CovertCraft:FPGA Synthesis of a Covert Channel in Fully Associative Cache with Random Eviction

Dhruv Gupta - BT/CSE/220361

Pragati Agrawal - BT/CSE/220779

Supervisors:

Prof. Mainak Chaudhuri

Prof. Debadatta Mishra

Ms. Yashika Verma (Mentor)

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

Introduction

Introduction

- CovertCraft is a timing-based covert channel on a fully associative cache with random replacement.

Introduction

- CovertCraft is a timing-based covert channel on a fully associative cache with random replacement.
- A sender and a receiver agree on a predefined protocol and exploit the latency difference between a cache hit and a cache miss to communicate covertly.

Introduction

- CovertCraft is a timing-based covert channel on a fully associative cache with random replacement.
- A sender and a receiver agree on a predefined protocol and exploit the latency difference between a cache hit and a cache miss to communicate covertly.
- It can lead to security breach.

Introduction

- CovertCraft is a timing-based covert channel on a fully associative cache with random replacement.
- A sender and a receiver agree on a predefined protocol and exploit the latency difference between a cache hit and a cache miss to communicate covertly.
- It can lead to security breach.
- Such a cache was proposed as a solution against timing-based covert channels.

Overview

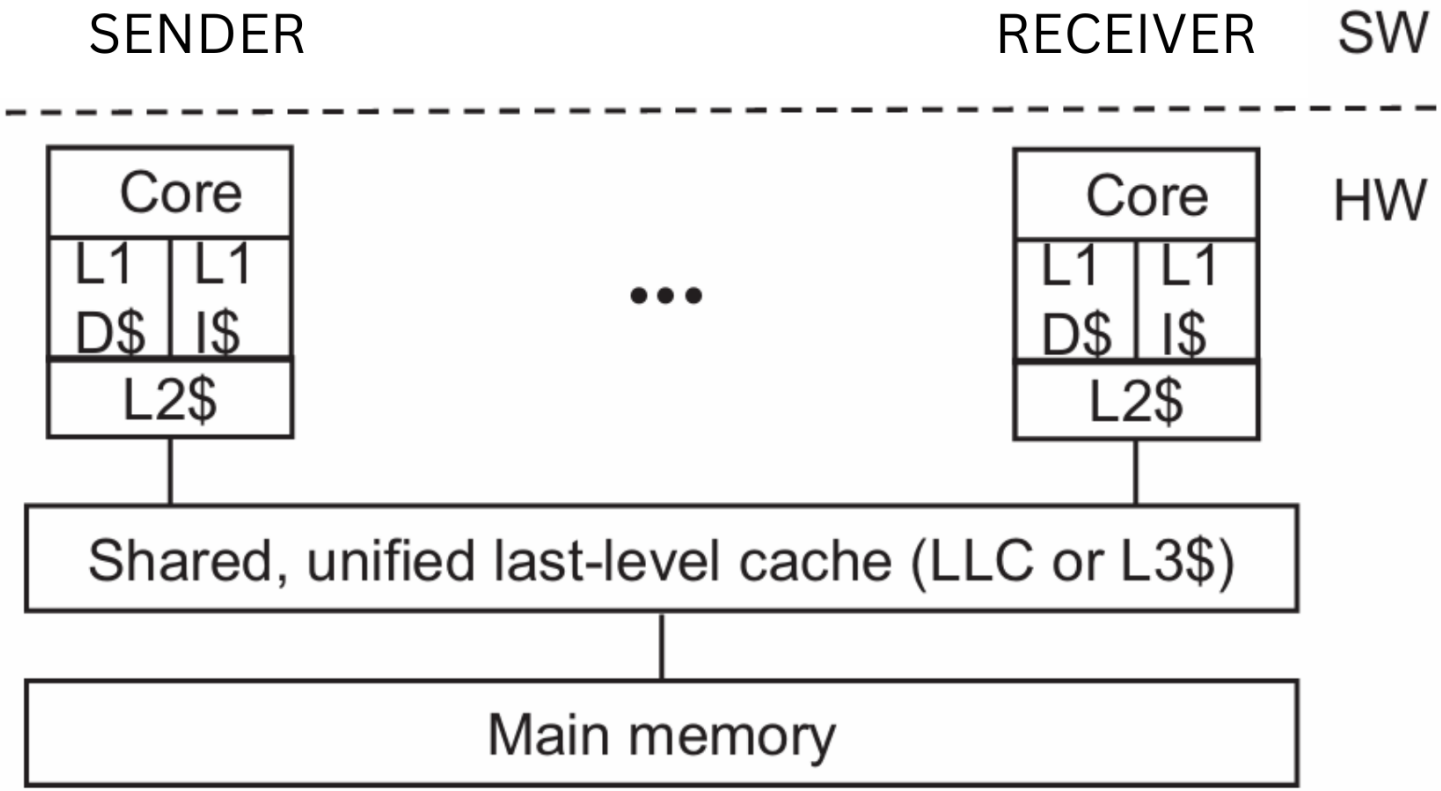


Figure: Covert channel in a real system[1]

[1] Source : https://ieeexplore.ieee.org/document/7163050

Scope of our UGP

- **LeakyRand** provides such a covert channel which is efficient and has a very low bit error rate.

Scope of our UGP

- **LeakyRand** provides such a covert channel which is efficient and has a very low bit error rate.
- **CovertCraft** takes inspiration from this and demonstrates a simpler version of it on a Spartan 3E FPGA board.

Scope of our UGP

- **LeakyRand** provides such a covert channel which is efficient and has a very low bit error rate.
- **CovertCraft** takes inspiration from this and demonstrates a simpler version of it on a Spartan 3E FPGA board.
- We synthesize a small 16-entry fully associative cache with random replacement policy and a sender and a receiver on the board.

Scope of our UGP

- **LeakyRand** provides such a covert channel which is efficient and has a very low bit error rate.
- **CovertCraft** takes inspiration from this and demonstrates a simpler version of it on a Spartan 3E FPGA board.
- We synthesize a small 16-entry fully associative cache with random replacement policy and a sender and a receiver on the board.
- We use this to demonstrate the working of **LeakyRand** on hardware.

Design

Design: Verilog

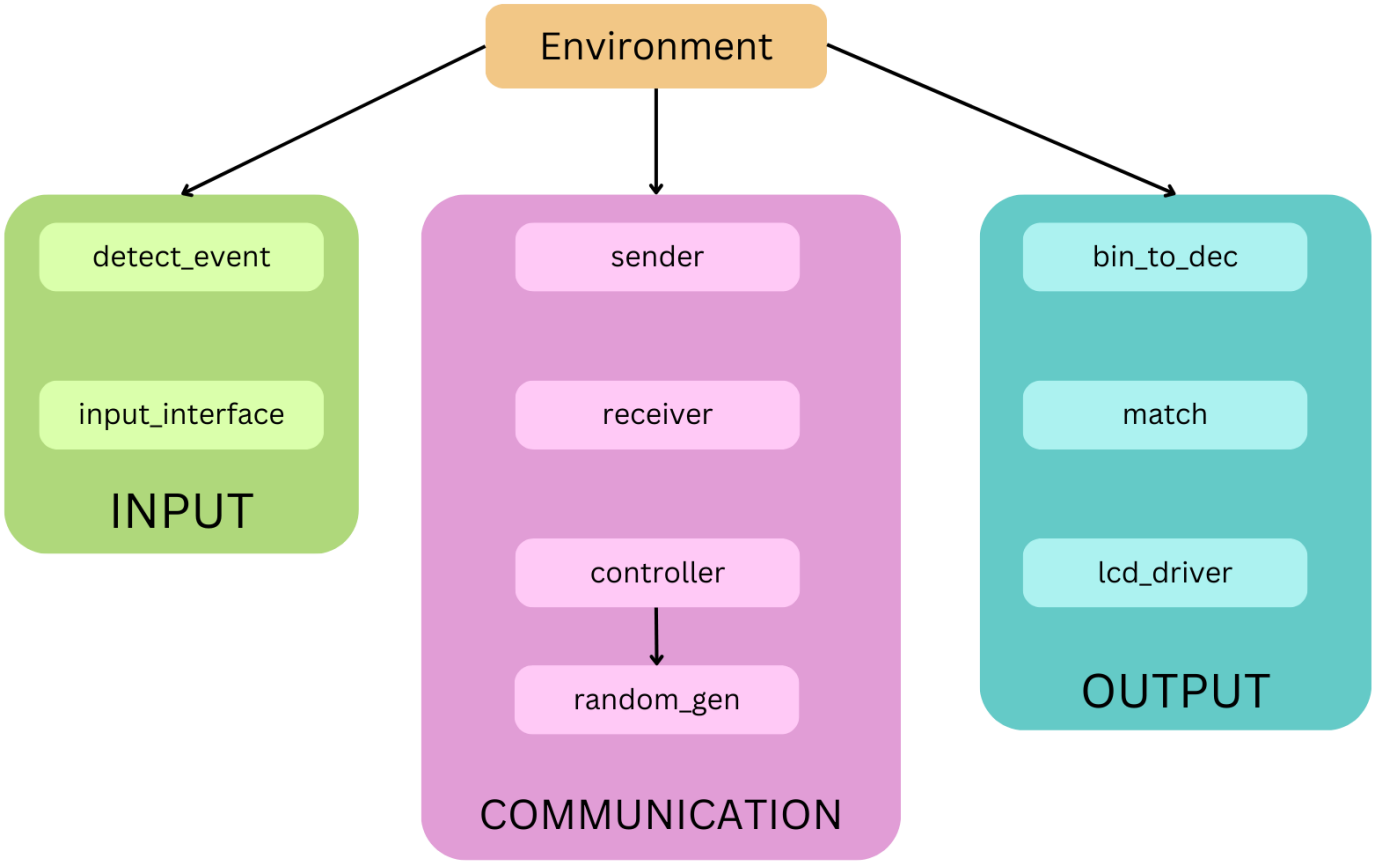


Figure: Classification of modules

Design:Environment

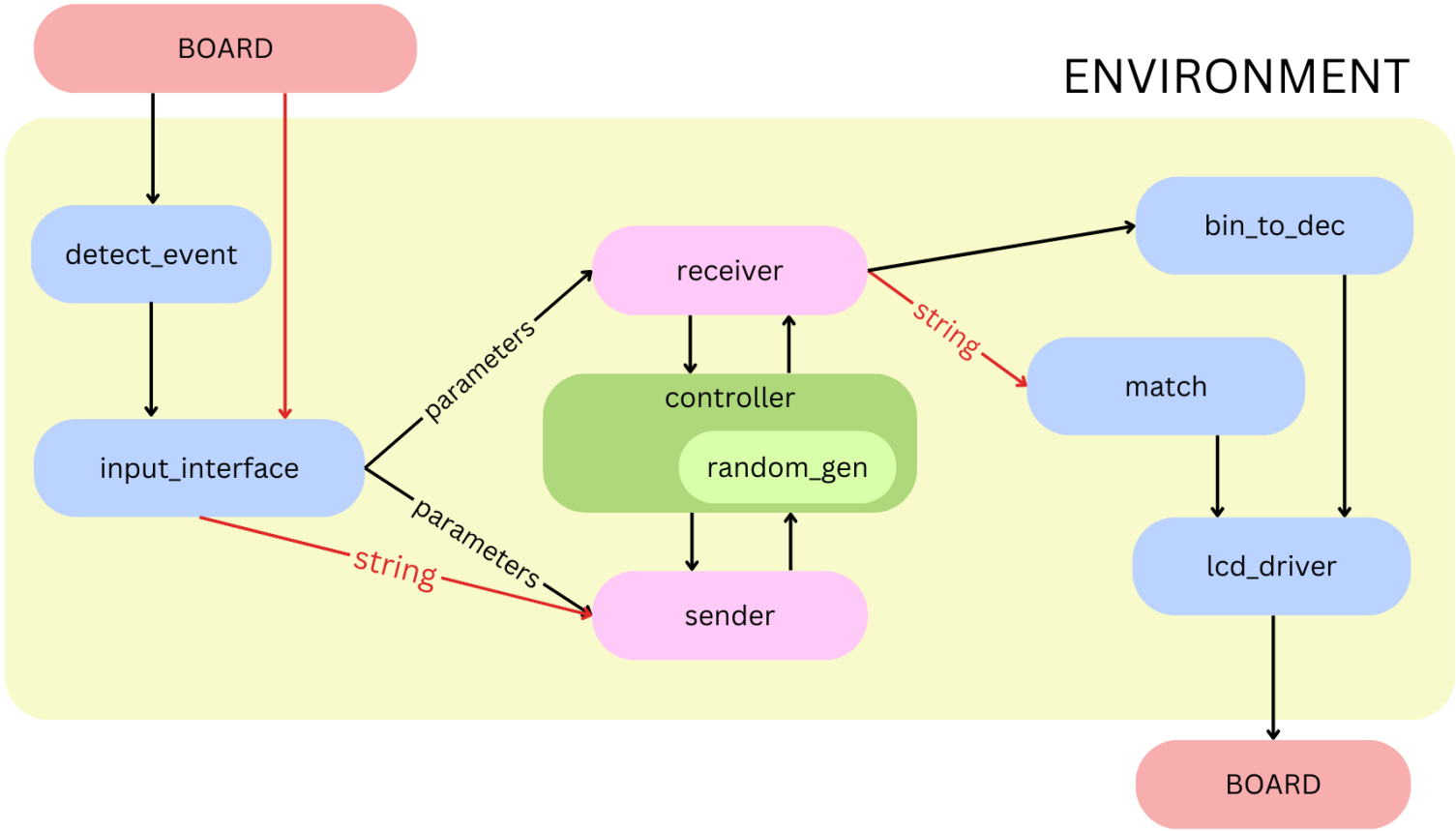


Figure: Flow of control and data between modules

Design:Controller

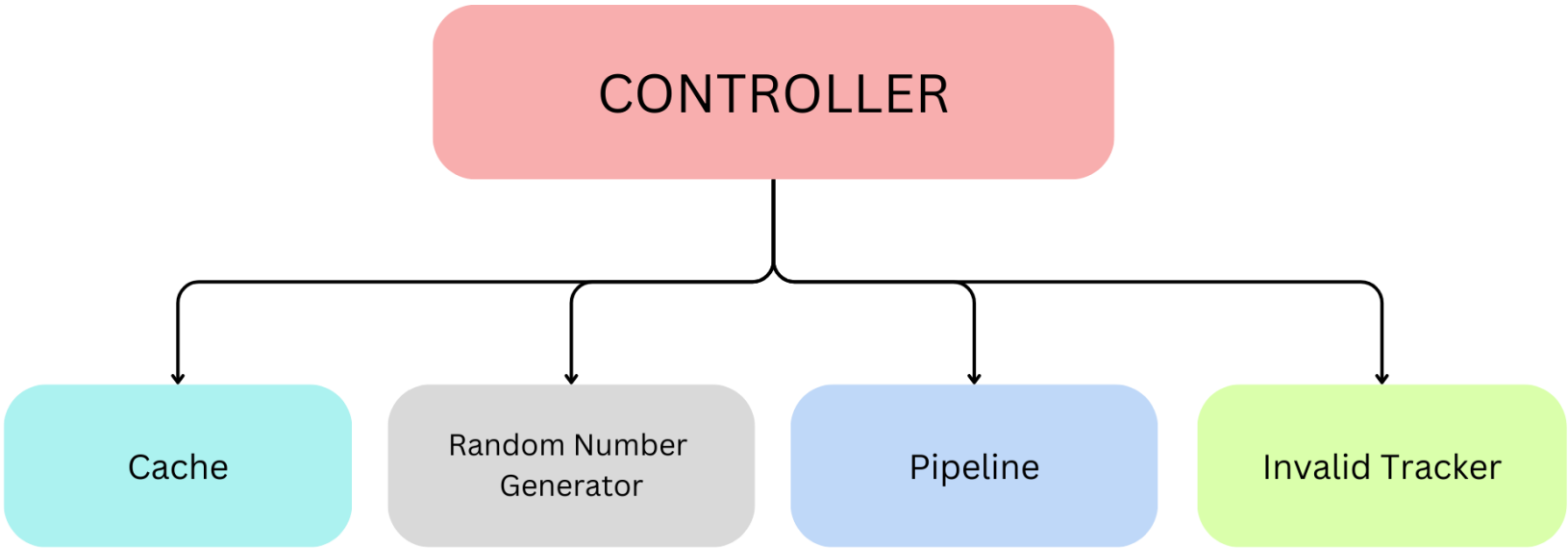


Figure: Parts of the Controller

Design:Controller

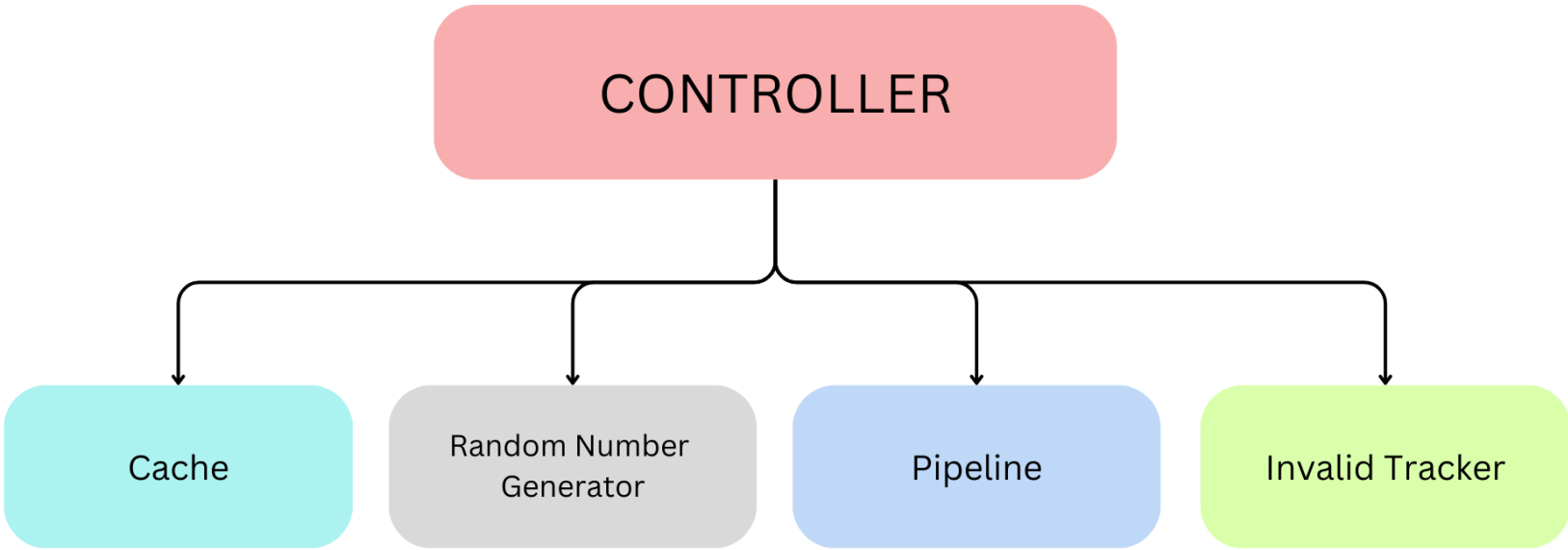


Figure: Parts of the Controller

- We used an LFSR based pseudo-random generator to have random replacement policy in the cache.

