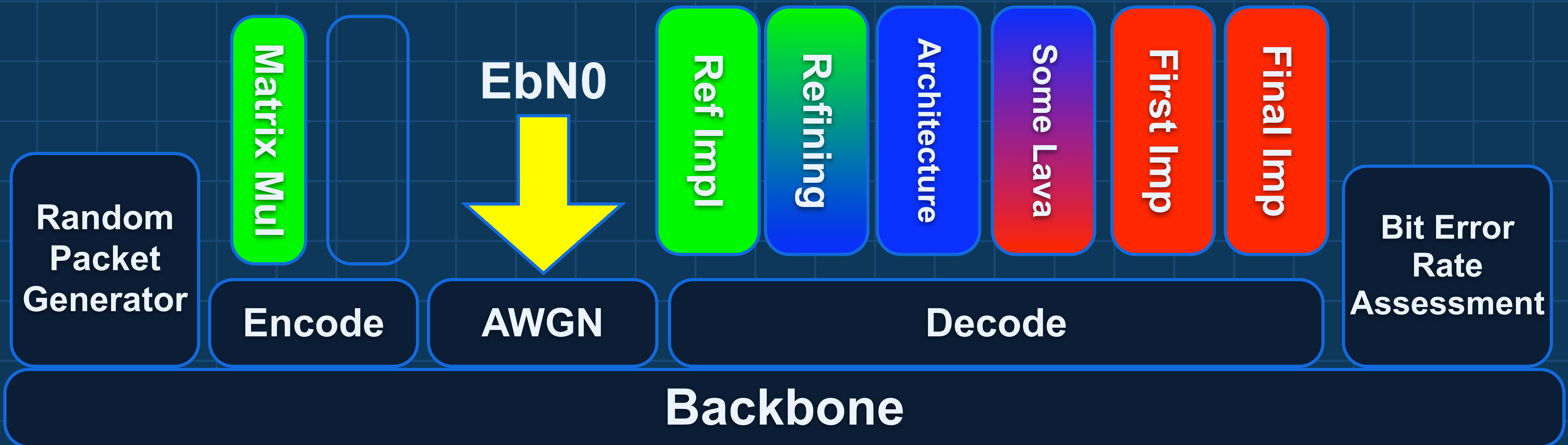


THE UNIVERSITY OF KANSAS

LDPC Backbone for Model, Architecture and Implementation

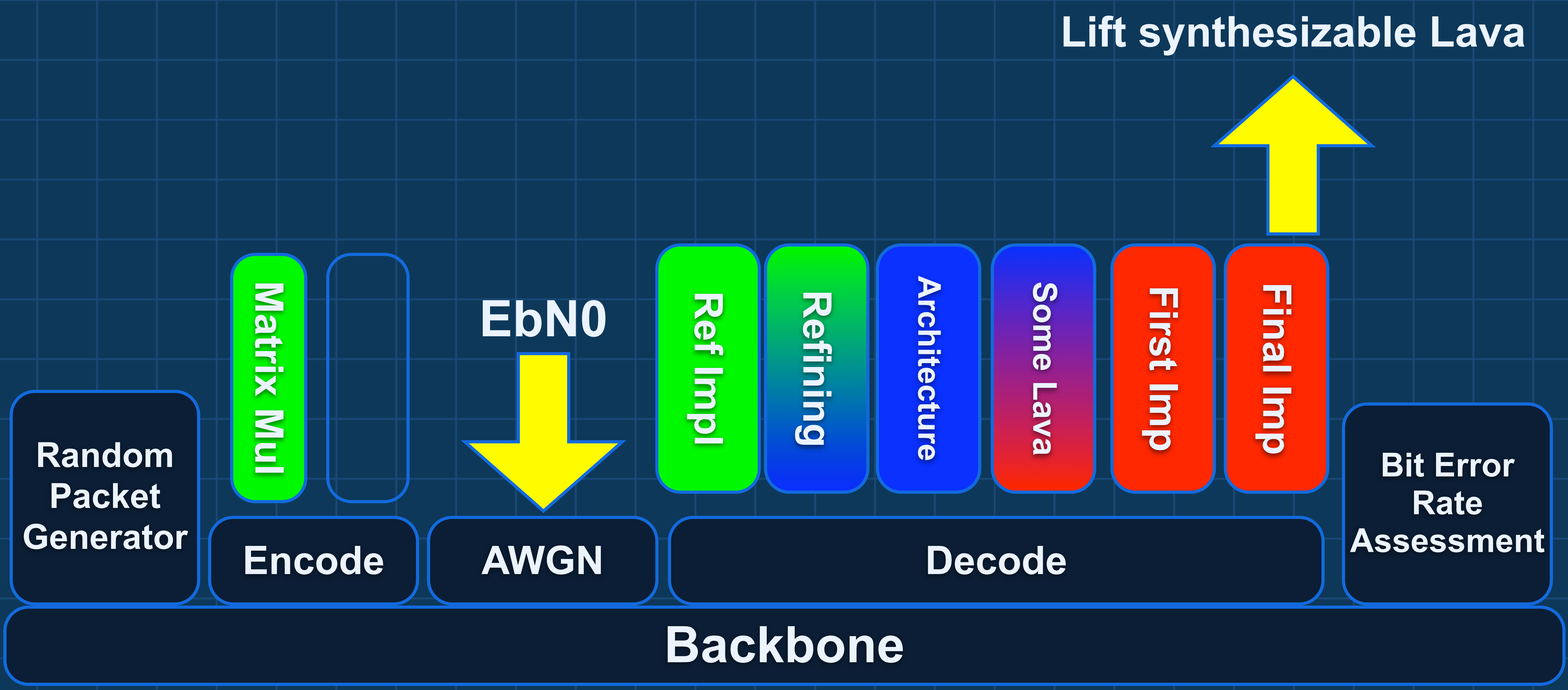


LDPC Backbone for Model, Architecture and Implementation



THE UNIVERSITY OF KANSAS

LDPC Backbone for Model, Architecture and Implementation



THE UNIVERSITY OF KANSAS

Divide and Conquer LDPC

1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

Divide and Conquer LDPC

1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -	- - -
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

Dividing LDPC horizontally requires performing an addition to combine answers

addition is associative

Divide and Conquer LDPC

1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

Dividing LDPC horizontally requires performing an addition to combine answers

addition is associative

Dividing LPDC vertically is <u>much</u> more challenging

There was considerable sharing between vertical siblings

KU THE UNIVERSITY OF KANSAS

min* – Common Implementation Trick

- tanh and $\tanh^{-1}$ are tricky in hardware
- Common implementation trick is to use the min* function instead

min* x y = signum(x) * signum(y) * (min (abs x) (abs y))

- This costs about 0.2dB
- min* when quantized is associative
- fold min* (...) replaces $\tanh^{-1}(\Pi \ldots)$

Before

$$f_j = \tanh(-0.5 * (lam_j - chk_j))$$

$$g_j = -2 * \tanh^{-1}(\Pi\ f(\ldots))$$

After

$$f_j = -1 * (lam_j - chk_j)$$

$$g_j = -0.75 * \text{fold min* } (f(\ldots))$$

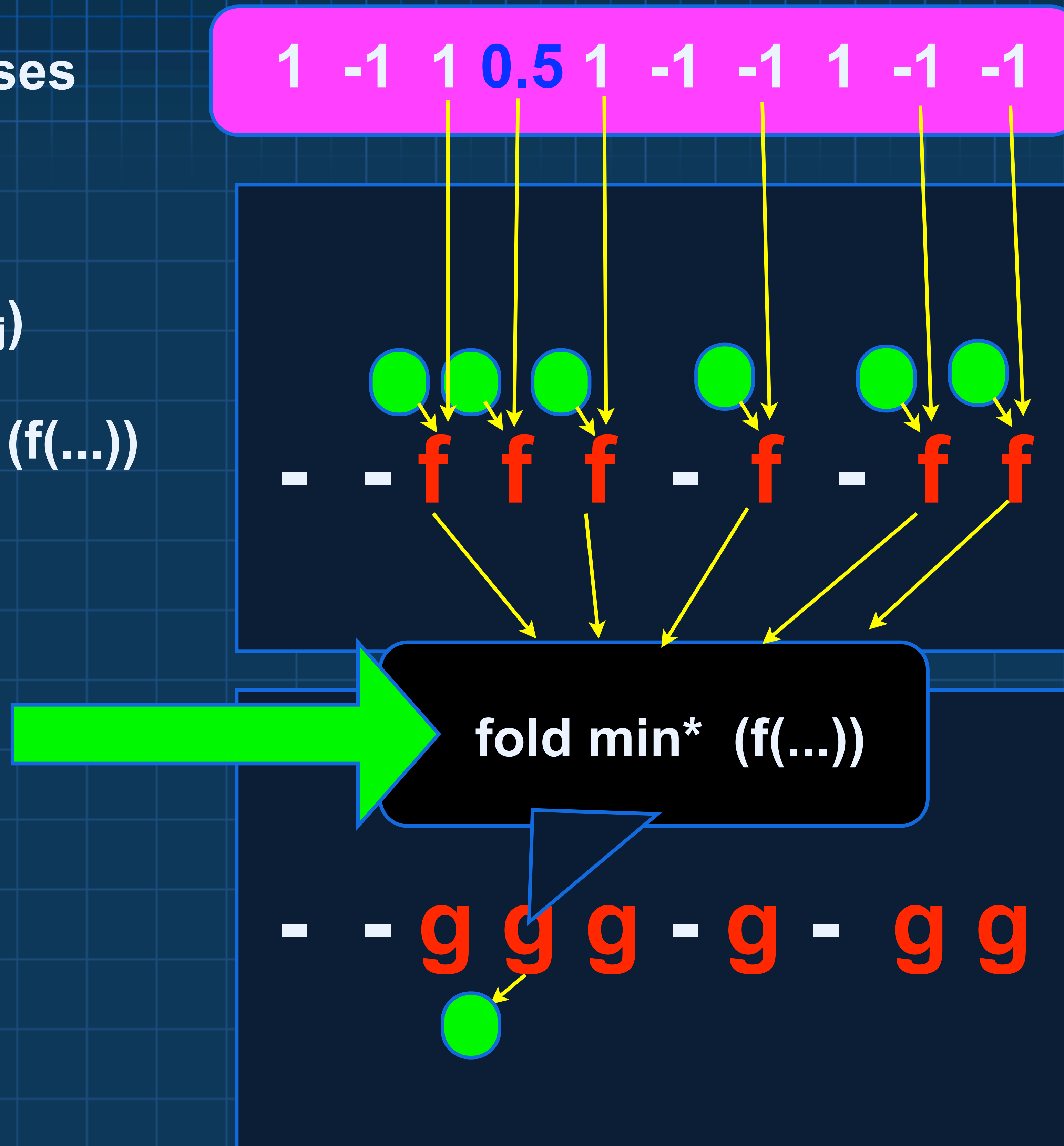
LDPC check nodes with min*

Use soft code guesses

$f_j = -1 * (lam_j - chk_j)$

$g_j = -0.75 * \text{fold min*}\ (f(...))$

fold min* inside g



LDPC check nodes with min*

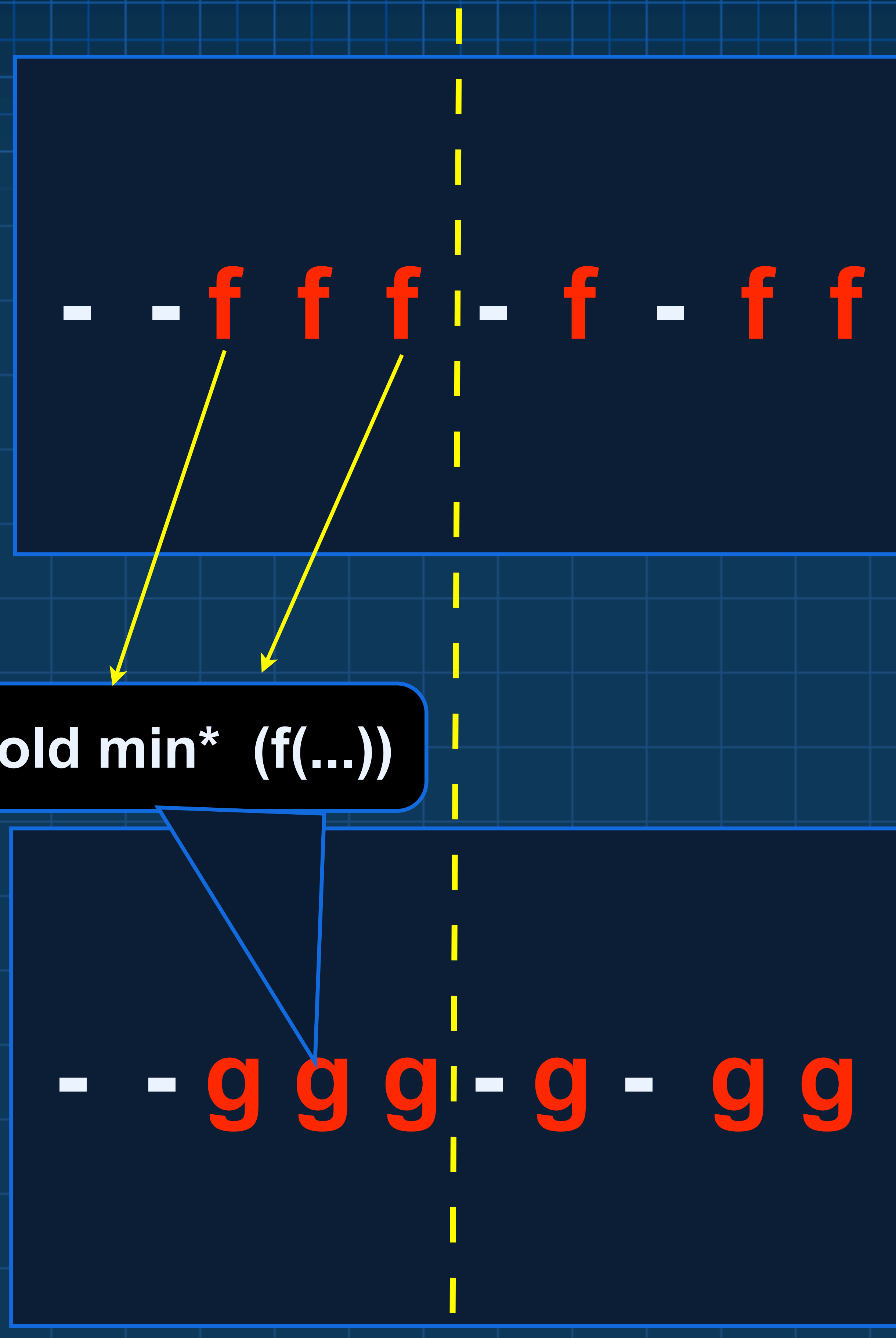
- - f f f - f - f f

- - g g g - g - g g

LDPC check nodes with min*

LDPC check nodes with min*

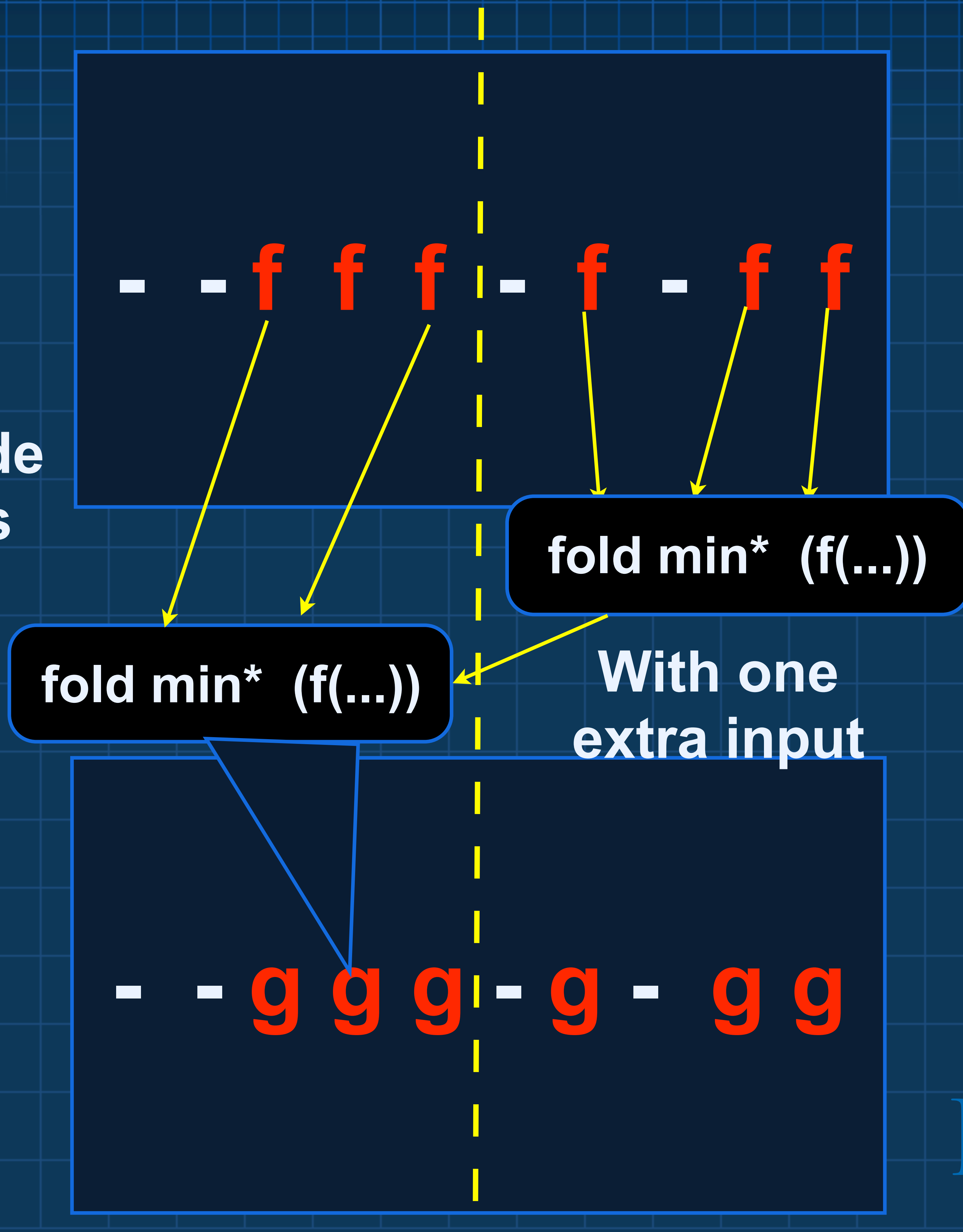
Inside one side the mixing is the same...



THE UNIVERSITY OF KANSAS

LDPC check nodes with min*

Inside one side the mixing is the same...



Cell Architecture

1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

In JPL LDPC code the matrix is built from many rotated identity matrixes (and empty matrixes)

Cell Architecture

1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

256 x 256

0	1
1	0

In JPL LDPC code the matrix is built from many rotated identity matrixes (and empty matrixes)