Design:Controller

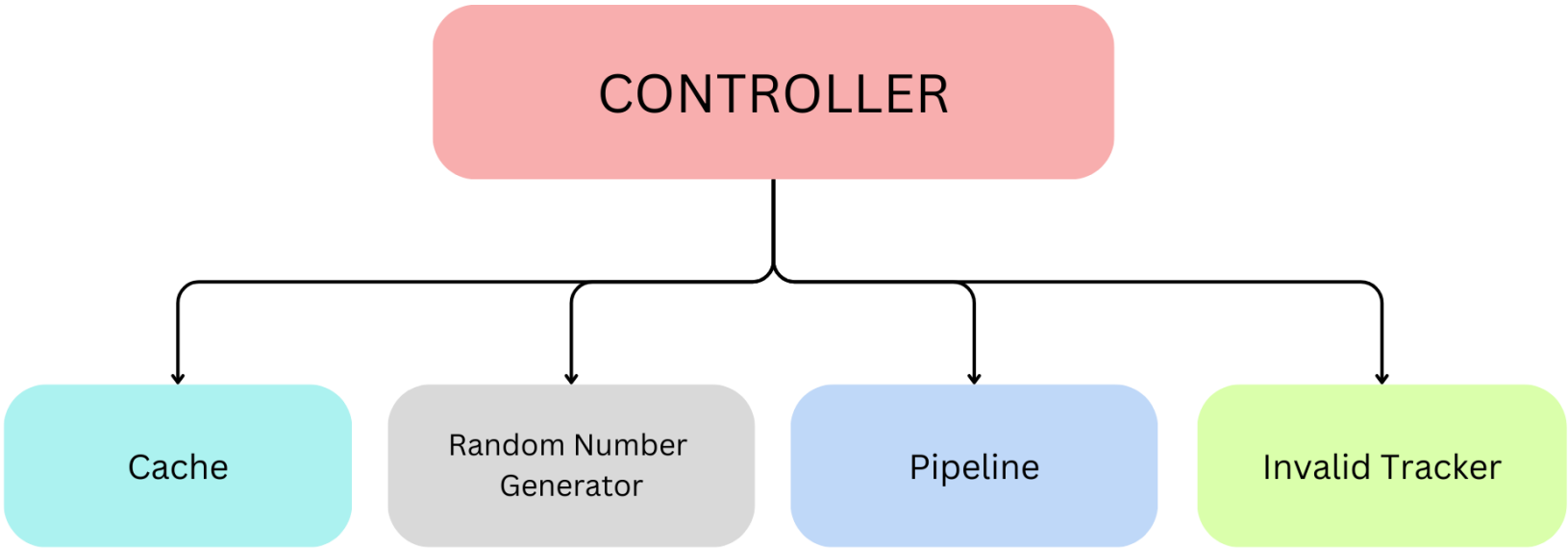


Figure: Parts of the Controller

- We used an LFSR based pseudo-random generator to have random replacement policy in the cache.
- The controller has a 2 stage blocking pipeline with a **4 cycle** miss latency and **2 cycle** hit latency.

Design:Controller

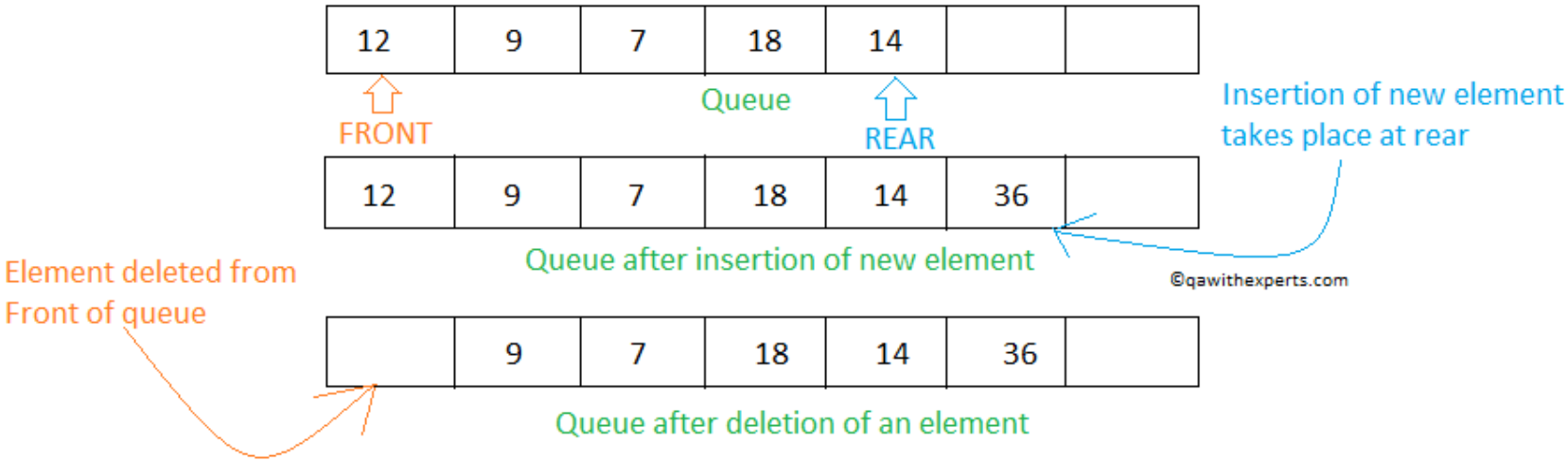


Figure: Implementation of invalid tracker

- Invalid tracker is a FIFO queue to keep track of invalid blocks in the cache.

Design:Receiver

Design:Receiver

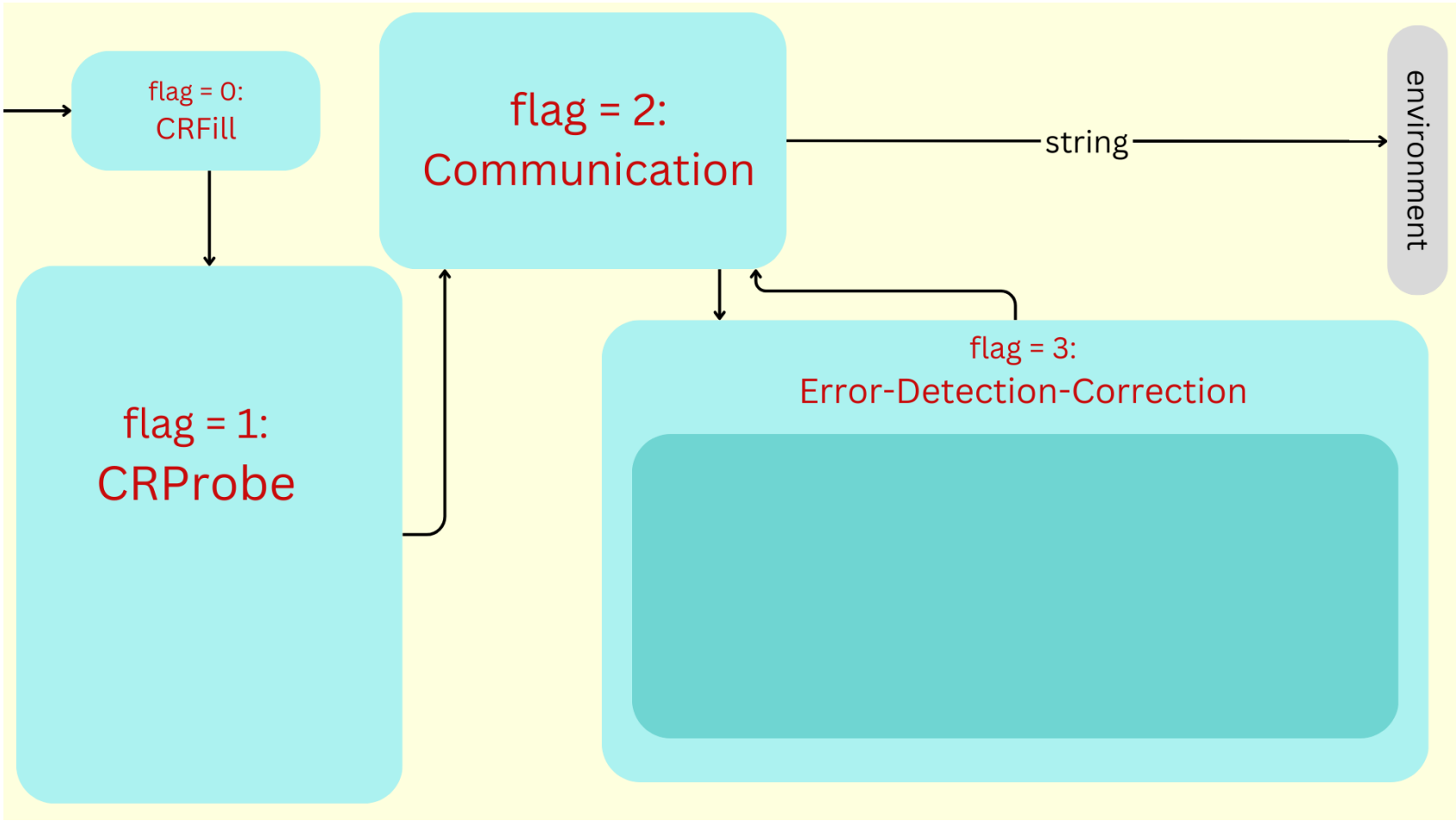


Figure: Receiver FSM

Receiver: CRFill step

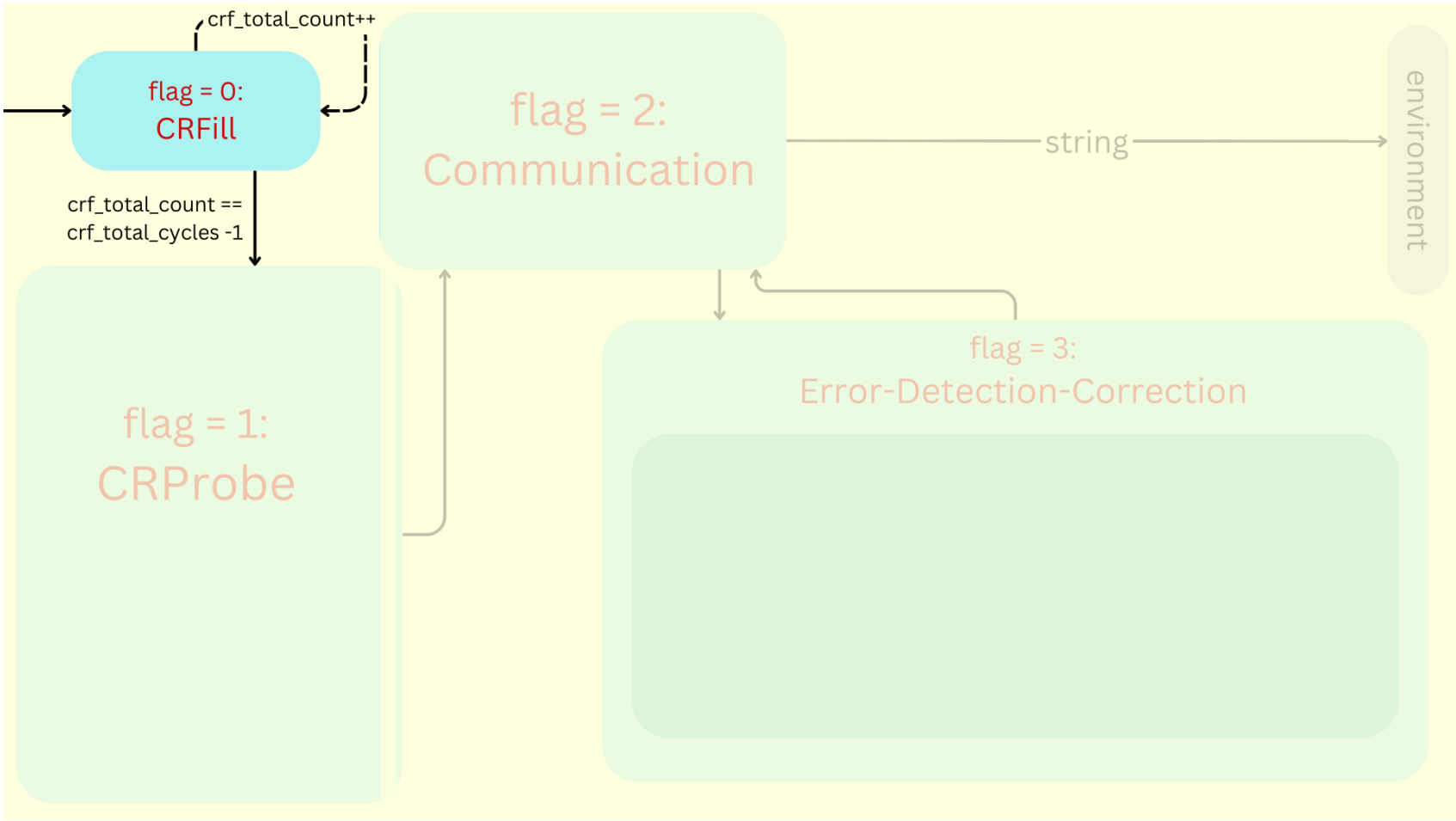
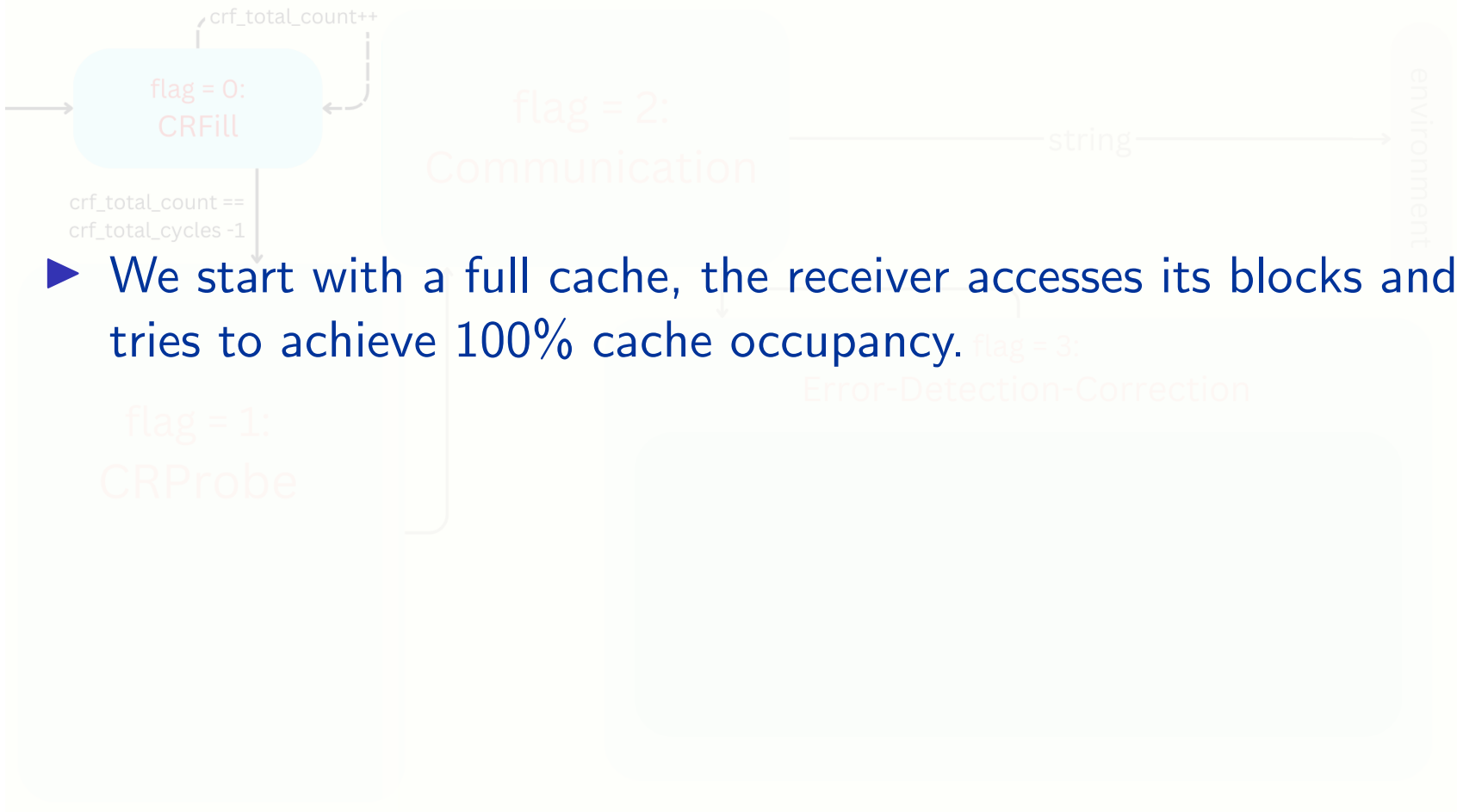


Figure: CRFill step

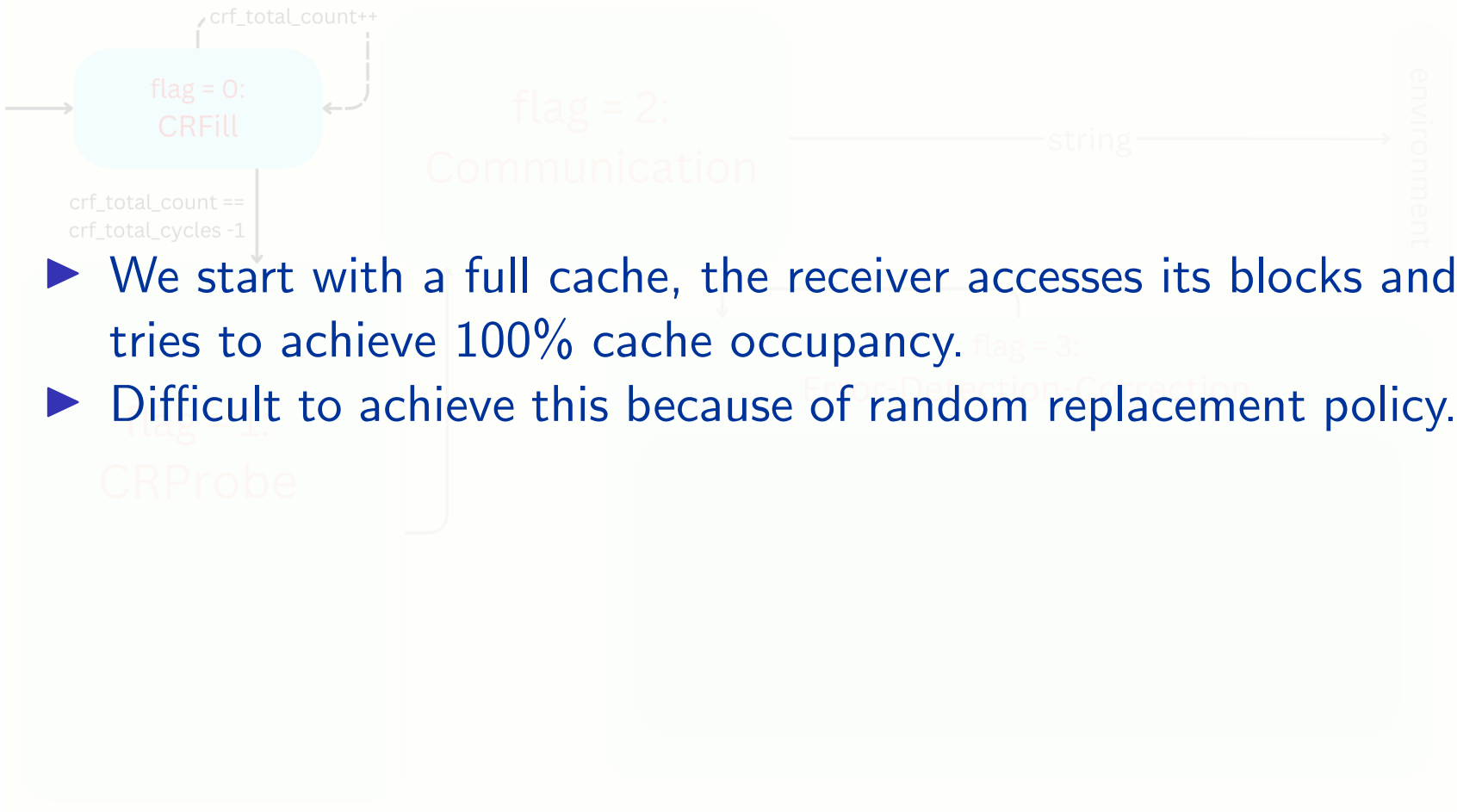
Receiver: CRFill step

Receiver: CRFill step



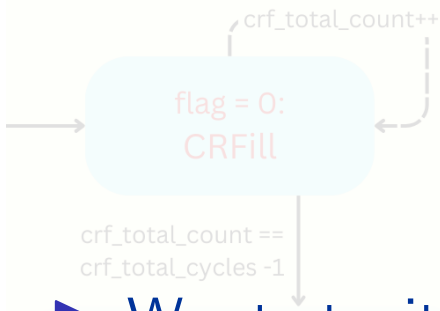
- We start with a full cache, the receiver accesses its blocks and tries to achieve 100% cache occupancy.

Receiver: CRFill step



- We start with a full cache, the receiver accesses its blocks and tries to achieve 100% cache occupancy.
- Difficult to achieve this because of random replacement policy.

Receiver: CRFill step



- We start with a full cache, the receiver accesses its blocks and tries to achieve 100% cache occupancy.
- Difficult to achieve this because of random replacement policy.
- Optimised *Occupancy Sequence* is used for this step, for example $(1x)(1.1x)^2(1.2x)^3$.

Receiver: CRFill step

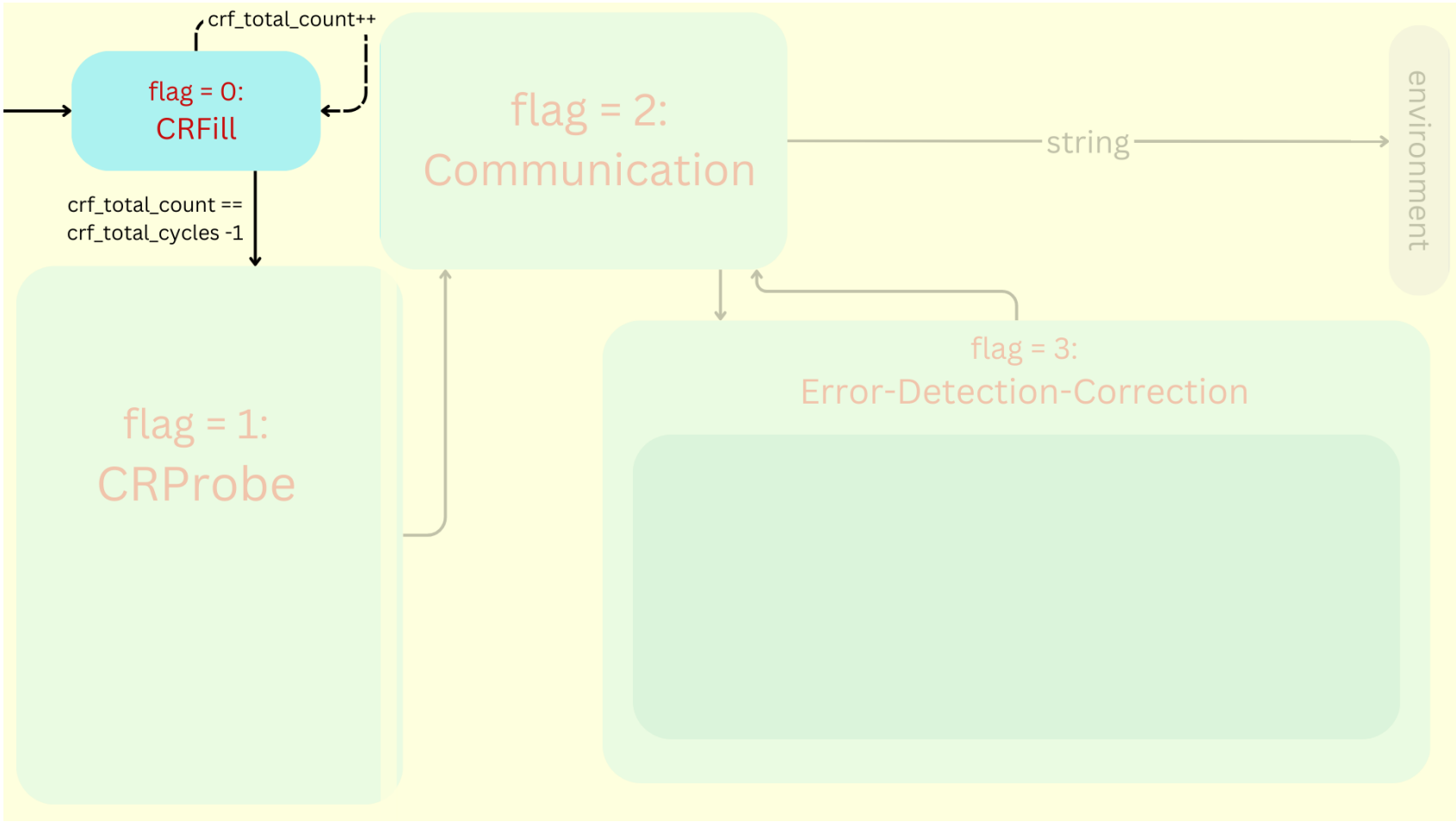


Figure: CRFill step

Receiver: CRProbe step

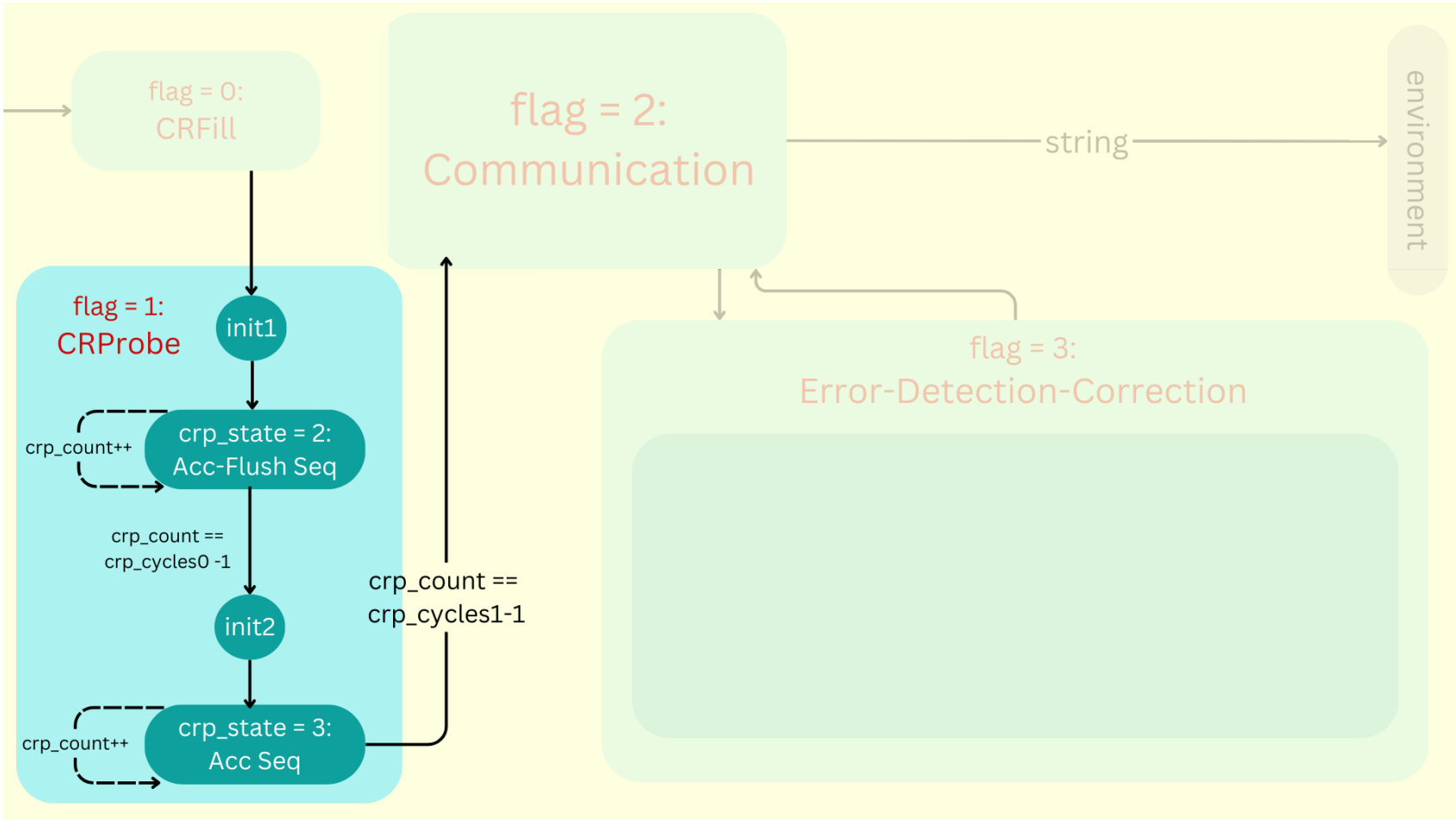
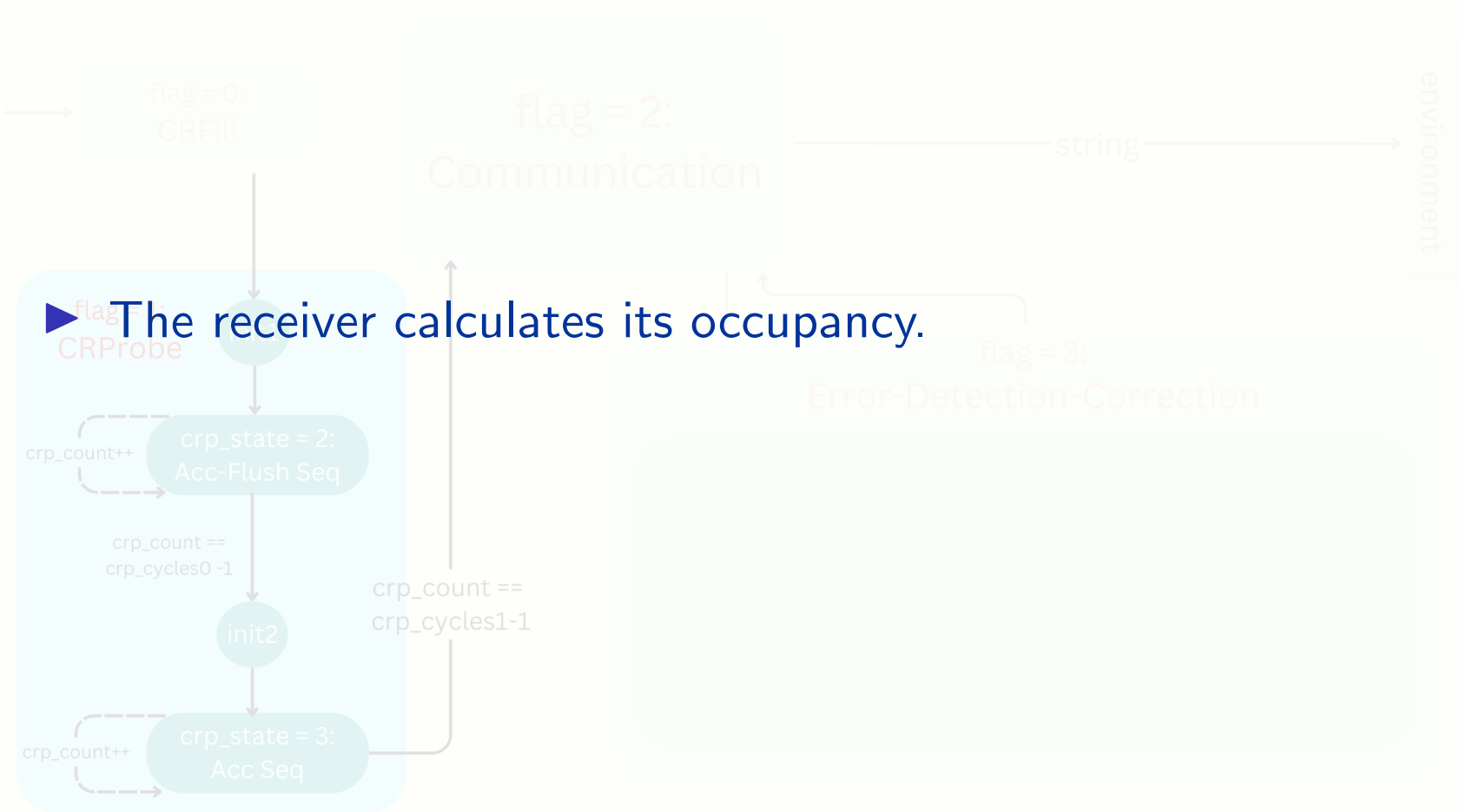