Cell Architecture

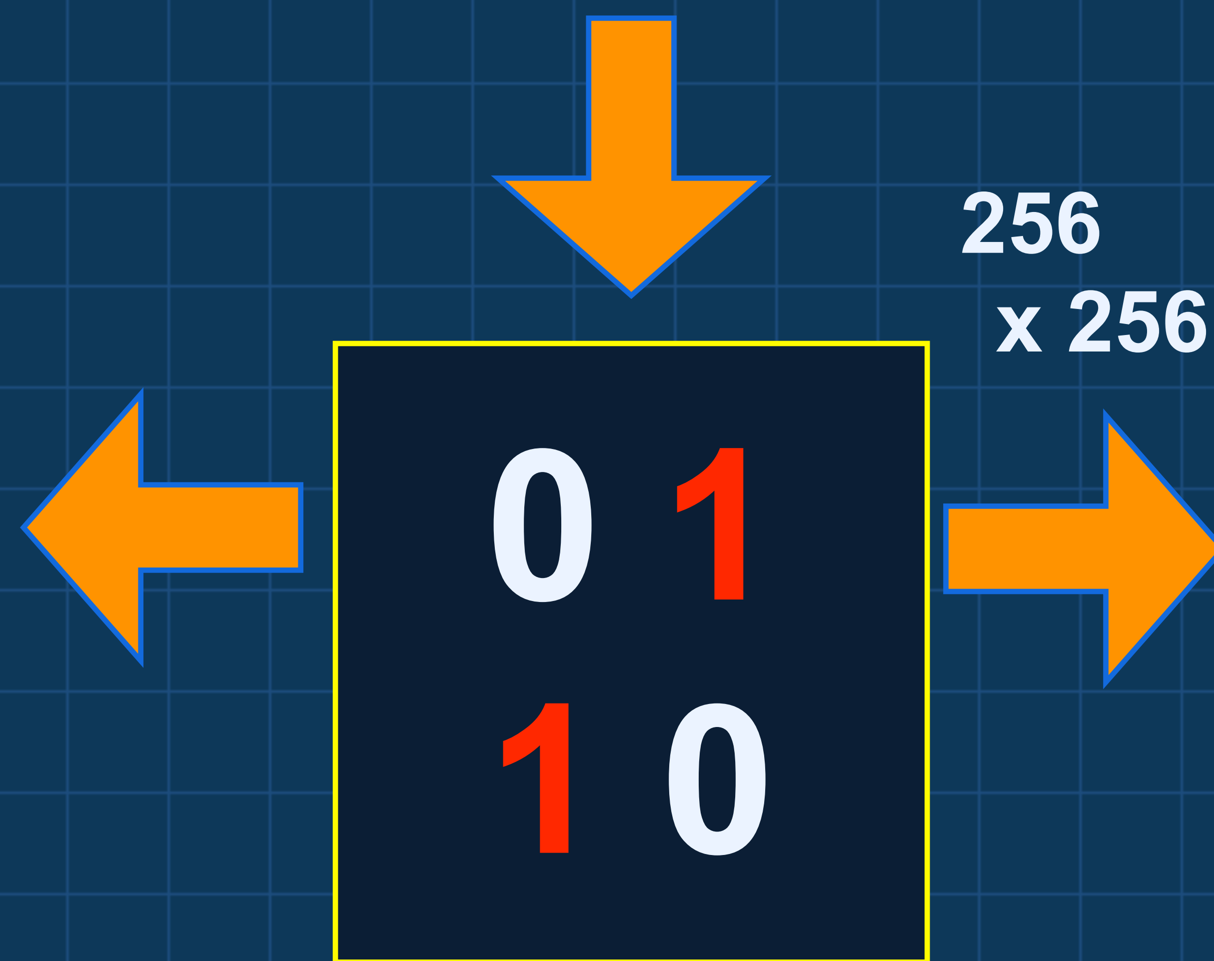
1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

0	1
1	0

256 x 256

In JPL LDPC code the matrix is built from many rotated identity matrixes (and empty matrixes)

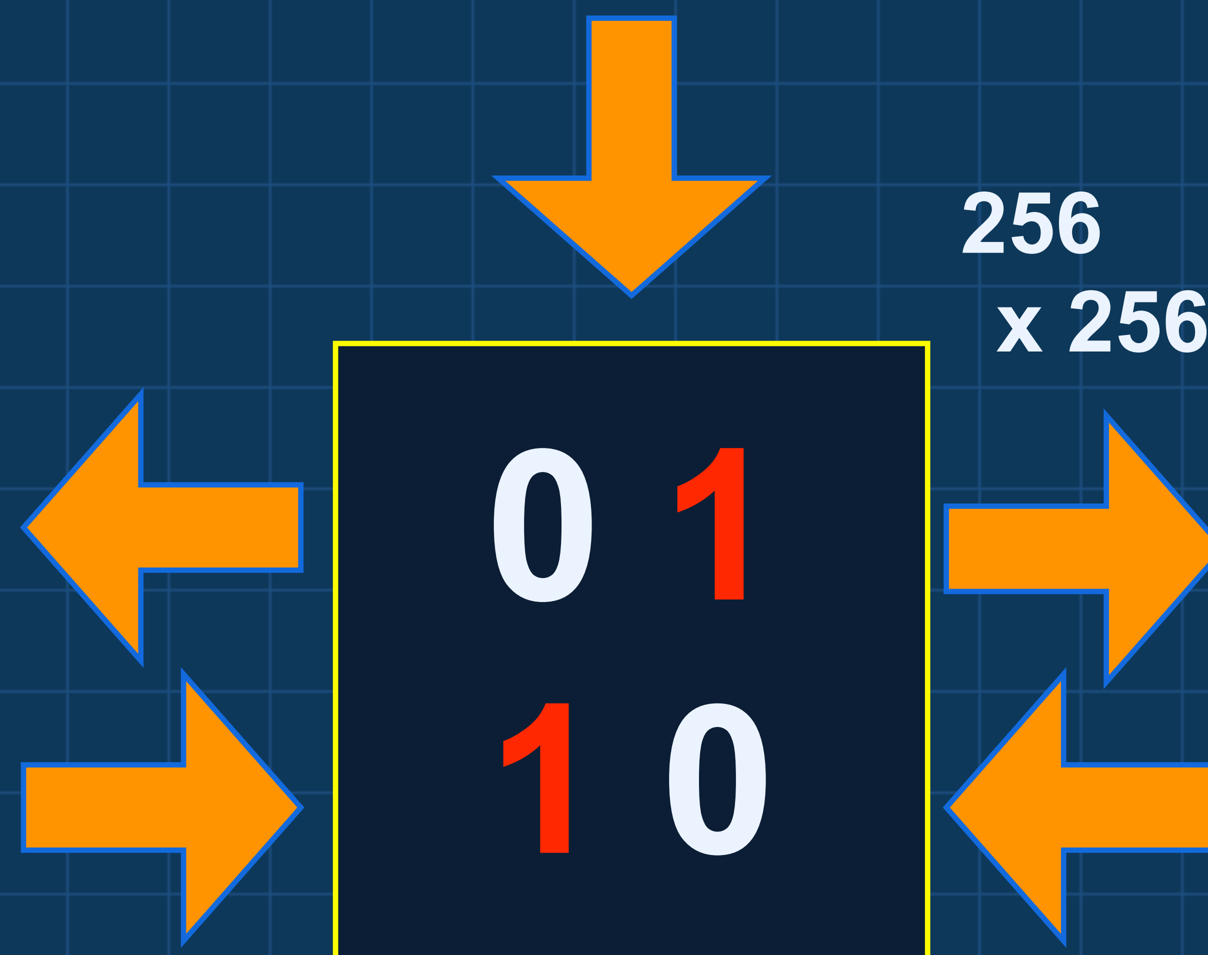
Cell Architecture



In JPL LDPC code
the matrix is built from
many rotated
identity matrixes
(and empty matrixes)

Cell Architecture

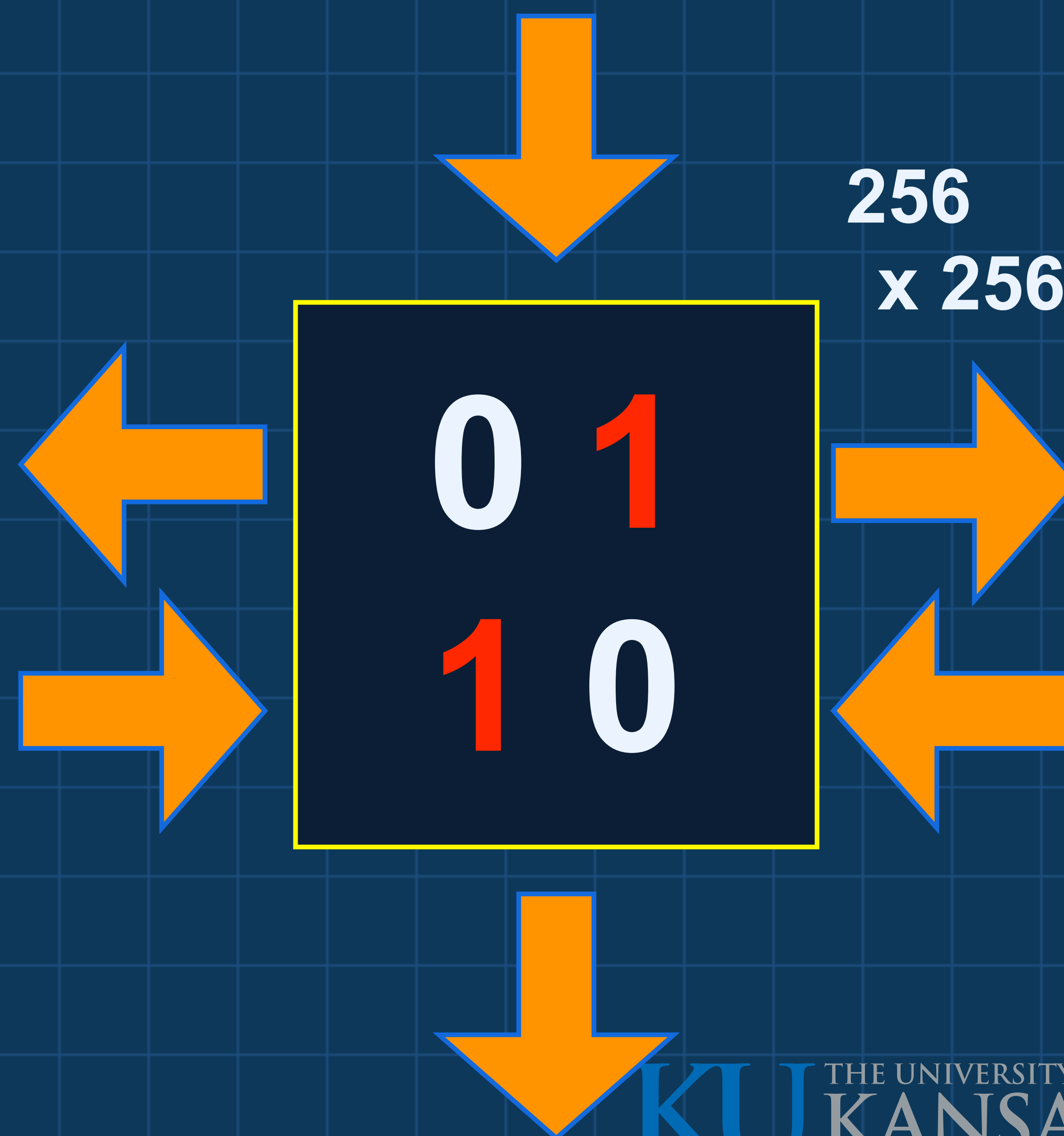
1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0



In JPL LDPC code
the matrix is built from
many rotated
identity matrixes
(and empty matrixes)

Cell Architecture

1	1	1	0	0	1	1	0	0	1
1	0	1	0	1	1	0	1	1	0
0	0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0