Figure: CRProbe step

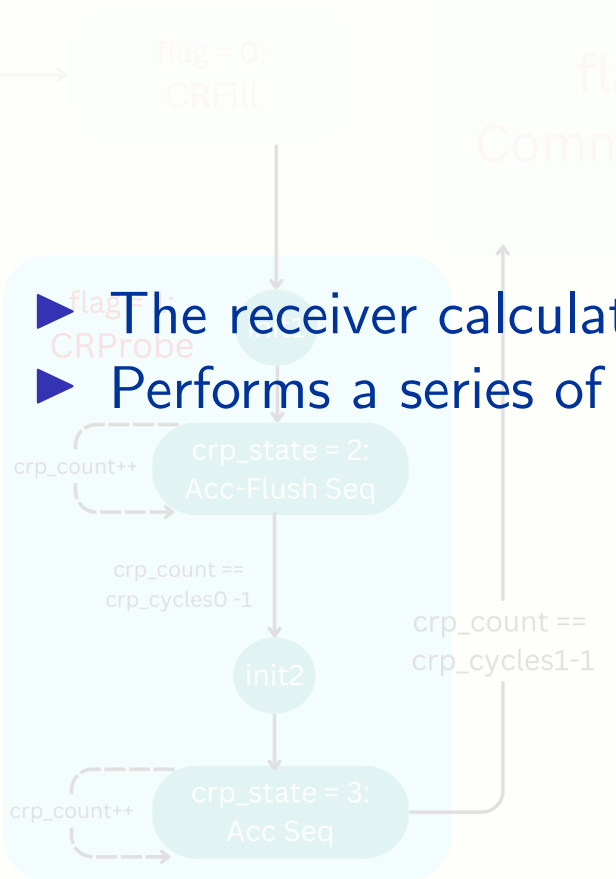
Receiver: CRProbe step

Receiver: CRProbe step



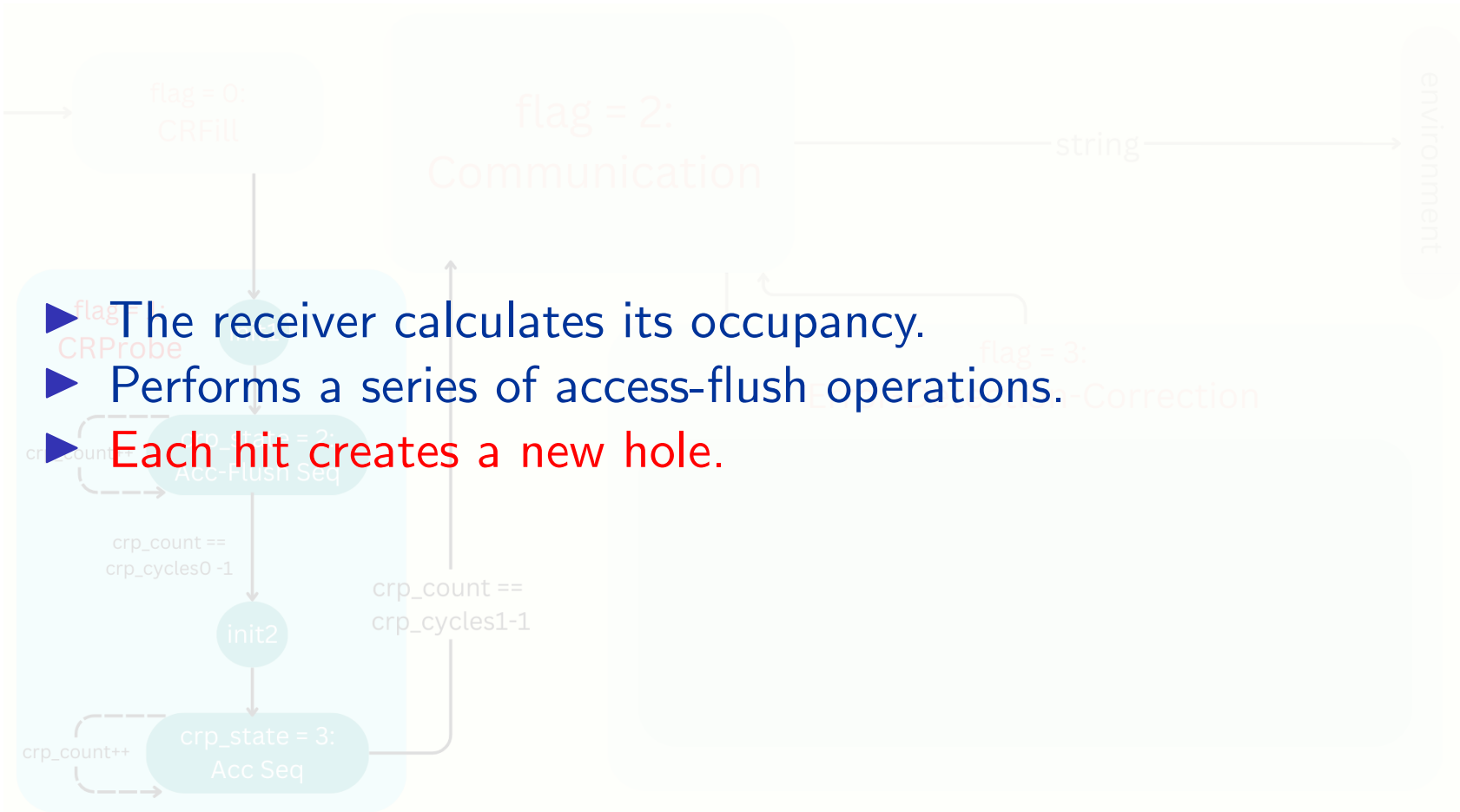
- The receiver calculates its occupancy.

Receiver: CRProbe step



- The receiver calculates its occupancy.
- Performs a series of access-flush operations.

Receiver: CRProbe step



- The receiver calculates its occupancy.
- Performs a series of access-flush operations.
- Each hit creates a new hole.

Receiver: CRProbe step

- The receiver calculates its occupancy.
- Performs a series of access-flush operations.
- Each hit creates a new hole.
- Fills all the holes.

Receiver: CRProbe step

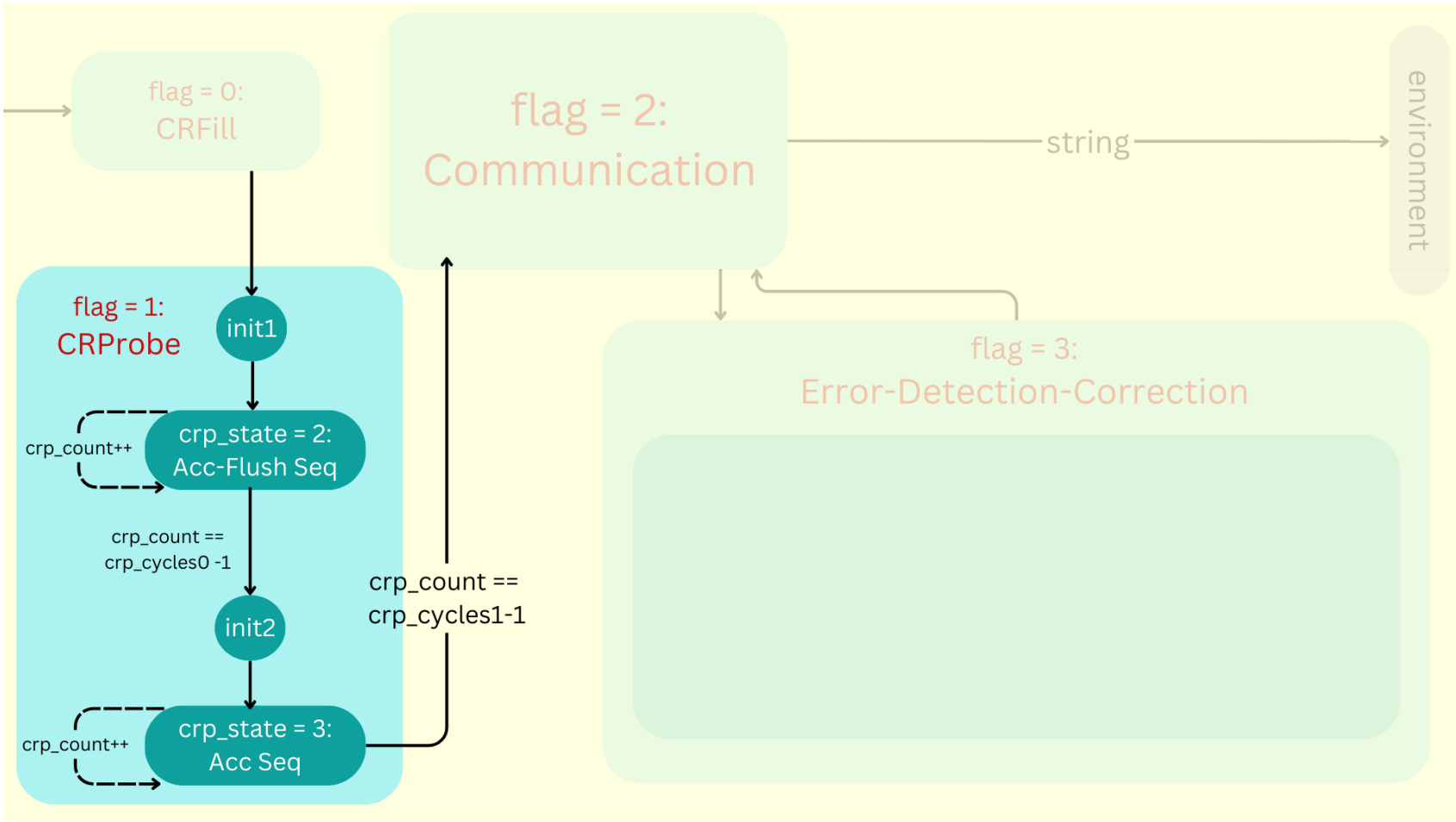


Figure: CRProbe step

Design:Sender

Design:Sender

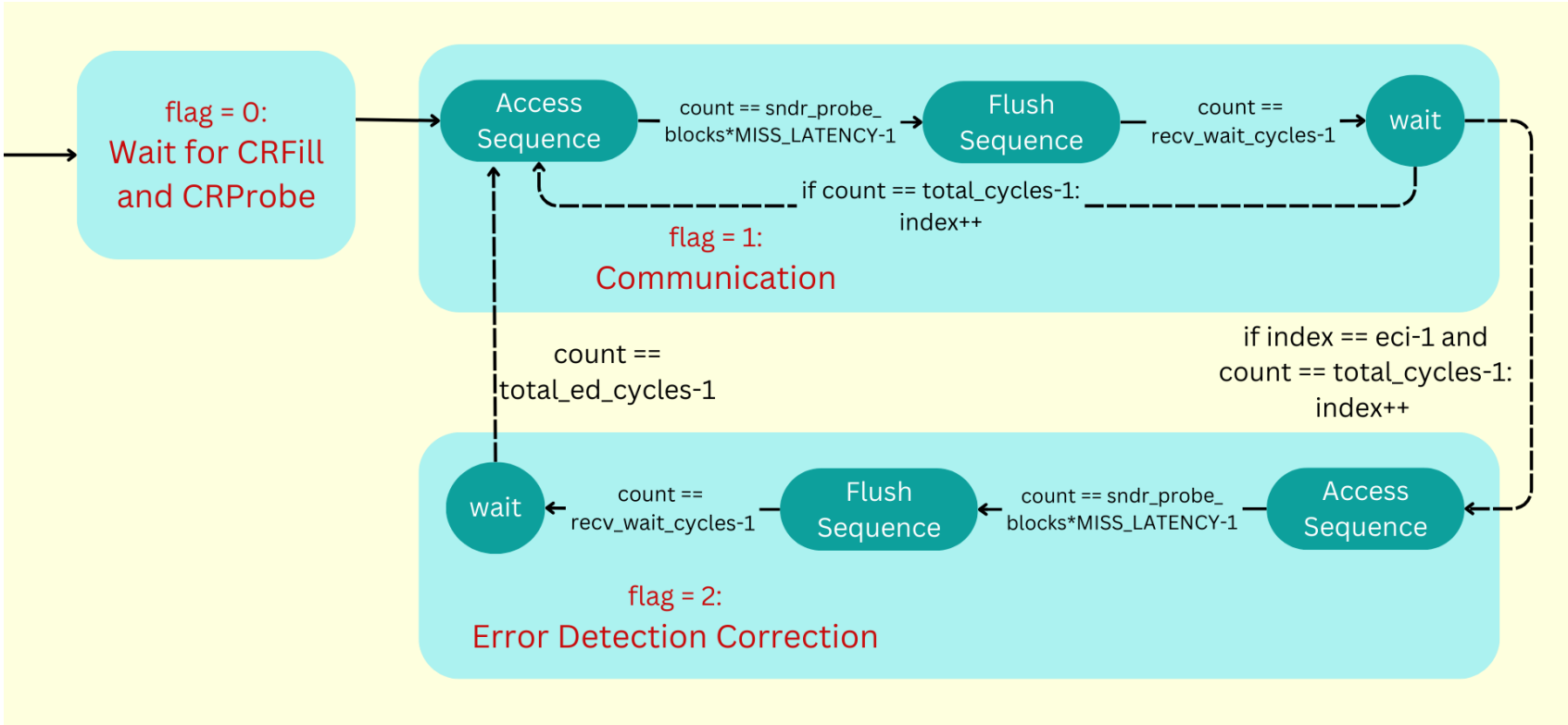
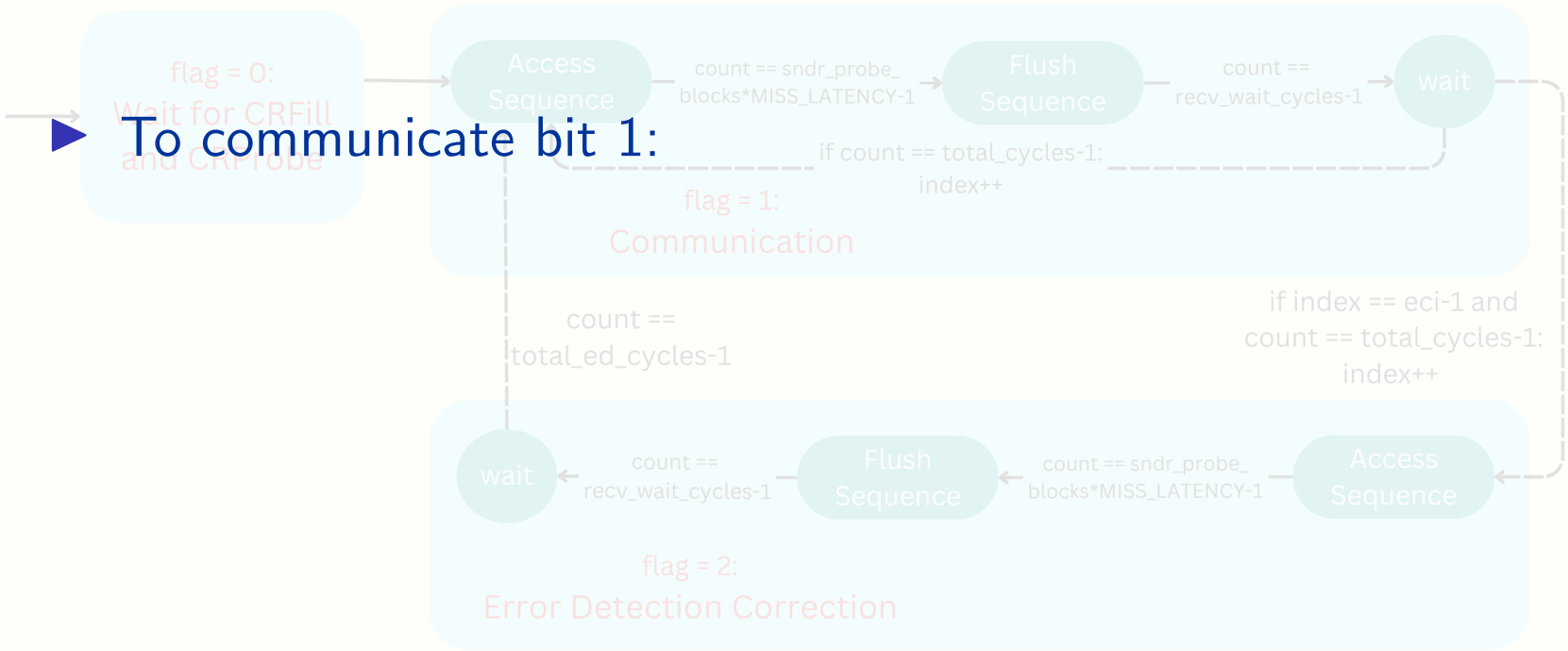


Figure: Sender FSM

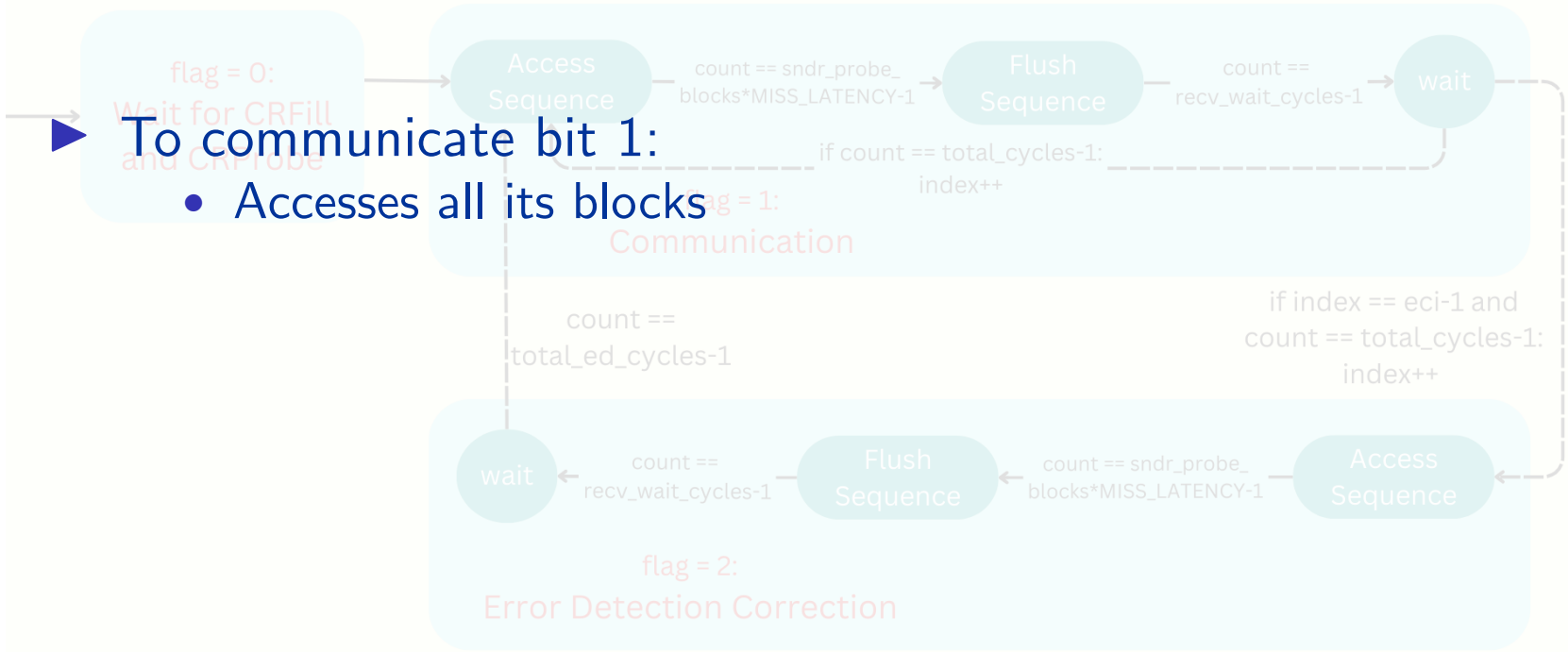
Sender: Communication

Sender: Communication

- To communicate bit 1:

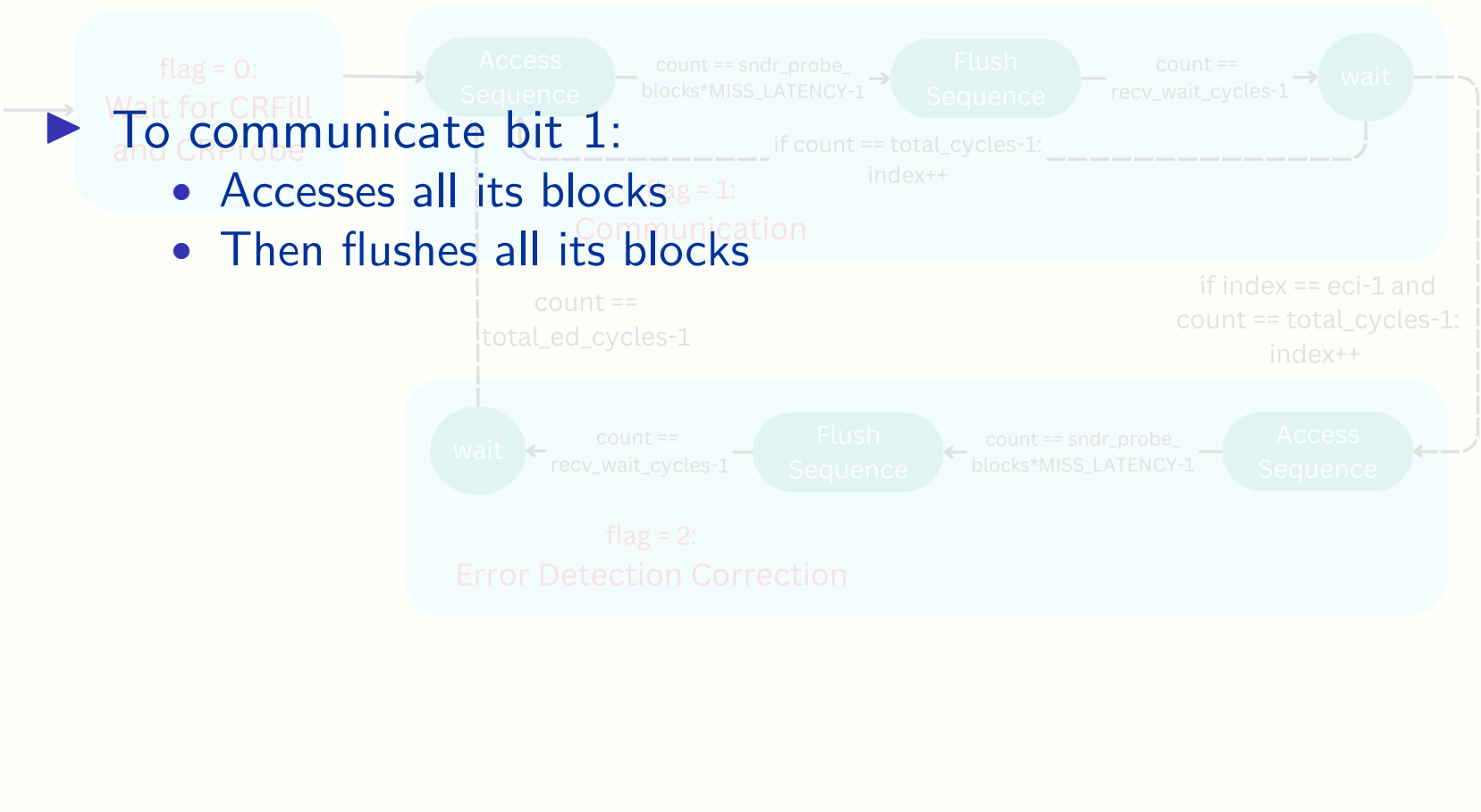


Sender: Communication



- To communicate bit 1:
  - Accesses all its blocks

Sender: Communication

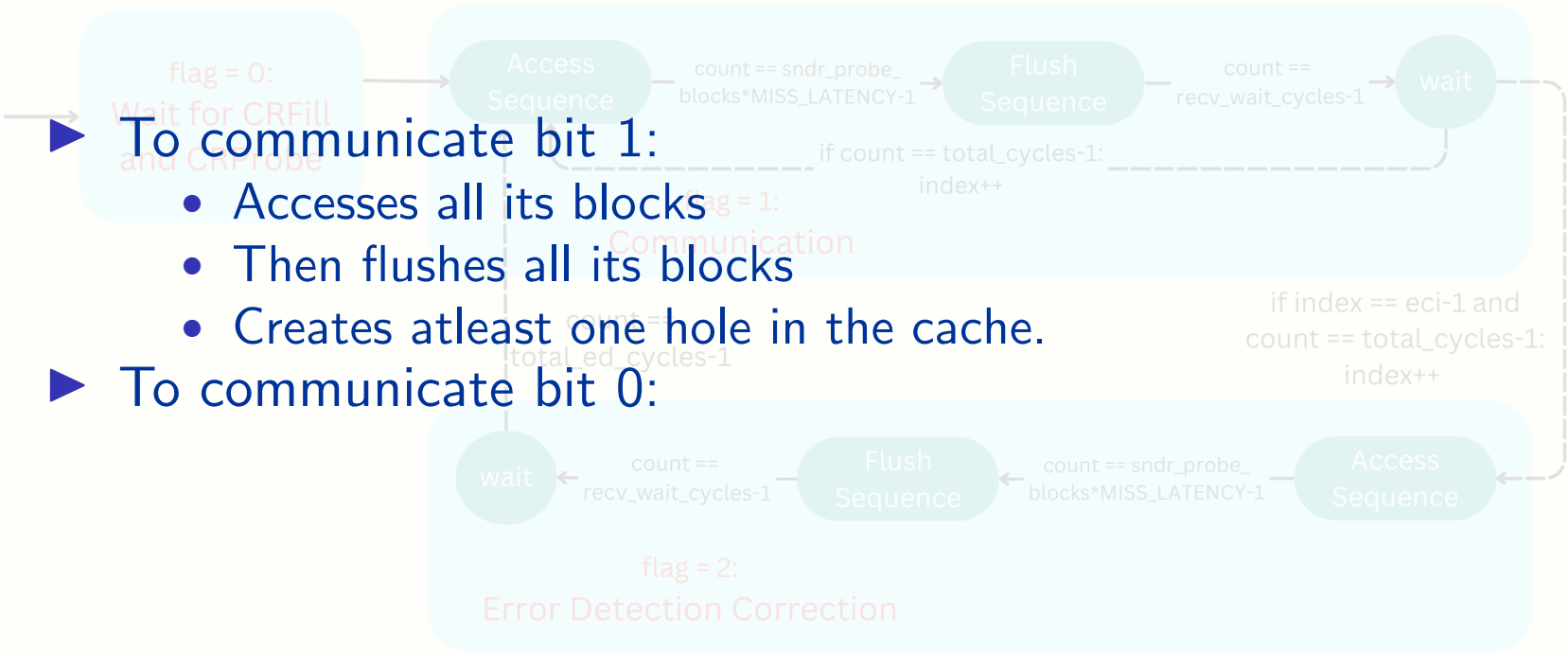


- To communicate bit 1:
  - Accesses all its blocks
  - Then flushes all its blocks

Sender: Communication

- To communicate bit 1:
  - Accesses all its blocks
  - Then flushes all its blocks
  - Creates atleast one hole in the cache.

Sender: Communication



- To communicate bit 1:
  - Accesses all its blocks
  - Then flushes all its blocks
  - Creates atleast one hole in the cache.
- To communicate bit 0:

Sender: Communication

- To communicate bit 1:
  - Accesses all its blocks
  - Then flushes all its blocks
  - Creates atleast one hole in the cache.
- To communicate bit 0:
  - Don't disturb reciever's blocks.

Sender: Communication

- To communicate bit 1:
  - Accesses all its blocks
  - Then flushes all its blocks
  - Creates atleast one hole in the cache.
- To communicate bit 0:
  - Don't disturb reciever's blocks.

Note

The set of addresses sender uses is disjoint from the set of addresses receiver uses.

Design:Sender

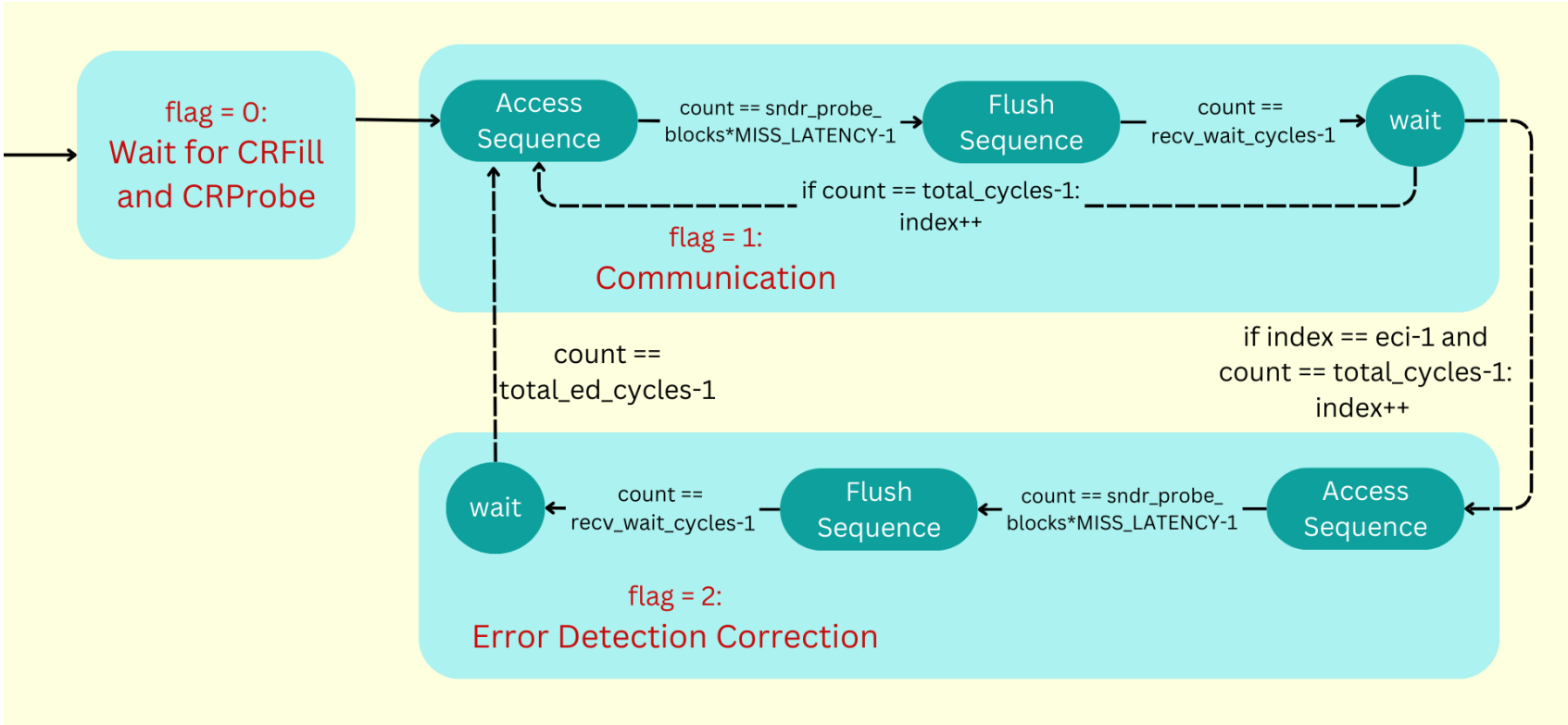


Figure: Sender FSM

Receiver: Communication

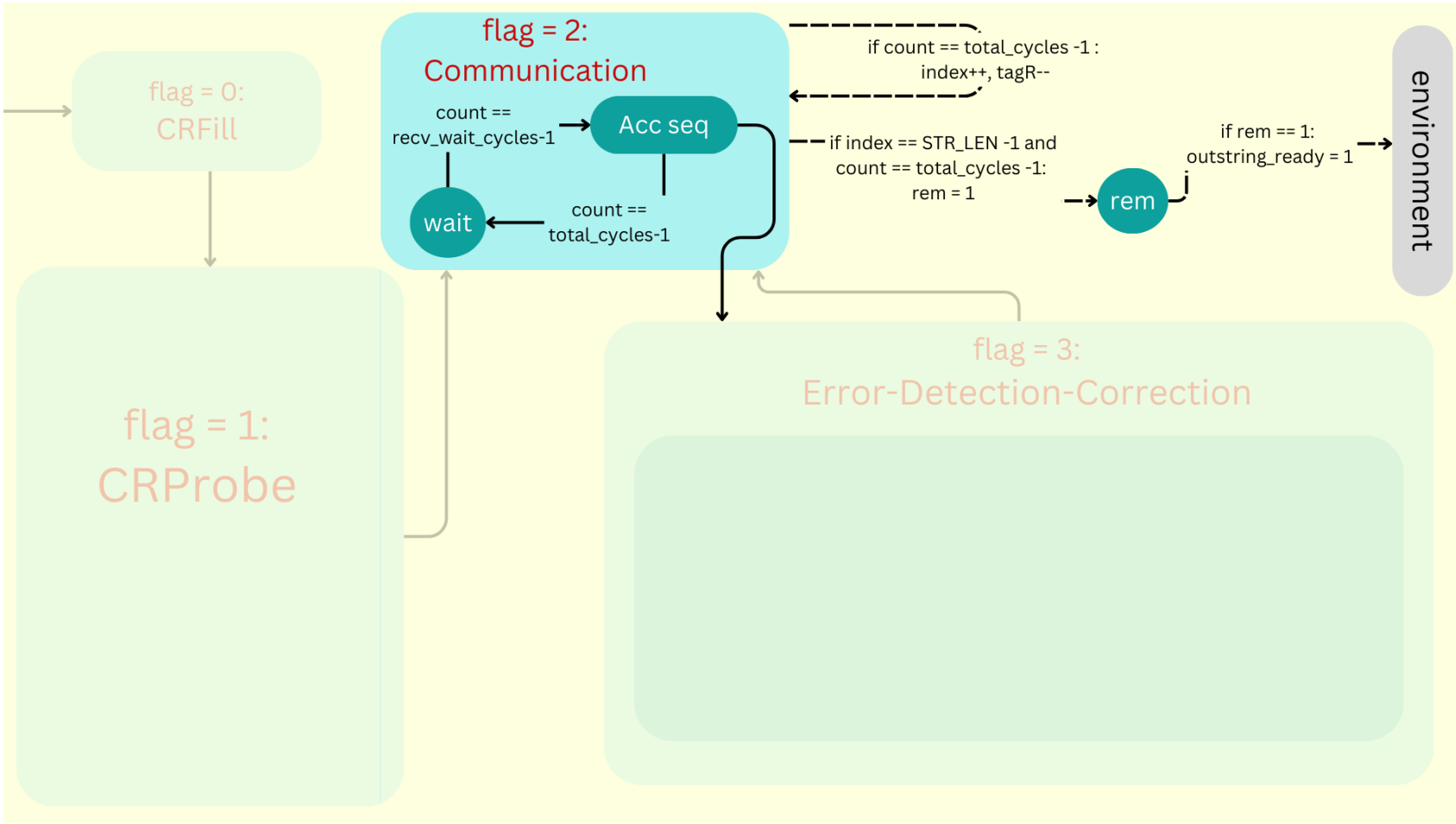
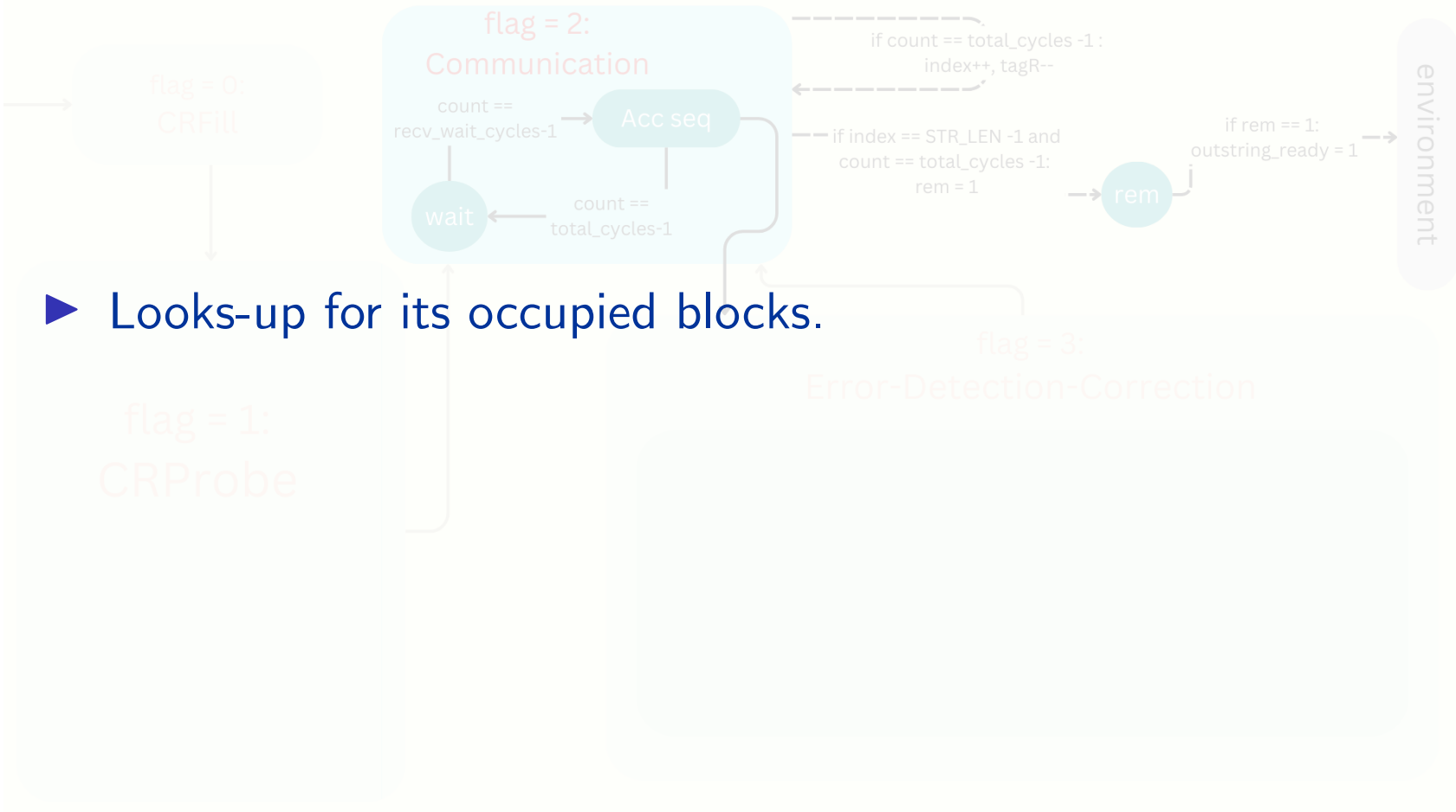