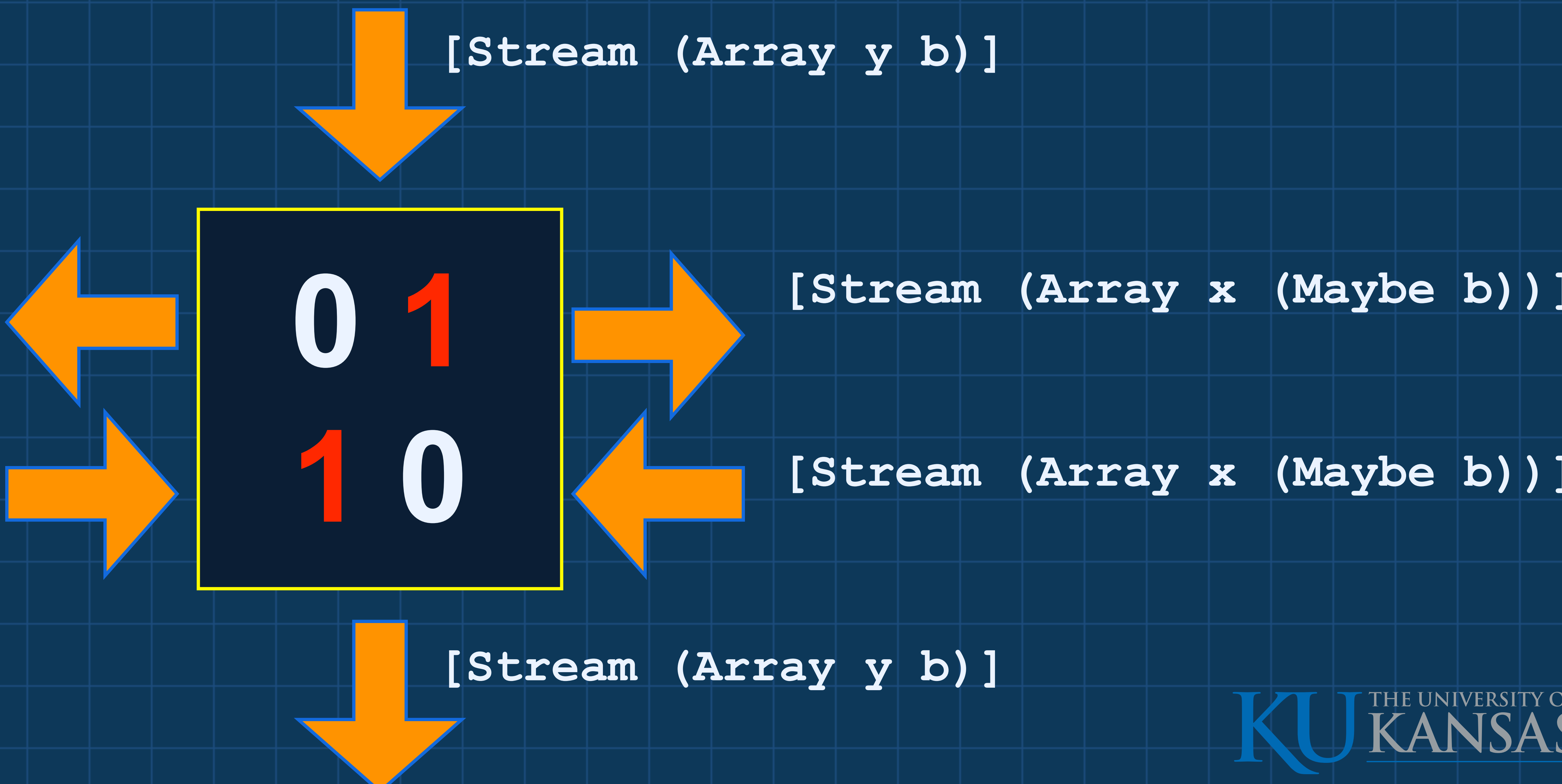


In JPL LDPC code the matrix is built from many rotated identity matrixes (and empty matrixes)

THE UNIVERSITY OF KANSAS

Type of Cell Architecture

```
:: ([Stream (Array y b)], [Stream (Array x (Maybe b))])
-> ([Stream (Array x (Maybe b))], [Stream (Array y b)])
```



THE UNIVERSITY OF KANSAS

Dimensions of a Channel

- Our channels are a composition of functors
- Consider our neighbor sharing channel

```
[Stream (Array x (Maybe b))]
```

List . Stream . Array$_x$. Maybe

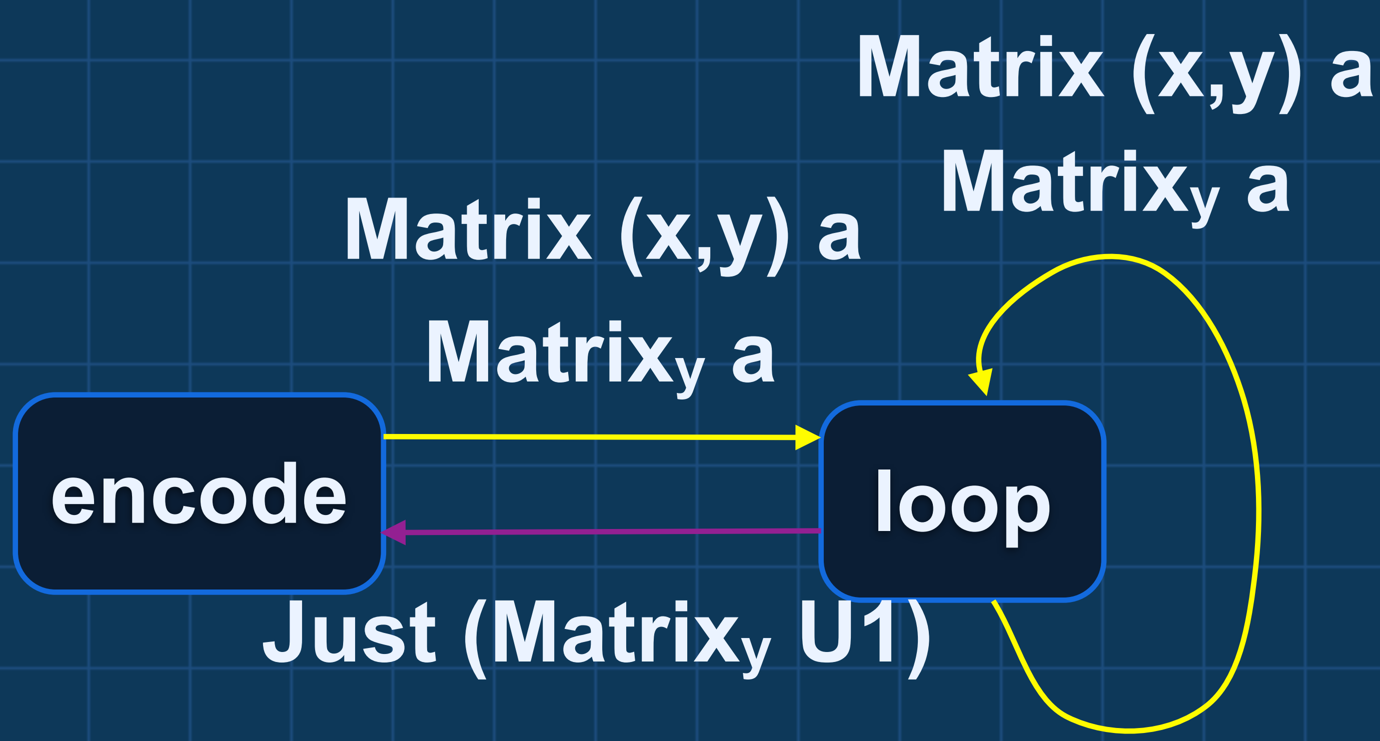
List	Stream	Array$_x$	Maybe
Many Cells	**Over Time**	**Collection**	**Optional**

THE UNIVERSITY OF KANSAS

Specification to Architecture (1)

- Fission into driver and execution unit
- Idea is execution unit will become our hardware entity
- Driver can be used to (automatically) generate test vectors.

```
loop :: Matrix y a -> IO (Maybe (Matrix y U1))
```

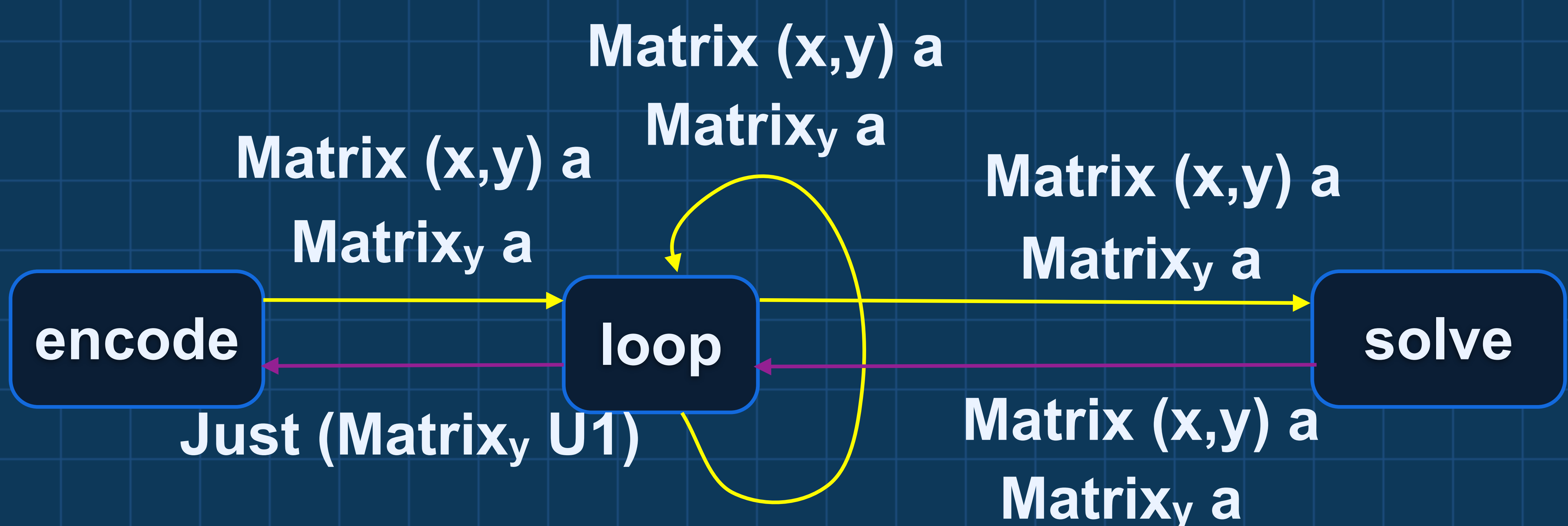


THE UNIVERSITY OF KANSAS

Specification to Architecture (1)