Figure: Communication step

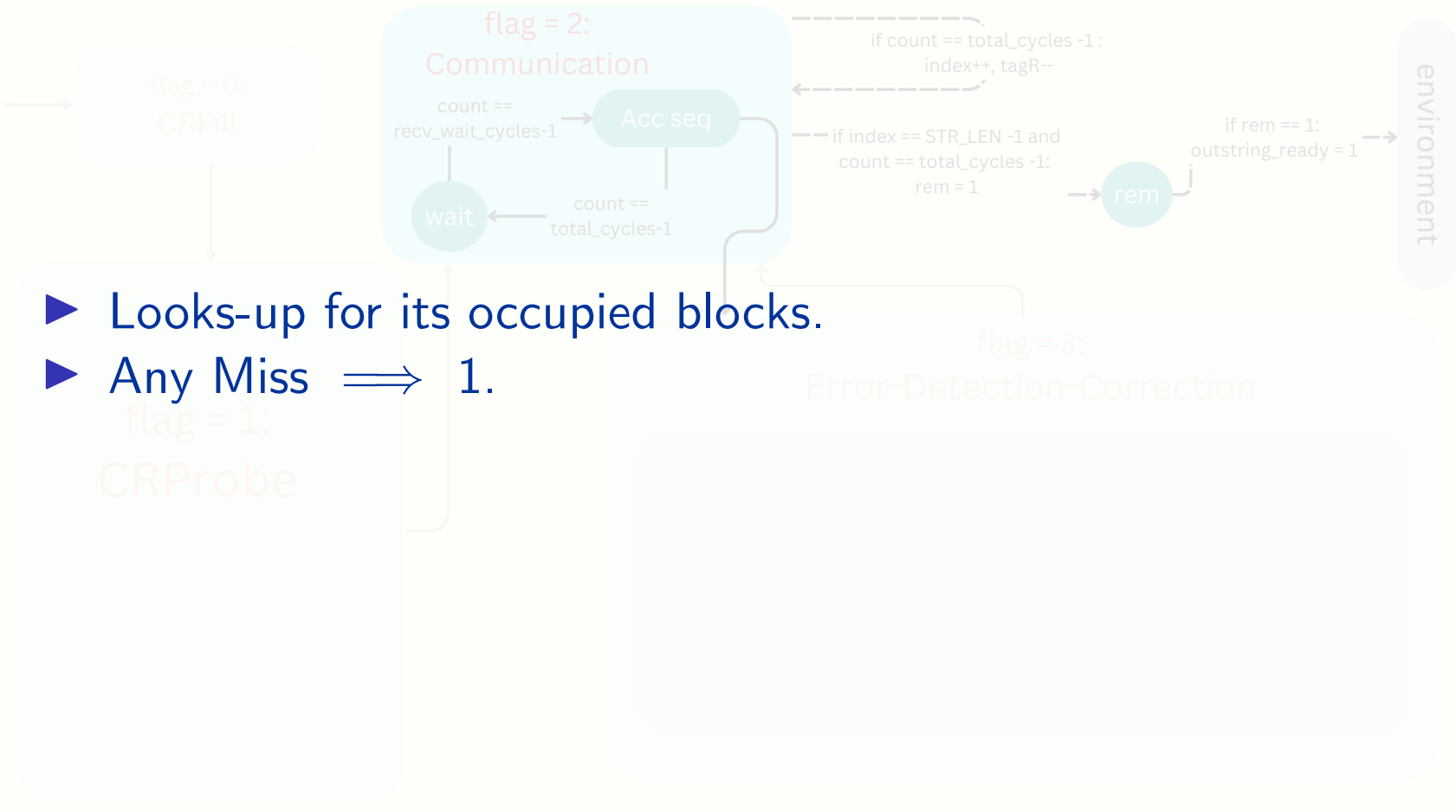
Receiver: Communication

Receiver: Communication



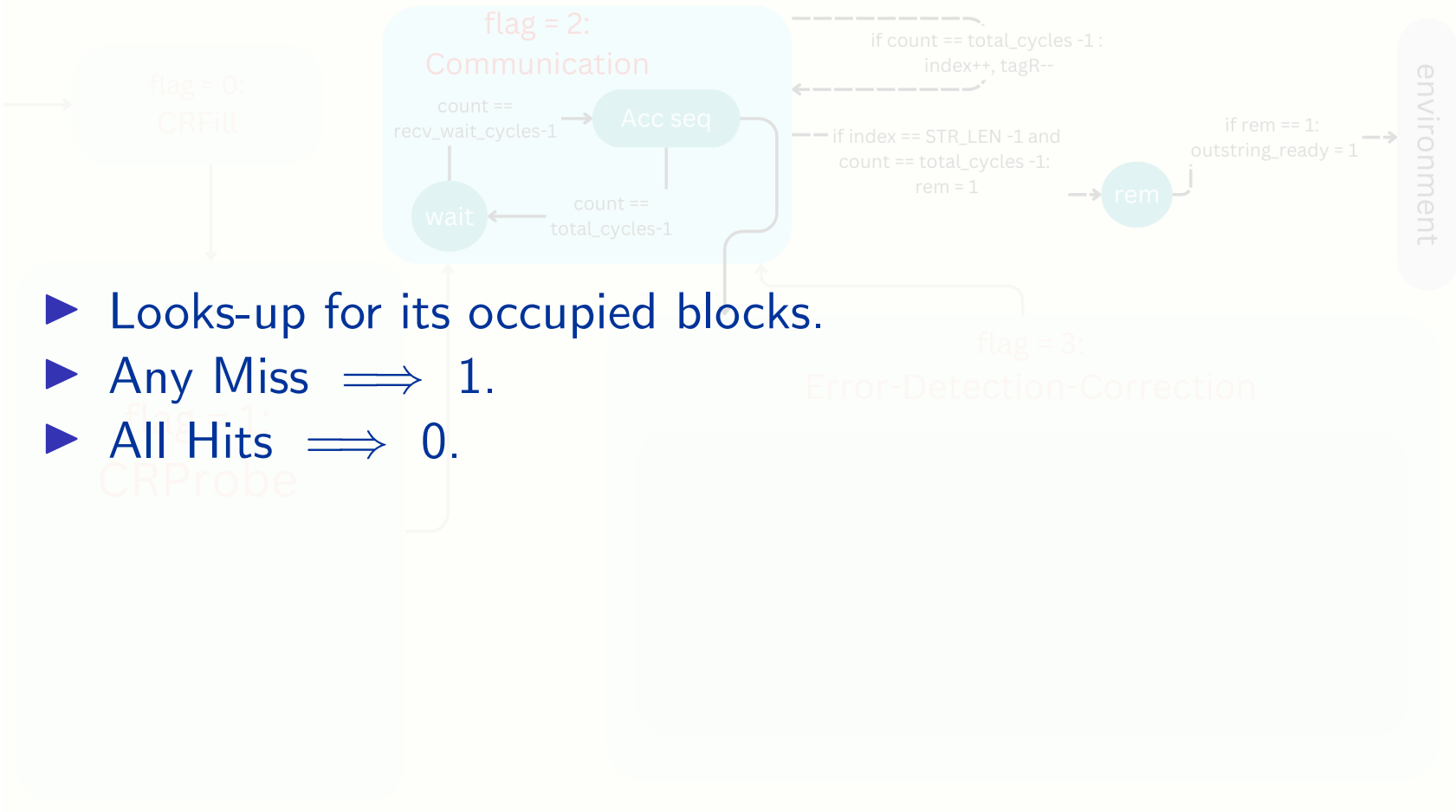
- Looks-up for its occupied blocks.

Receiver: Communication



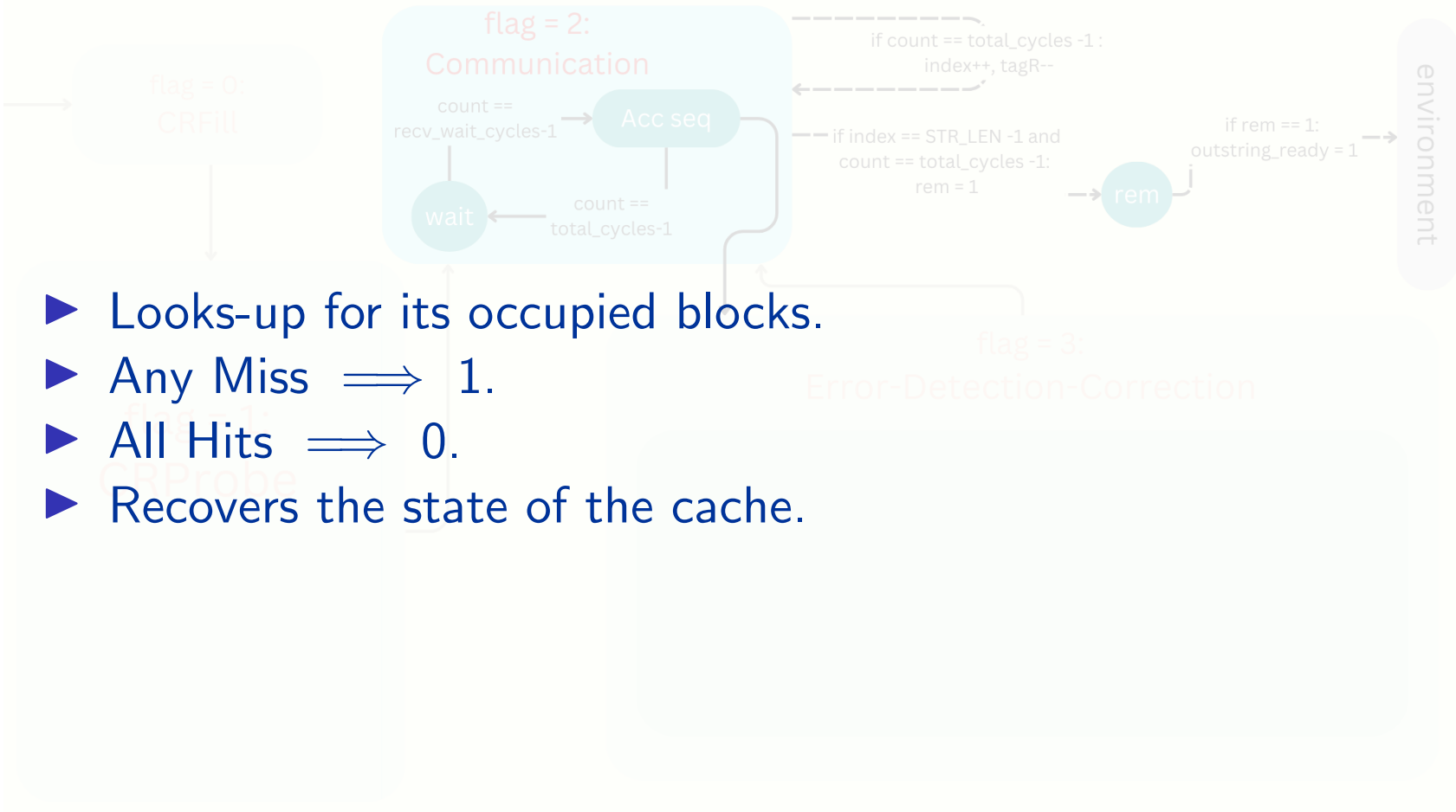
- Looks-up for its occupied blocks.
- Any Miss $\Longrightarrow$ 1.

Receiver: Communication



- Looks-up for its occupied blocks.
- Any Miss $\Longrightarrow$ 1.
- All Hits $\Longrightarrow$ 0.

Receiver: Communication



- Looks-up for its occupied blocks.
- Any Miss $\implies$ 1.
- All Hits $\implies$ 0.
- Recovers the state of the cache.

Receiver: Communication

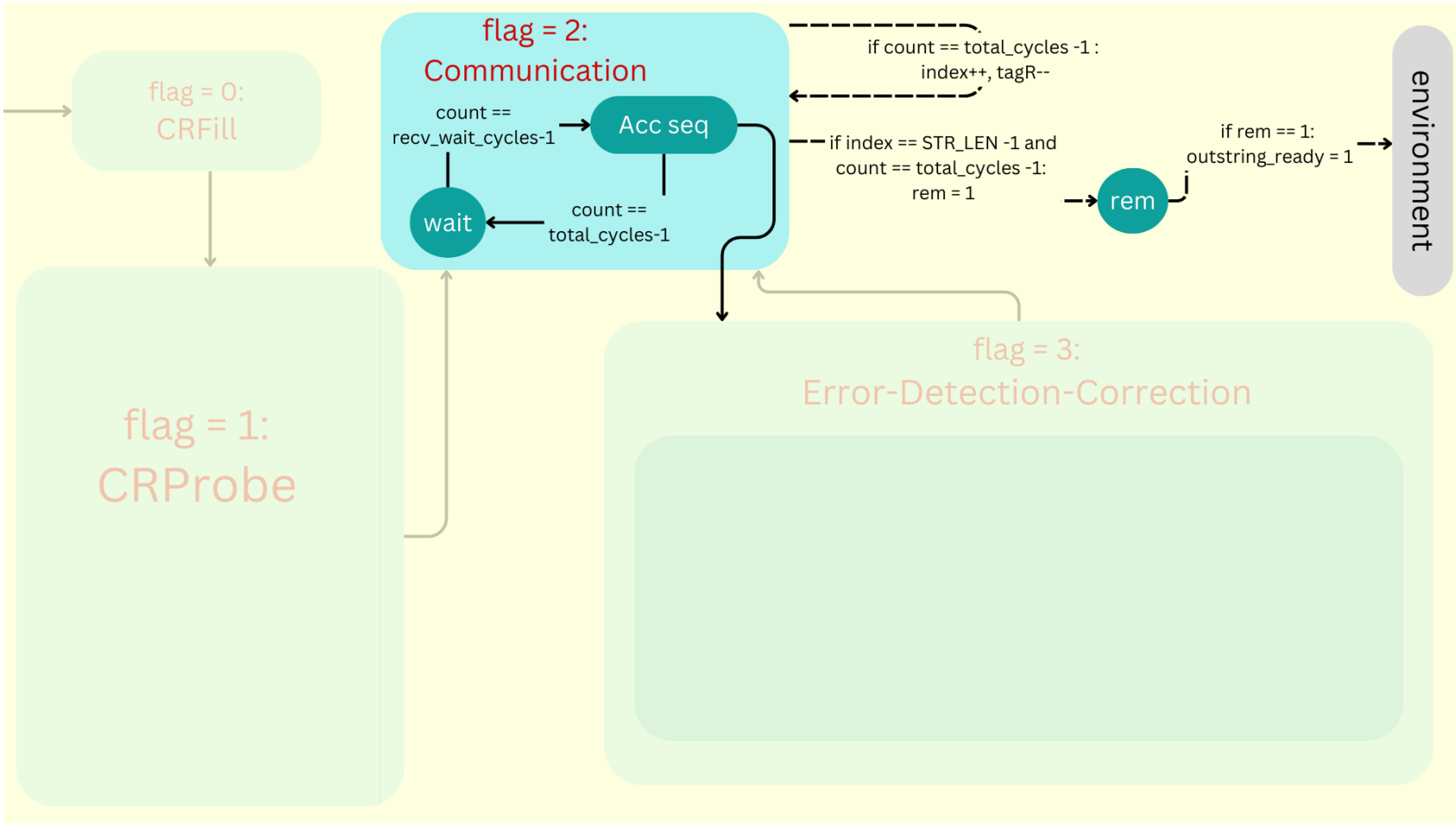


Figure: Communication step

How errors can occur?

How errors can occur?

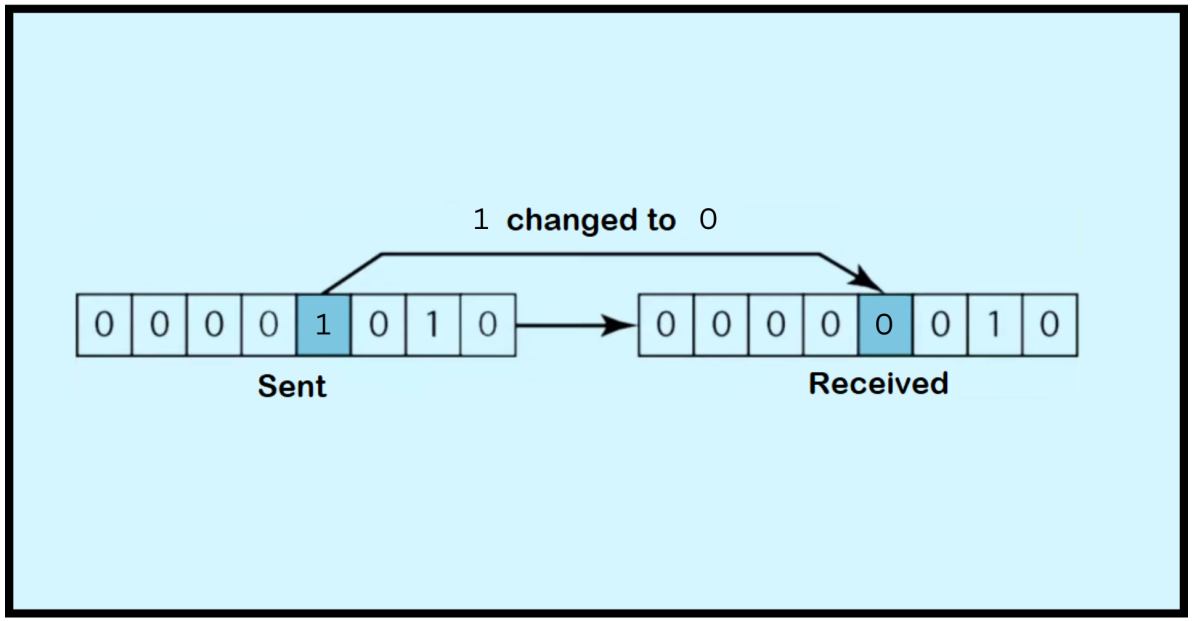


Figure: Possibility of Error

How errors can occur?

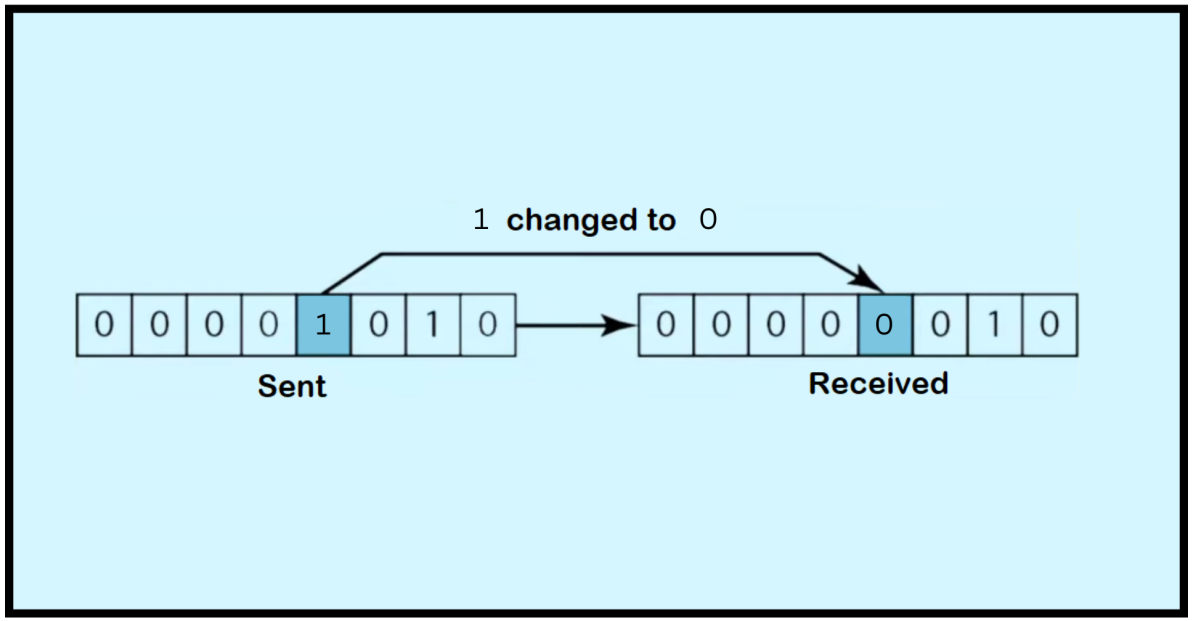


Figure: Possibility of Error

- If occupancy $\neq 100\%$, sender may only evict non-receiver blocks.

How errors can occur?

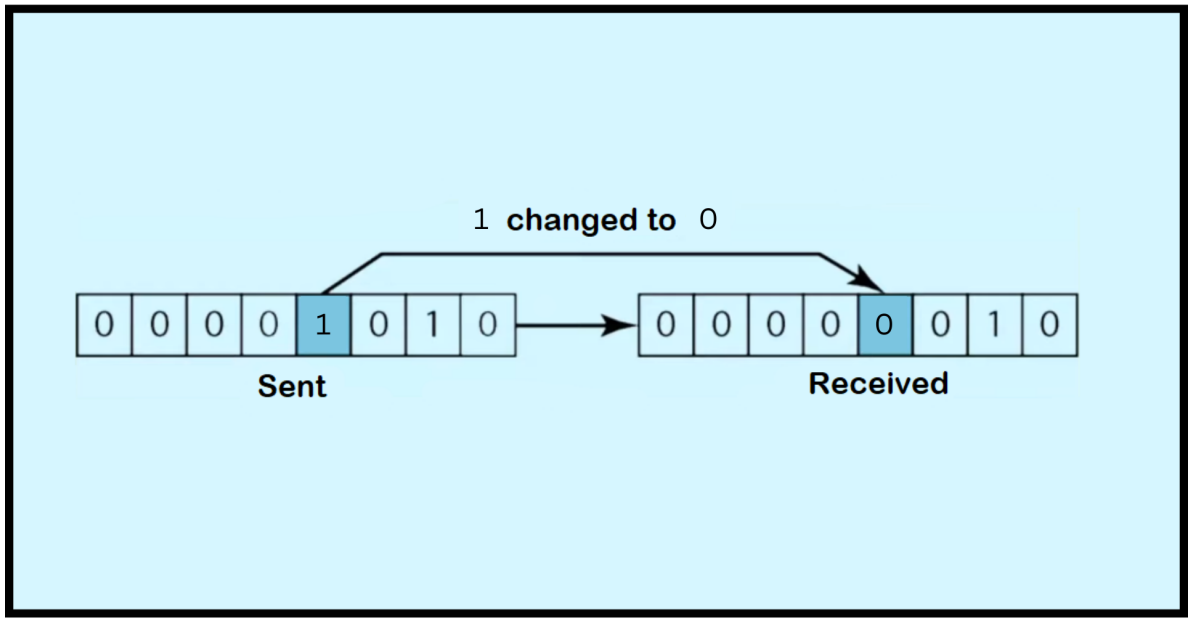


Figure: Possibility of Error

- If occupancy $\neq 100\%$, sender may only evict non-receiver blocks.
- Holes created by the sender may never get filled.

How errors can occur?

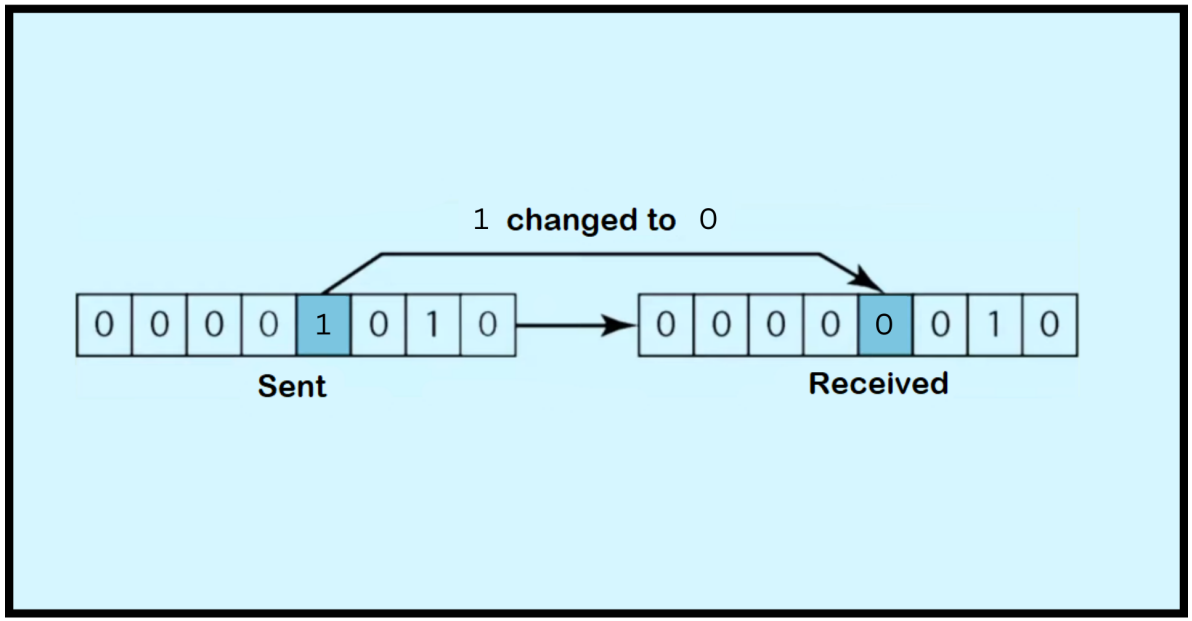


Figure: Possibility of Error

- If occupancy $\neq 100\%$, sender may only evict non-receiver blocks.
- Holes created by the sender may never get filled.
- Result: Sender sent 1, but receiver received 0!

Receiver: EDC step

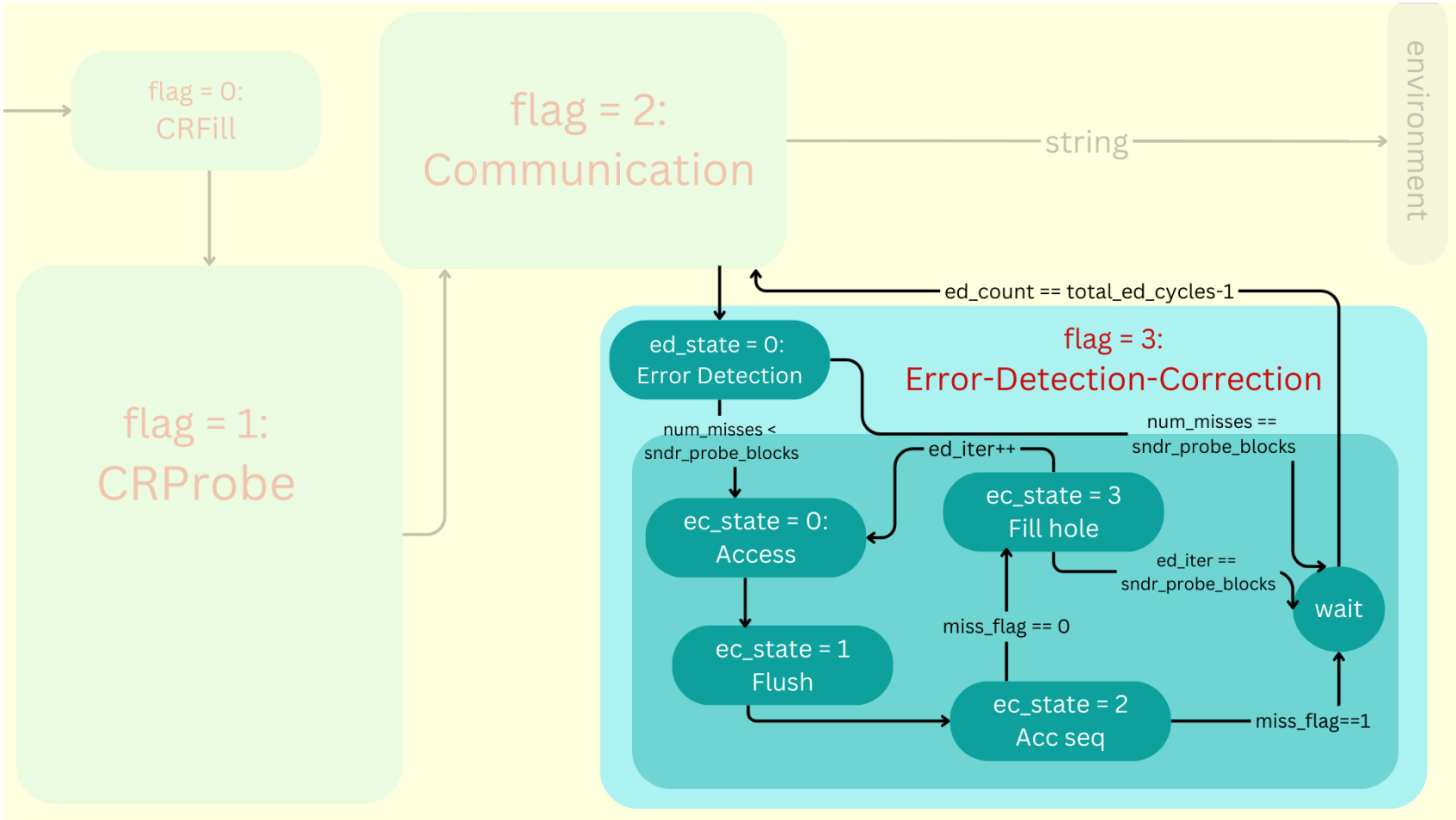
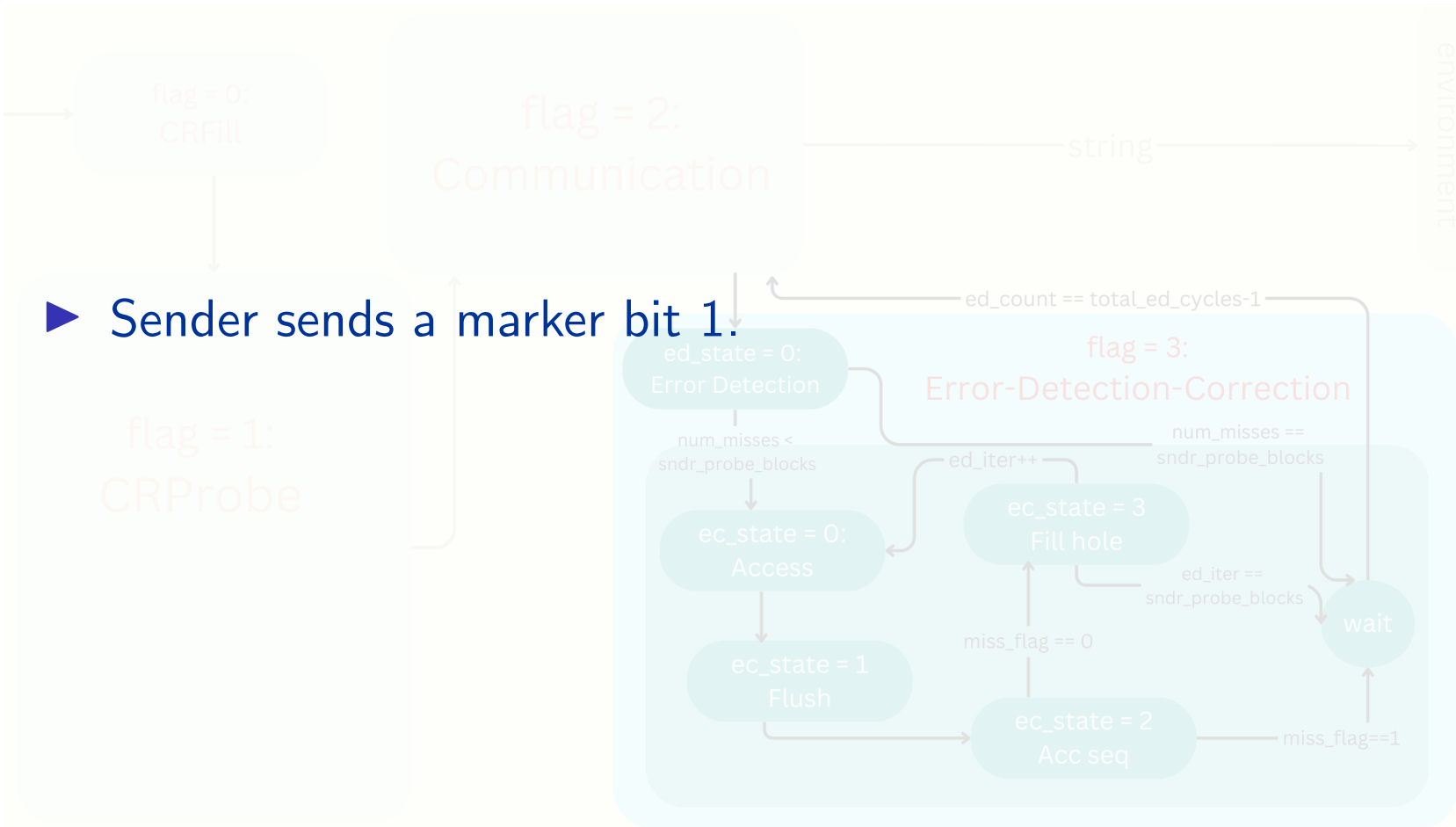


Figure: Error-Detection-Correction Step

Receiver: Error Detection

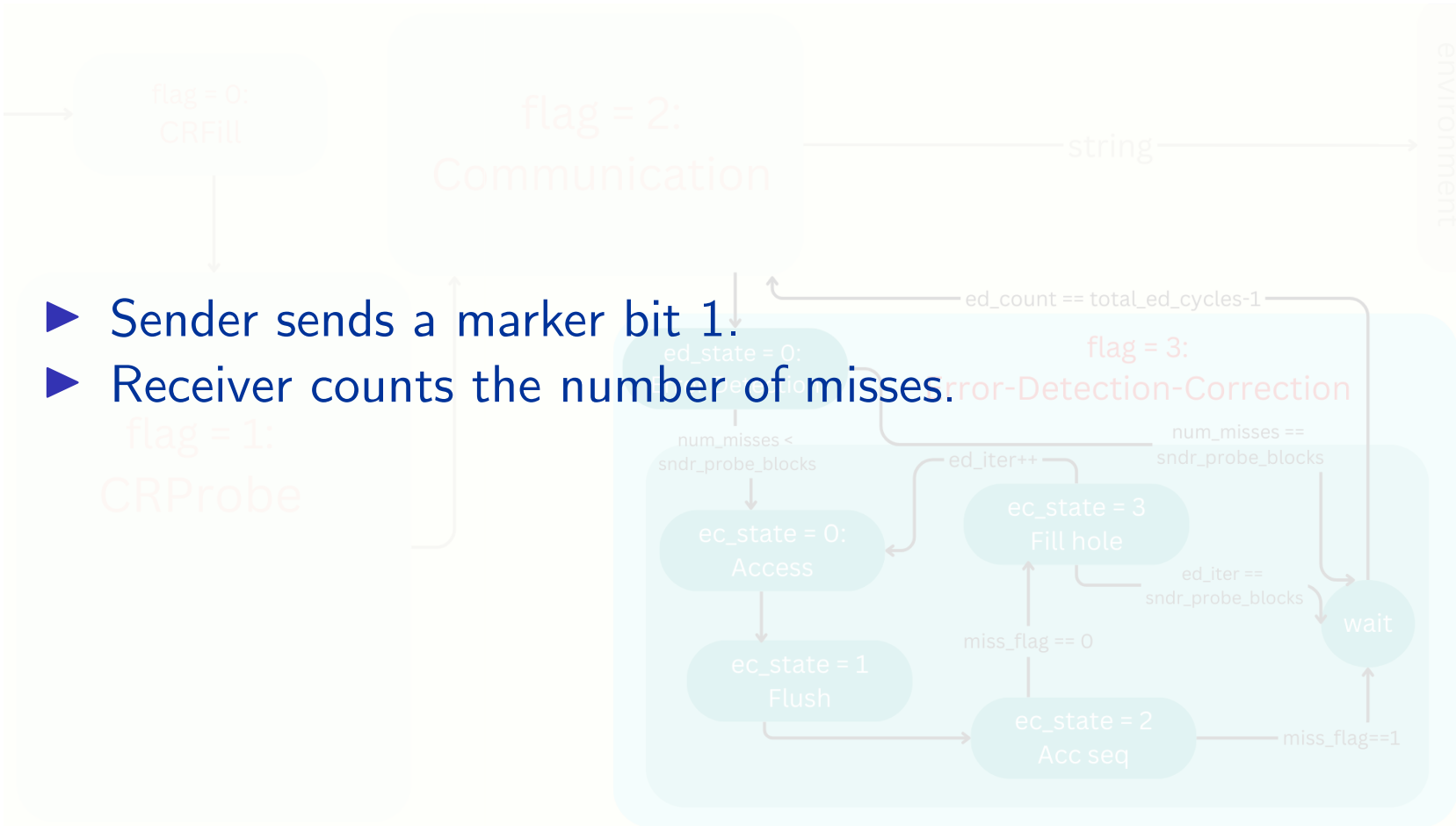
Receiver: Error Detection

- Sender sends a marker bit 1.