- Fission into driver and execution unit
- Idea is execution unit will become our hardware entity
- Driver can be used to (automatically) generate test vectors.

```
loop :: Matrix y a -> IO (Maybe (Matrix y U1))
```

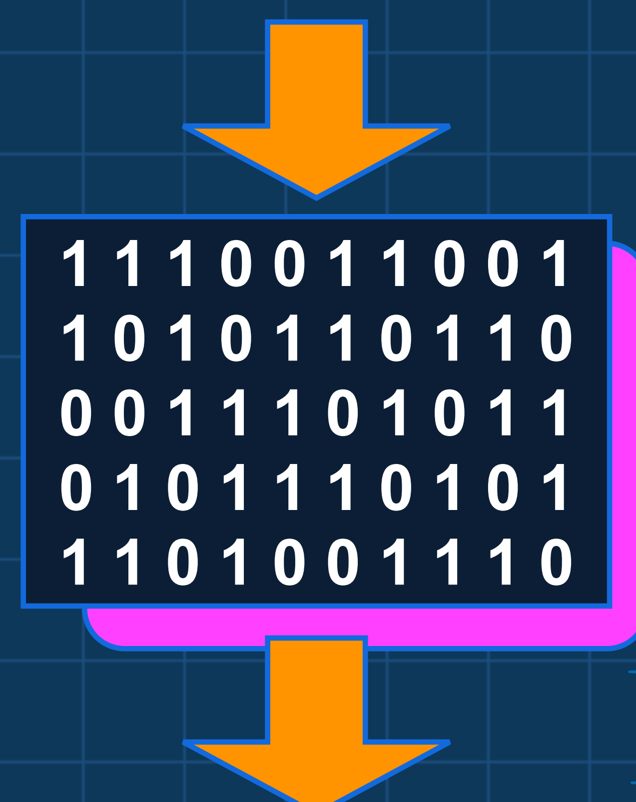



```
solve :: (SM.Matrix (x,y) a, Matrix y a)
      -> (SM.Matrix (x,y) a, Matrix y a)
```

THE UNIVERSITY OF KANSAS

Specification to Architecture (2)

- First, push memory into computation.

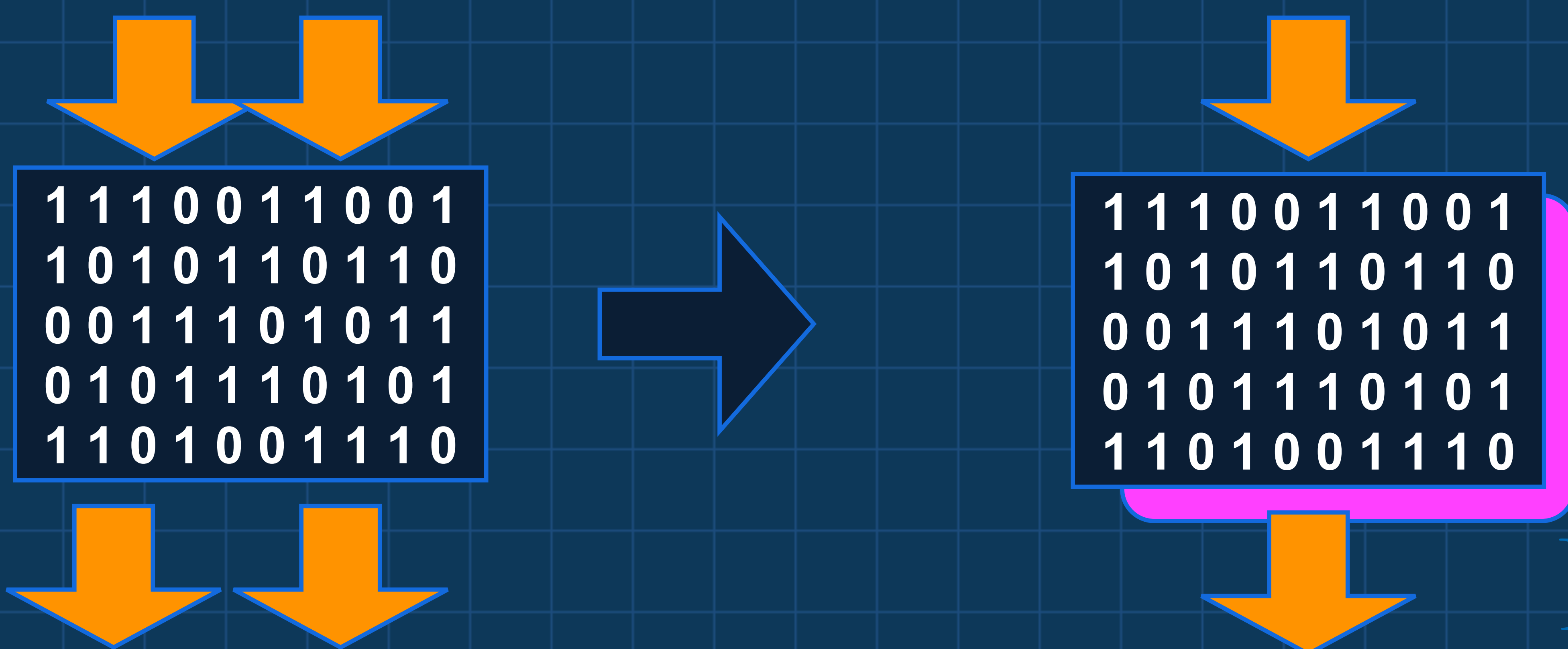


Specification to Architecture (2)

```
solve0 :: (SM.Matrix (x,y) a, Matrix y a)
       -> (SM.Matrix (x,y) a, Matrix y a)
```

- First, push memory into computation.

```
solve1 :: Stream (Matrix y a)
       -> Stream (Matrix y a)
```

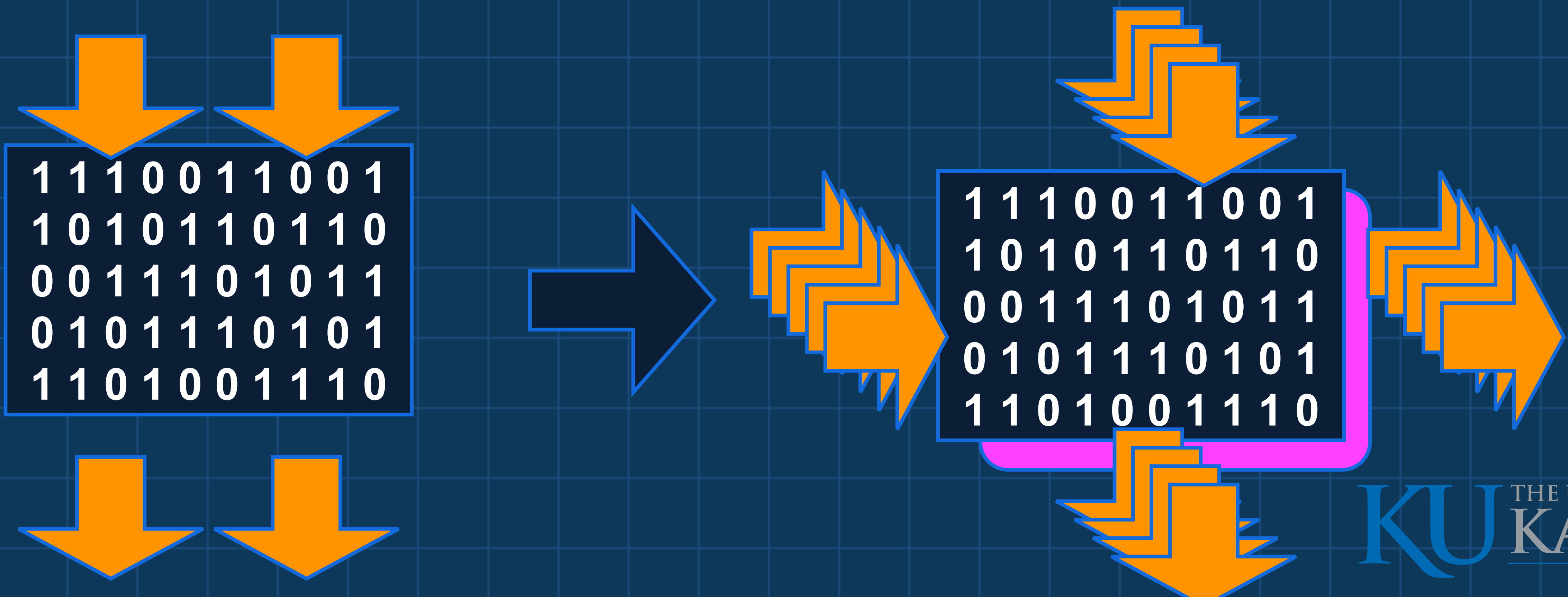


Specification to Architecture (3)

```
solve0 :: (SM.Matrix (x,y) a, Matrix y a)
       -> (SM.Matrix (x,y) a, Matrix y a)
```

- Push memory into computation (introduce Stream)
- Generalize the size of the solution (List of (Stream of) Array)
- Accept and send data to neighbors (extra in and out arguments)

```
:: ([Stream (Array y b)], [Stream (Array x (Maybe b))])
-> ([Stream (Array x (Maybe b))], [Stream (Array y b)])
```

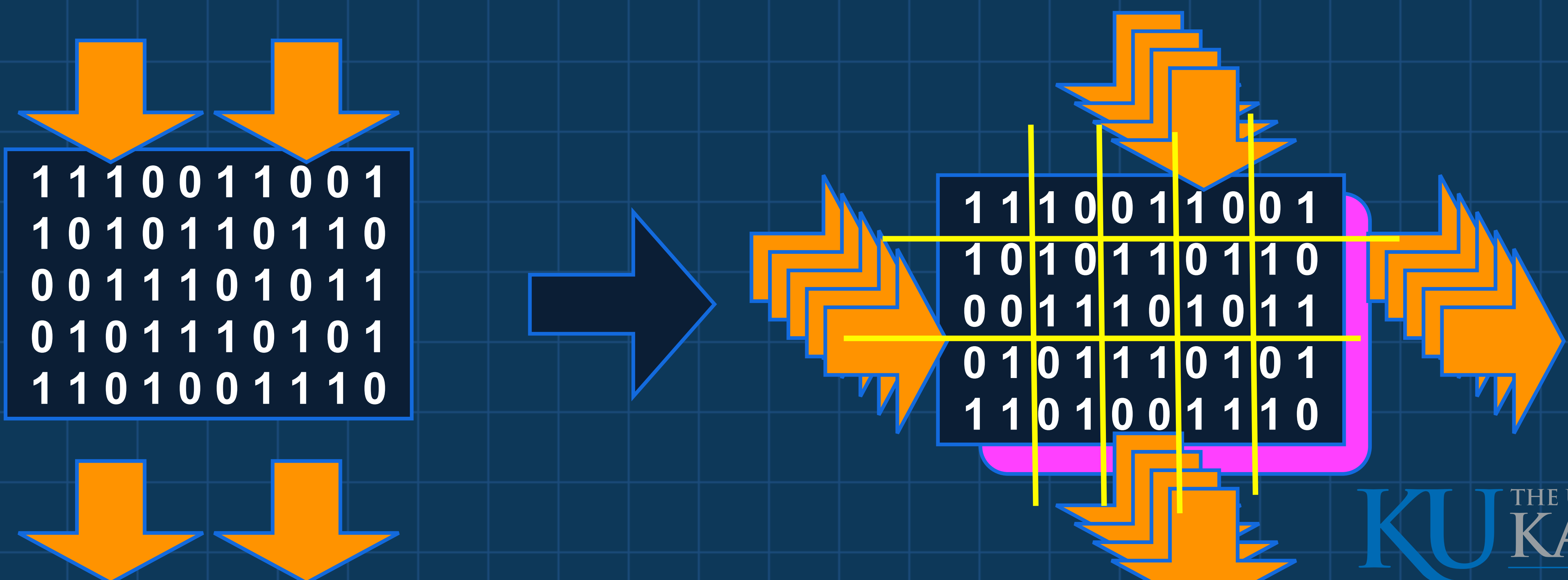


THE UNIVERSITY OF KANSAS

Specification to Architecture (4)

```
:: ([Stream (Array y b)], [Stream (Array x (Maybe b))])
-> ([Stream (Array x (Maybe b))], [Stream (Array y b)])
```

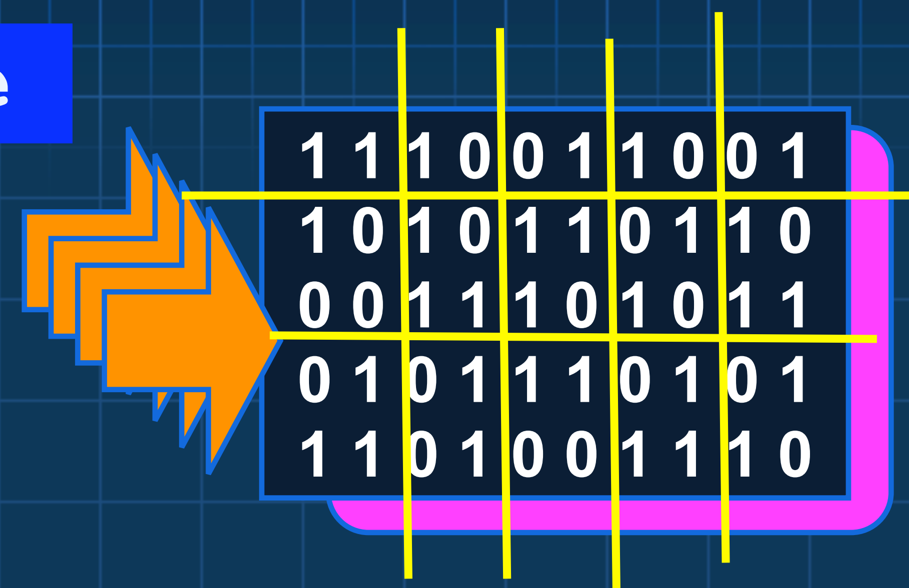
- Divide and conquer in both vertical and horizontal
- Zero “cells” are trivial to handle
- Our chosen “cell” size calls the earlier solution
- We are ready to start the implementation



THE UNIVERSITY OF KANSAS

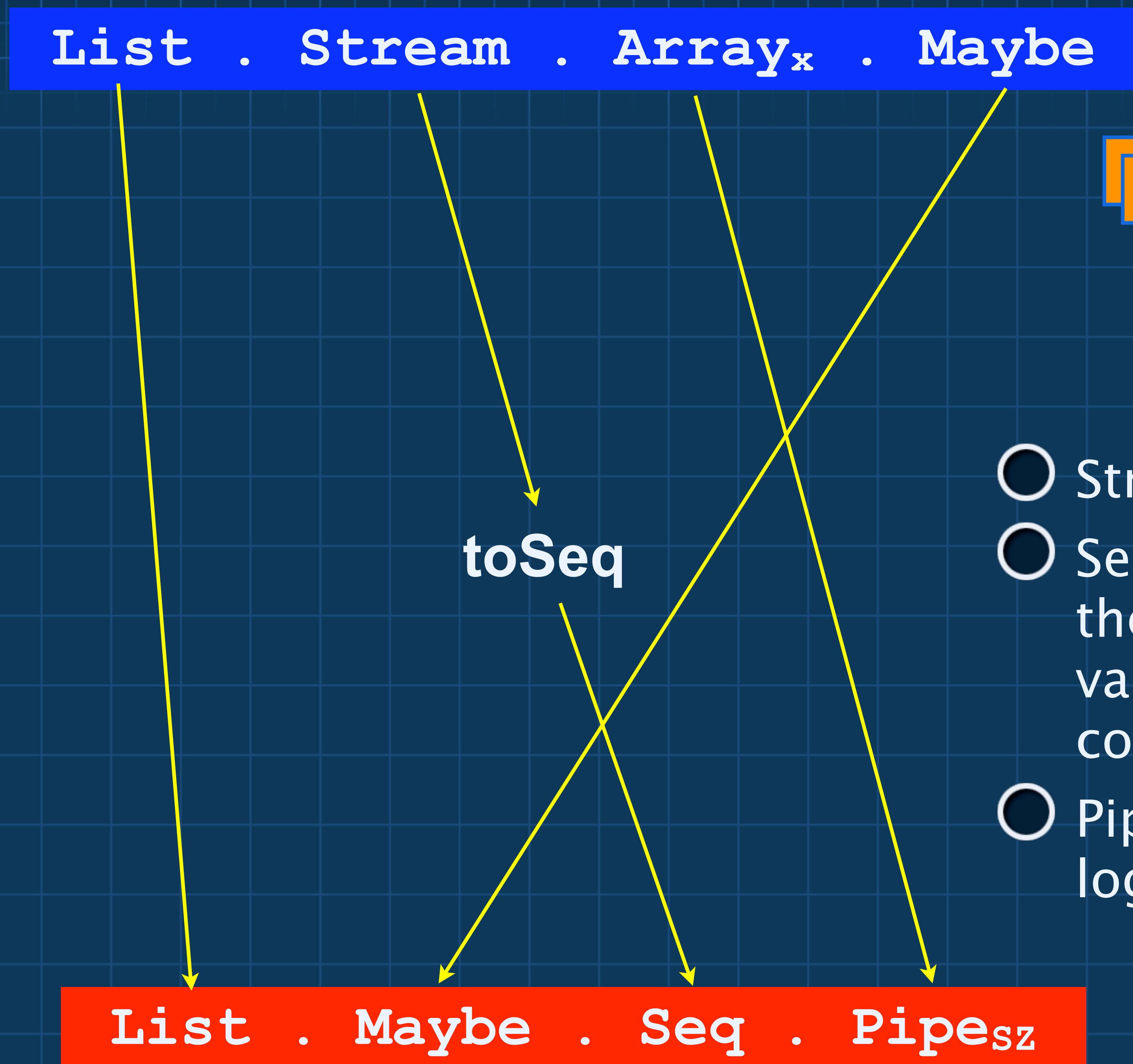
Architecture to Implementation

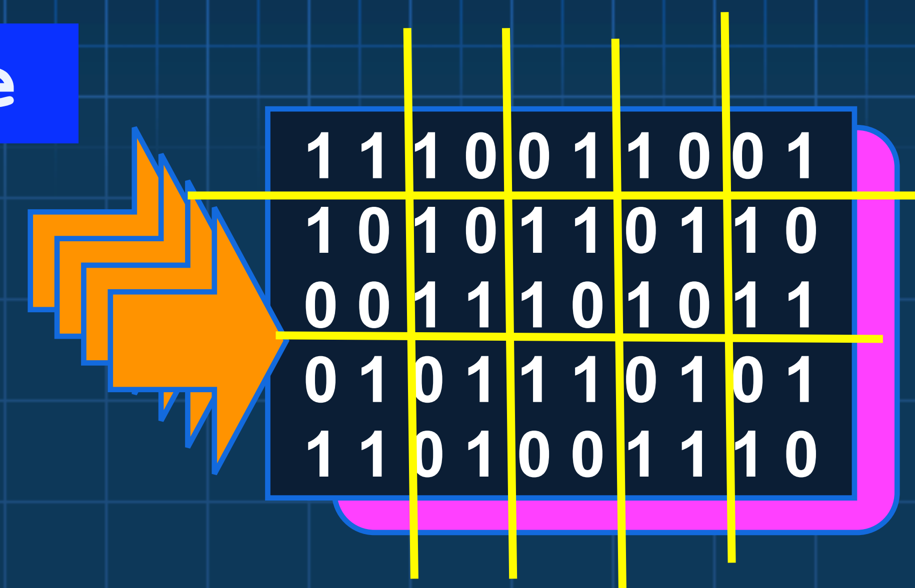
List . Stream . Array$_x$. Maybe



List . Maybe . Seq . Pipe$_{sz}$

Architecture to Implementation





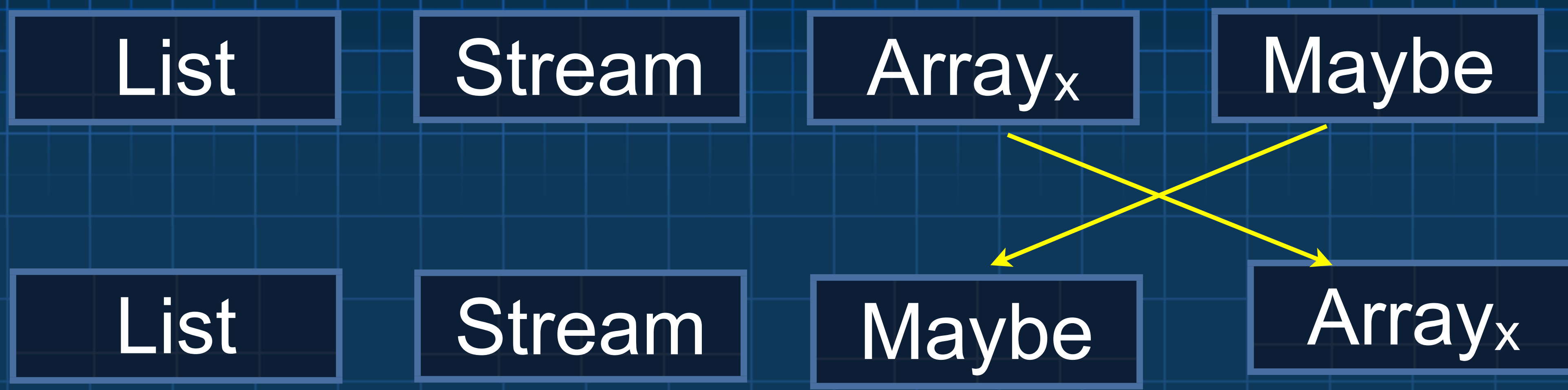
- Streams run every “cycle”.
- Seq always are active, but the meaning of their value depends on some control logic.
- Pipe has its own control logic

Commuting Functors

List | Stream | $Array_x$ | Maybe

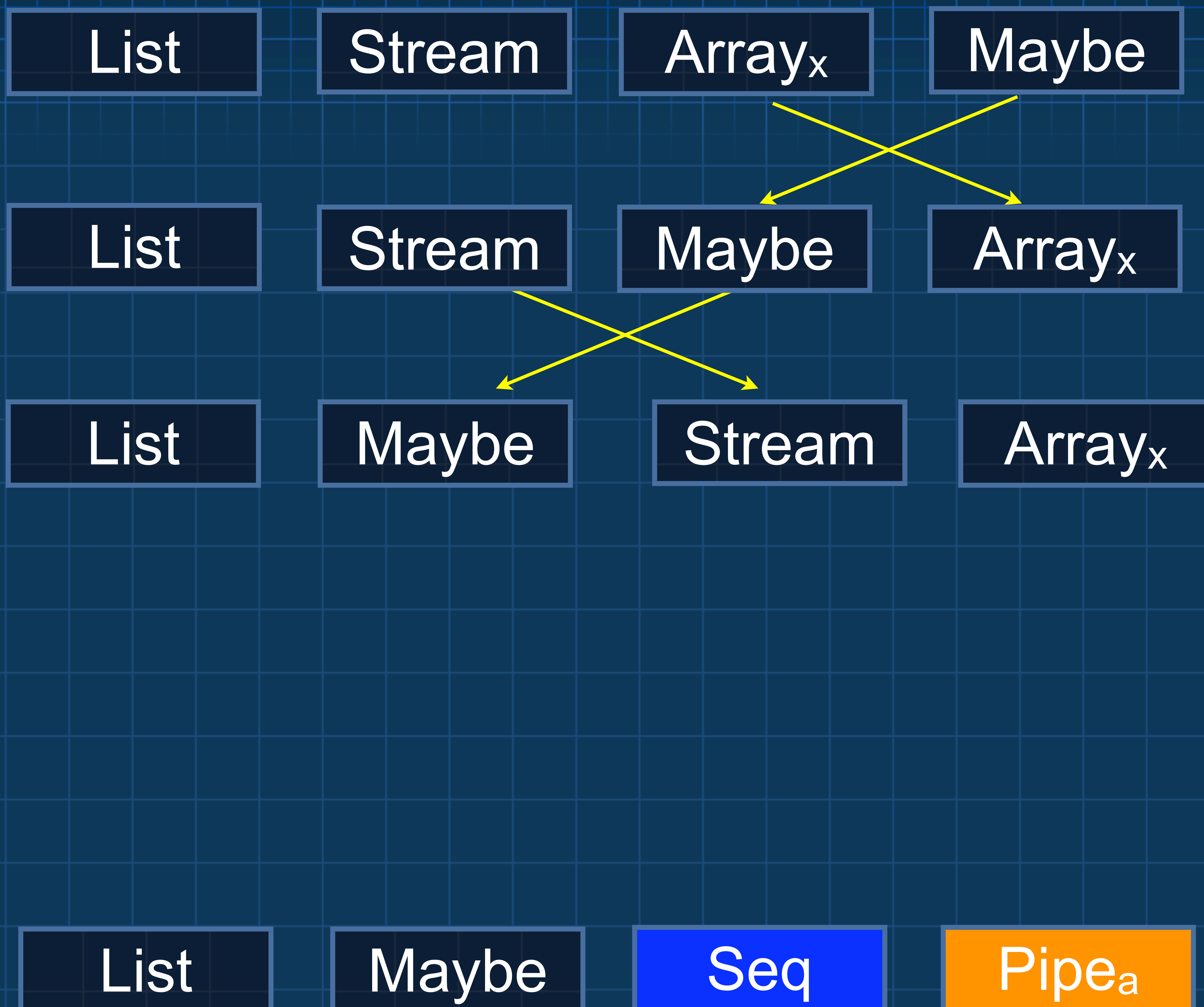
List | Maybe | Seq | $Pipe_a$

Commuting Functors

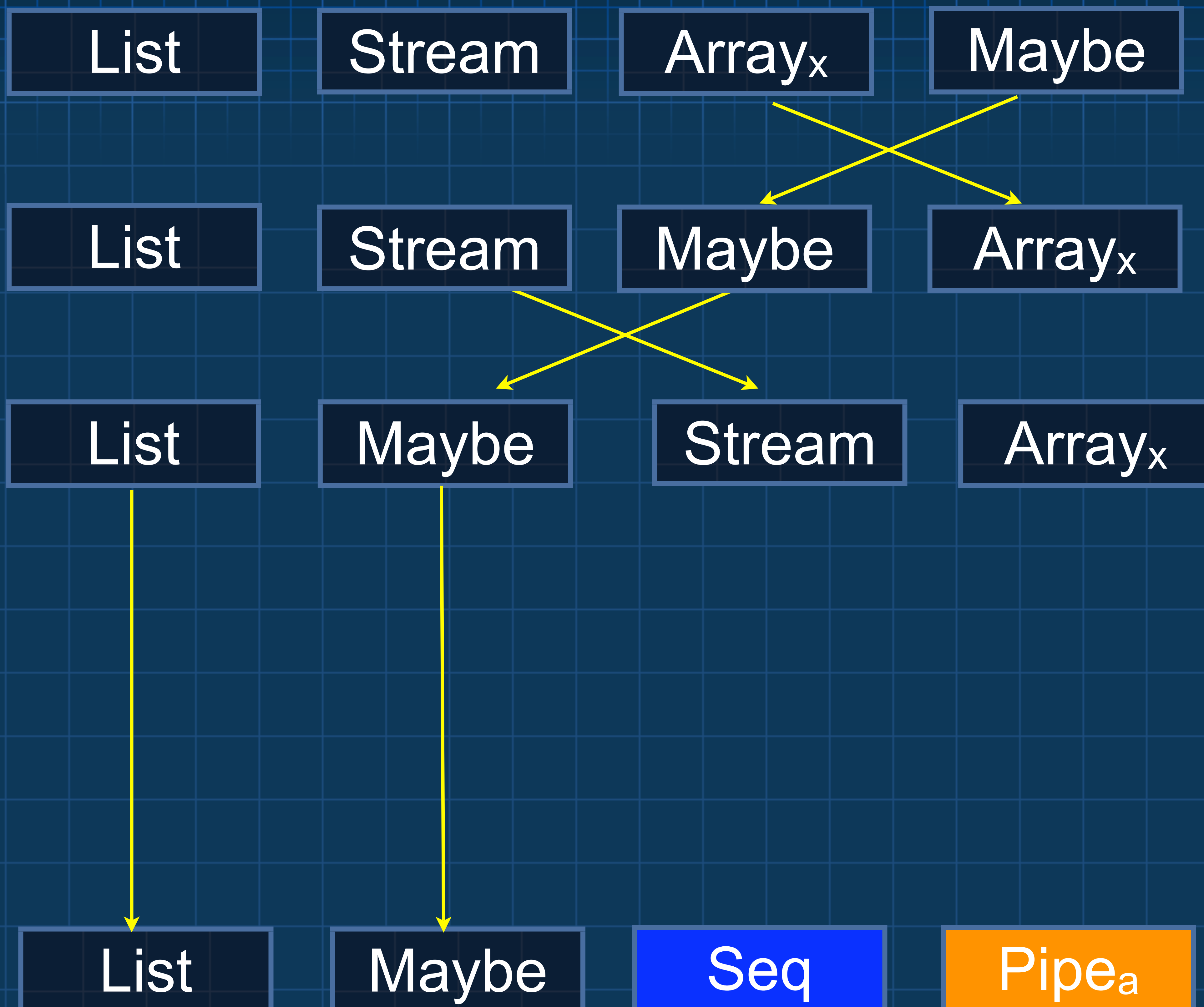


List | Maybe | Seq | Pipe$_a$

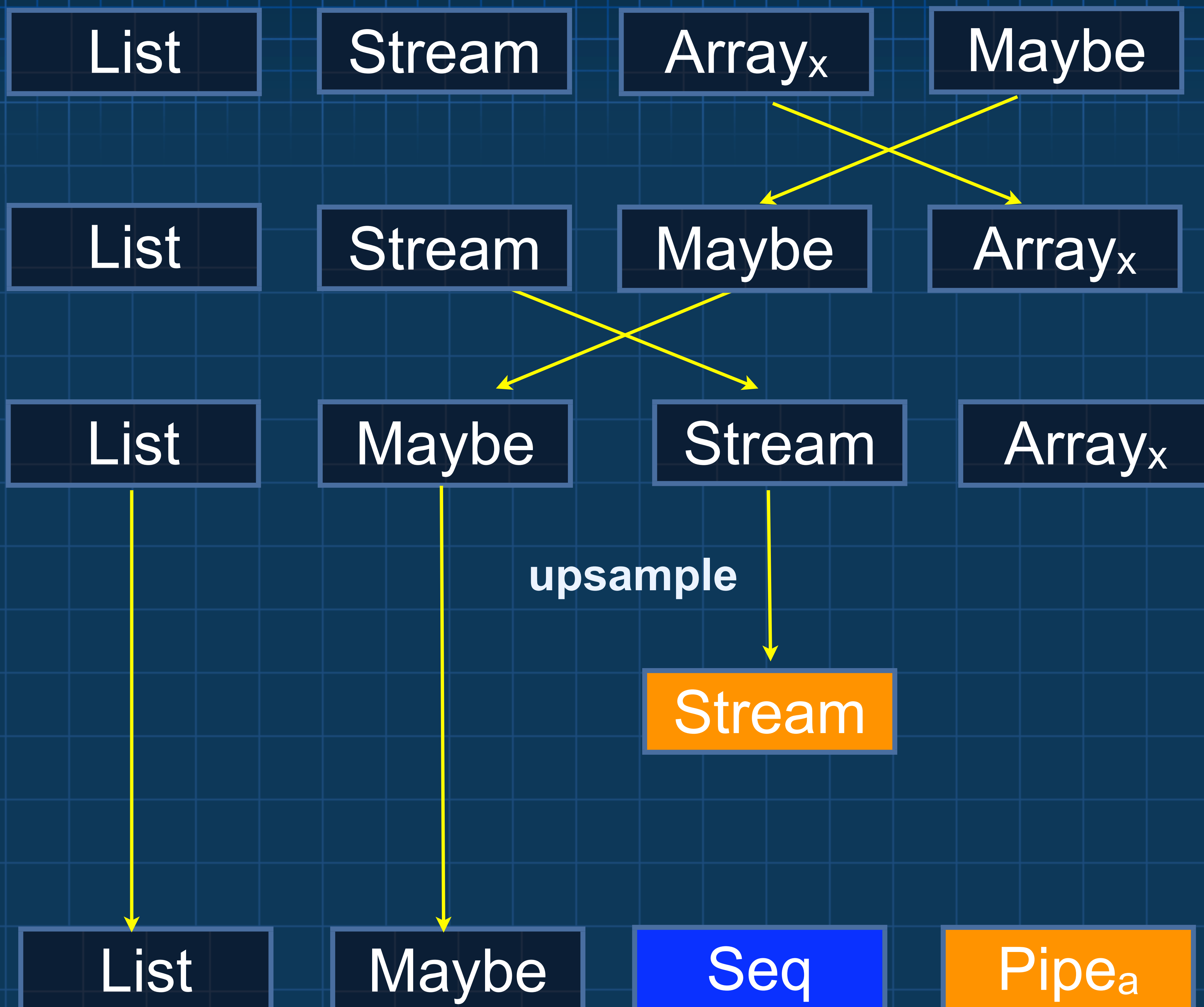
Commuting Functors



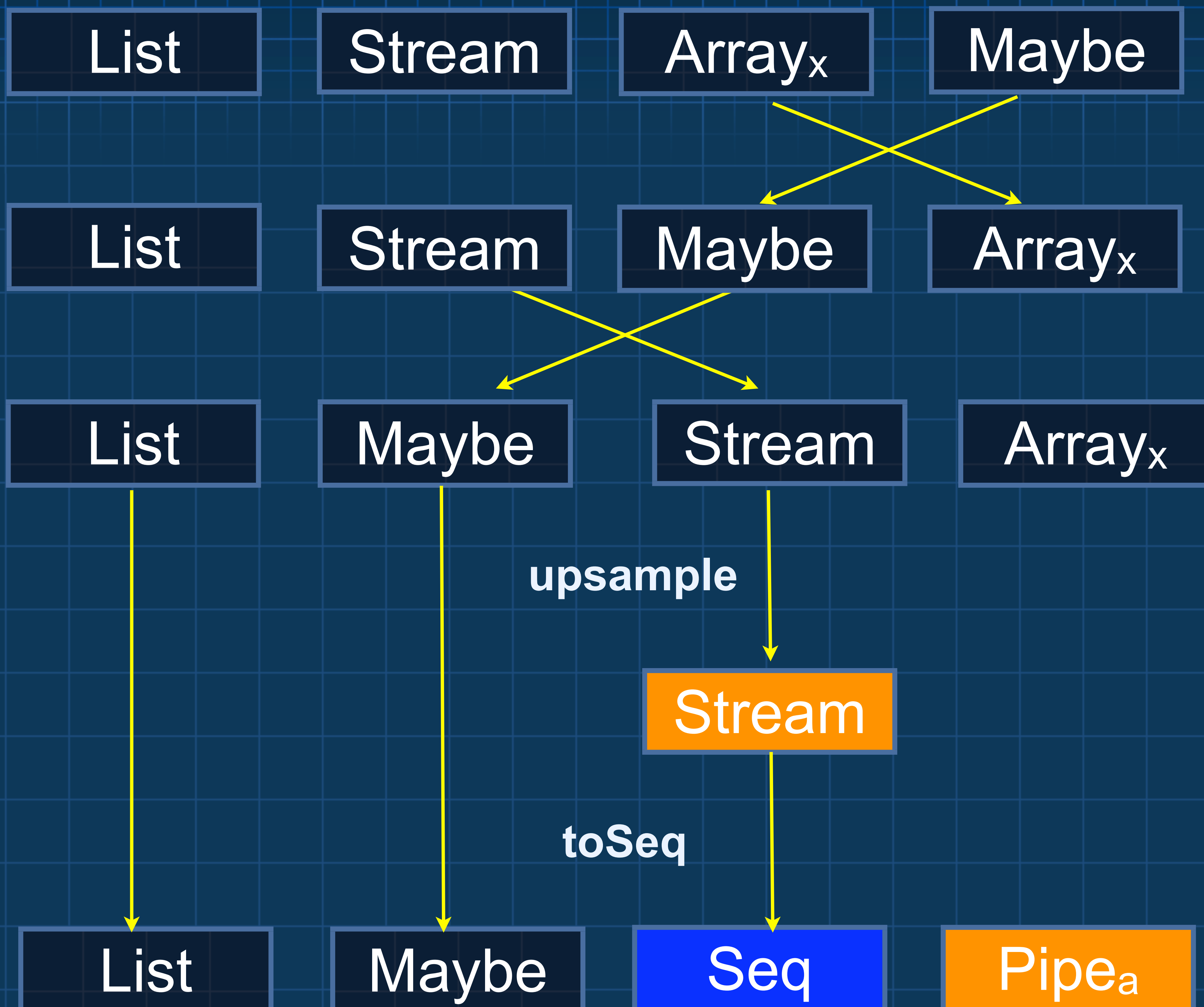
Commuting Functors



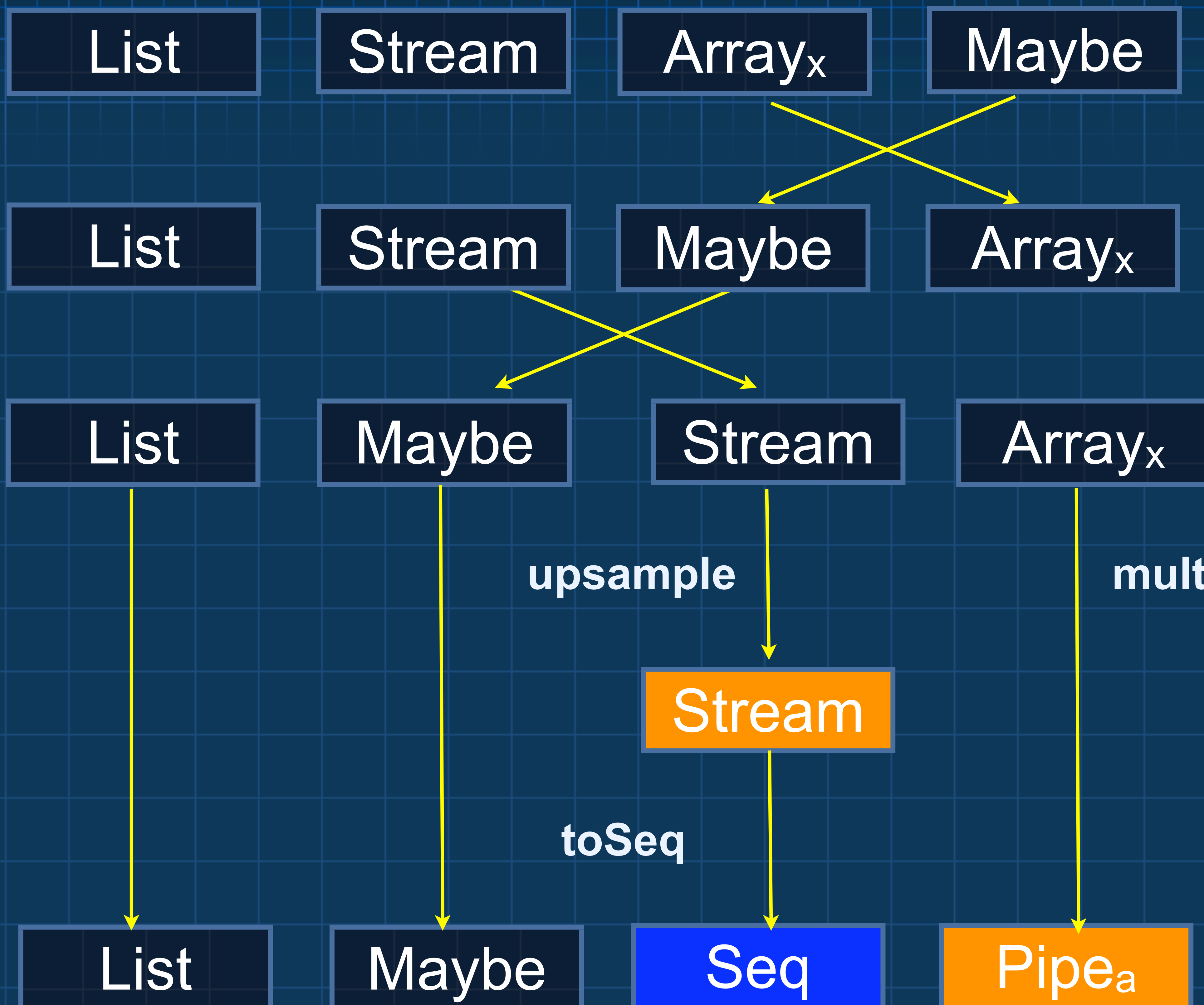
Commuting Functors



Commuting Functors



Commuting Functors



THE UNIVERSITY OF KANSAS

Conclusion and Status

- The methodology of communing functors guided our rewriting towards the types we wanted to use.
  - We used the worker/wrapper transformation to manual rewrite the types of the functions each time.
  - This should be possible to automate.
  - We could focus on the meaning of values under control logic.
- Our model (and mix* model) matched exactly the published bit error rate curves, and implementation came in exactly where expected.
- We have working simulations of the complete LDPC in Modelsim.
- No FPGA version of LDPC (yet)
- We want to make our system more flexible; exploring design decisions.

THE UNIVERSITY OF KANSAS