Receiver: Error Detection

- Sender sends a marker bit 1.
- Receiver counts the number of misses.



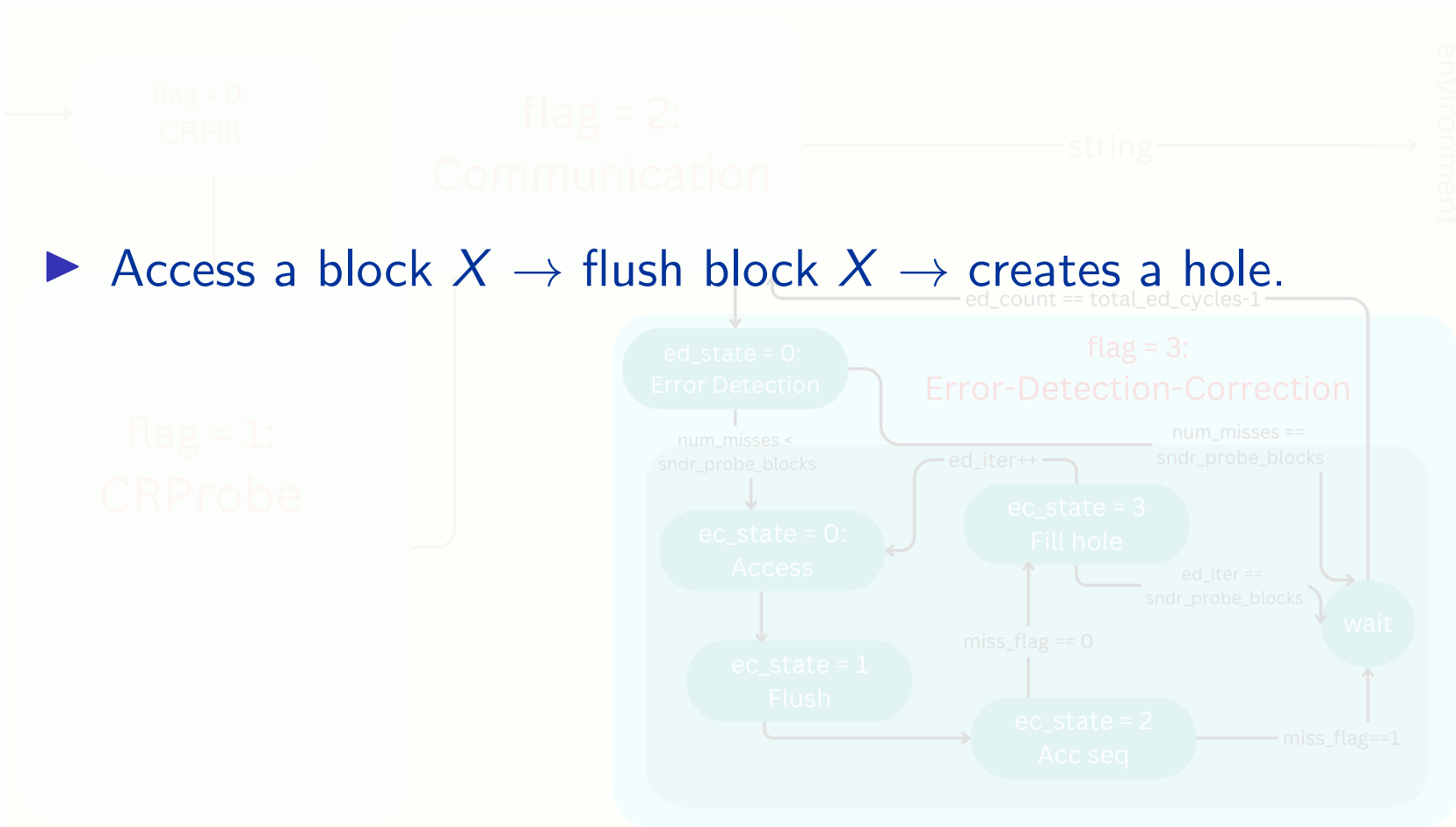
Receiver: Error Detection

- Sender sends a marker bit 1.
- Receiver counts the number of misses.
- Number of misses = *sndr_probe_blocks* $\implies$ a hole cannot be present $\implies$ Error Correction not needed.

Receiver: Error Correction

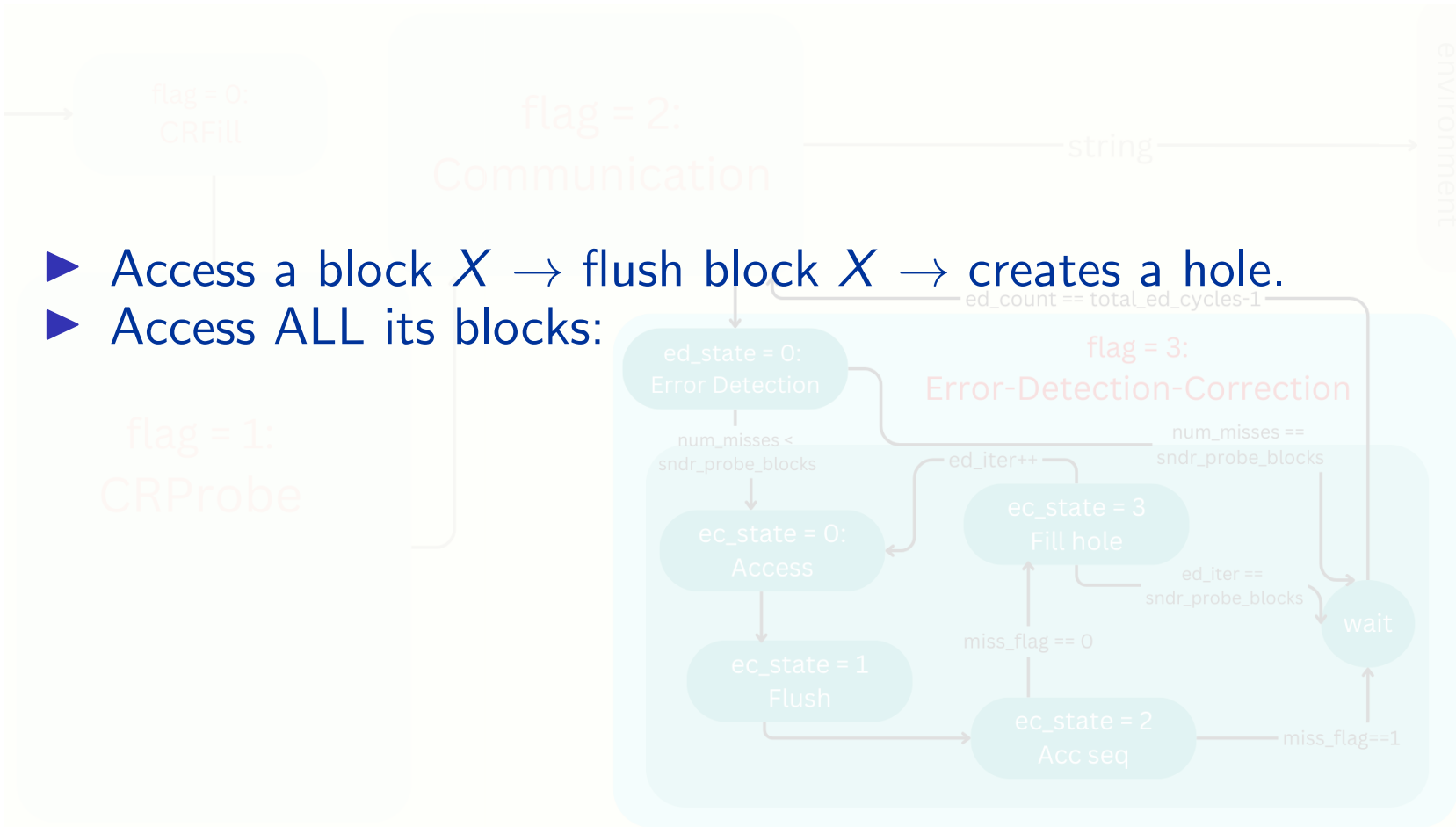
Receiver: Error Correction

- Access a block $X \rightarrow$ flush block $X \rightarrow$ creates a hole.



Receiver: Error Correction

- Access a block $X \rightarrow$ flush block $X \rightarrow$ creates a hole.
- Access ALL its blocks:



Receiver: Error Correction

- Access a block $X \rightarrow$ flush block $X \rightarrow$ creates a hole.
- Access ALL its blocks:
  - No miss detected $\implies$ A hole was there, access X again, **increase occupancy** by 1.

Receiver: Error Correction

- Access a block $X \rightarrow$ flush block $X \rightarrow$ creates a hole.
- Access ALL its blocks:
  - No miss detected $\implies$ A hole was there, access X again, **increase occupancy** by 1.
  - Miss detected $\implies$ No hole in cache, stop error correction.

Receiver: Error Correction

- Access a block $X \rightarrow$ flush block $X \rightarrow$ creates a hole.
- Access ALL its blocks:
  - No miss detected $\implies$ A hole was there, access X again, **increase occupancy** by 1.
  - Miss detected $\implies$ No hole in cache, stop error correction.
- Repeat until miss detected.

Receiver: EDC step

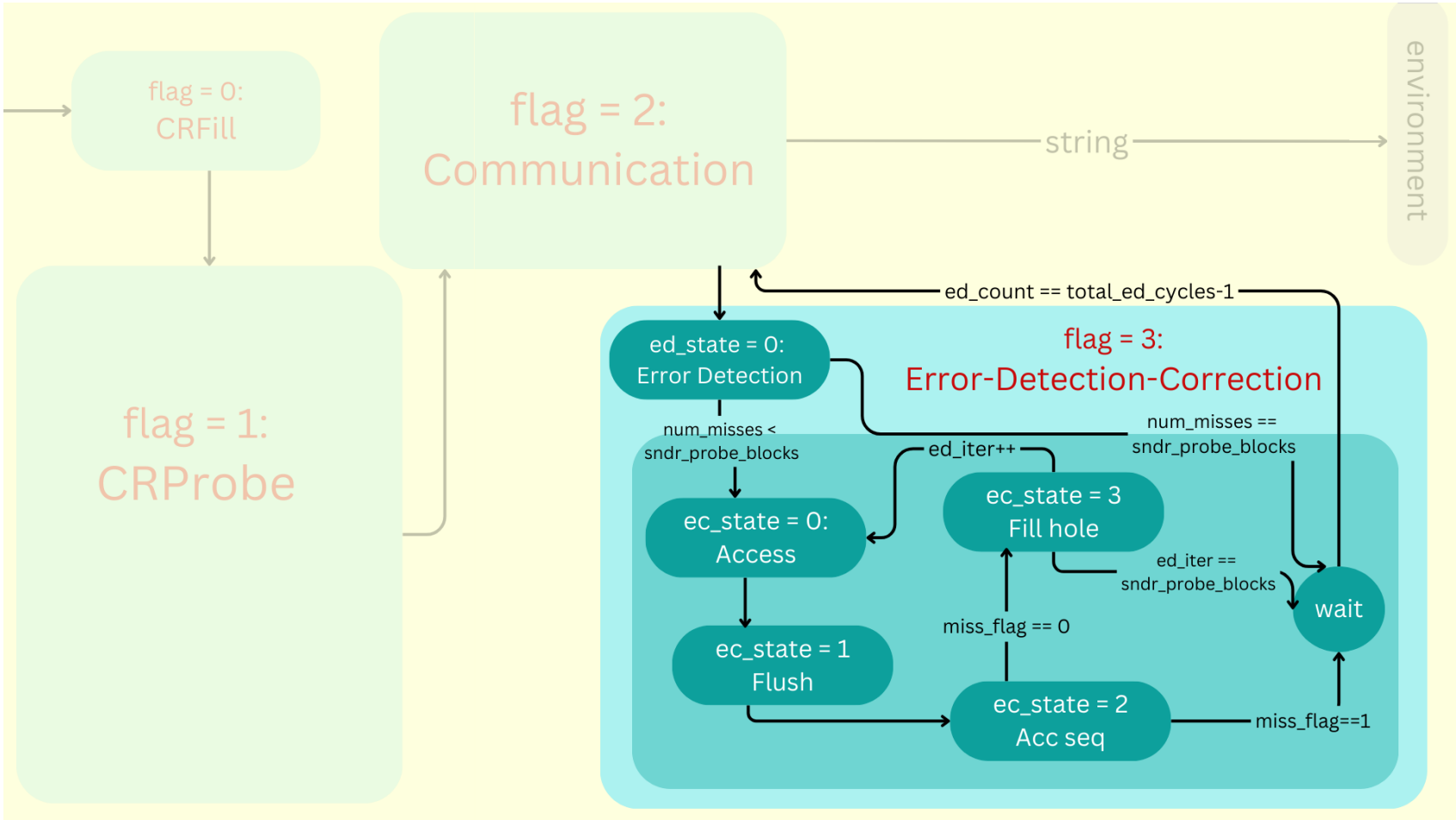


Figure: Error-Detection-Correction Step

Design:Receiver

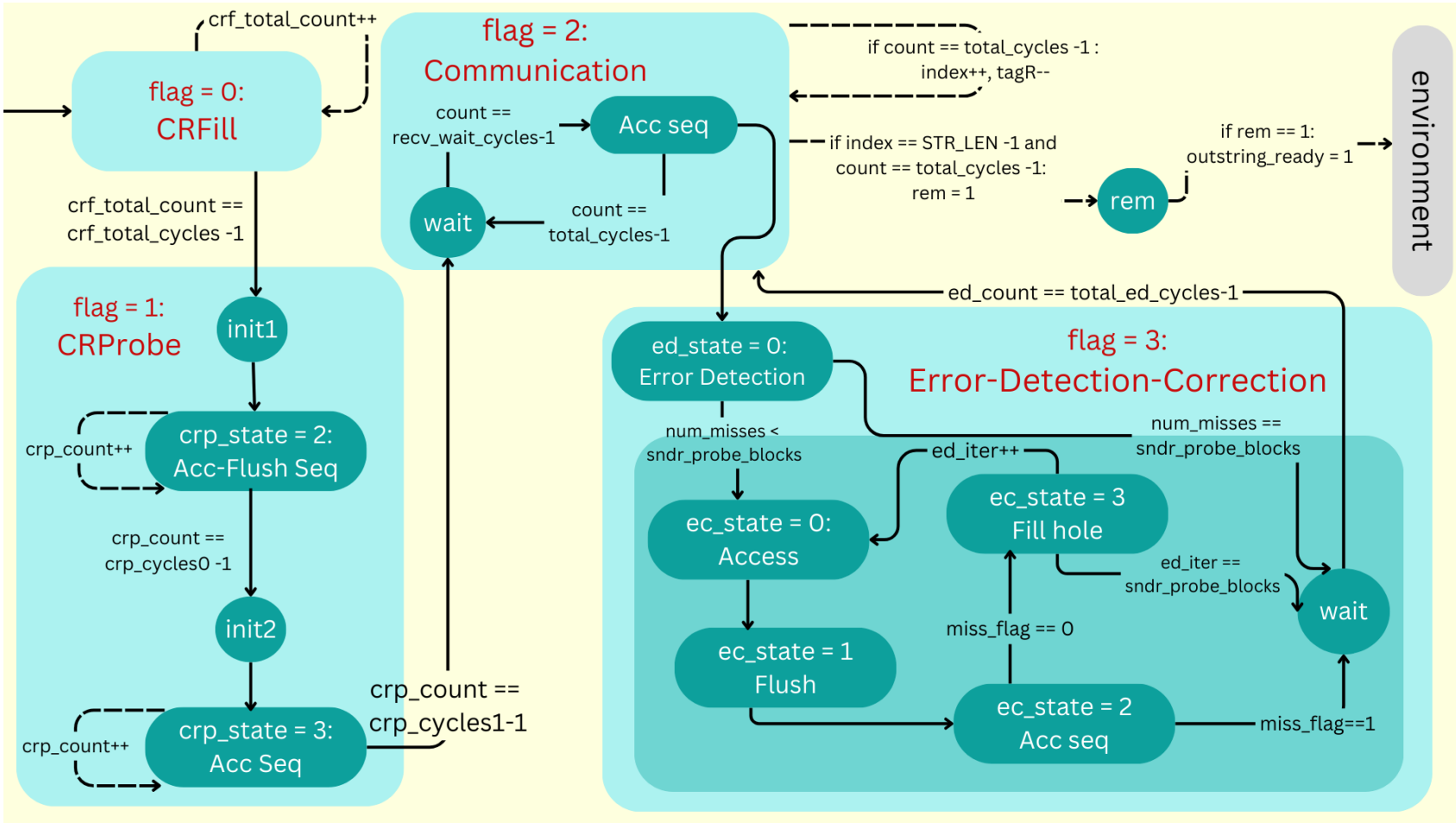
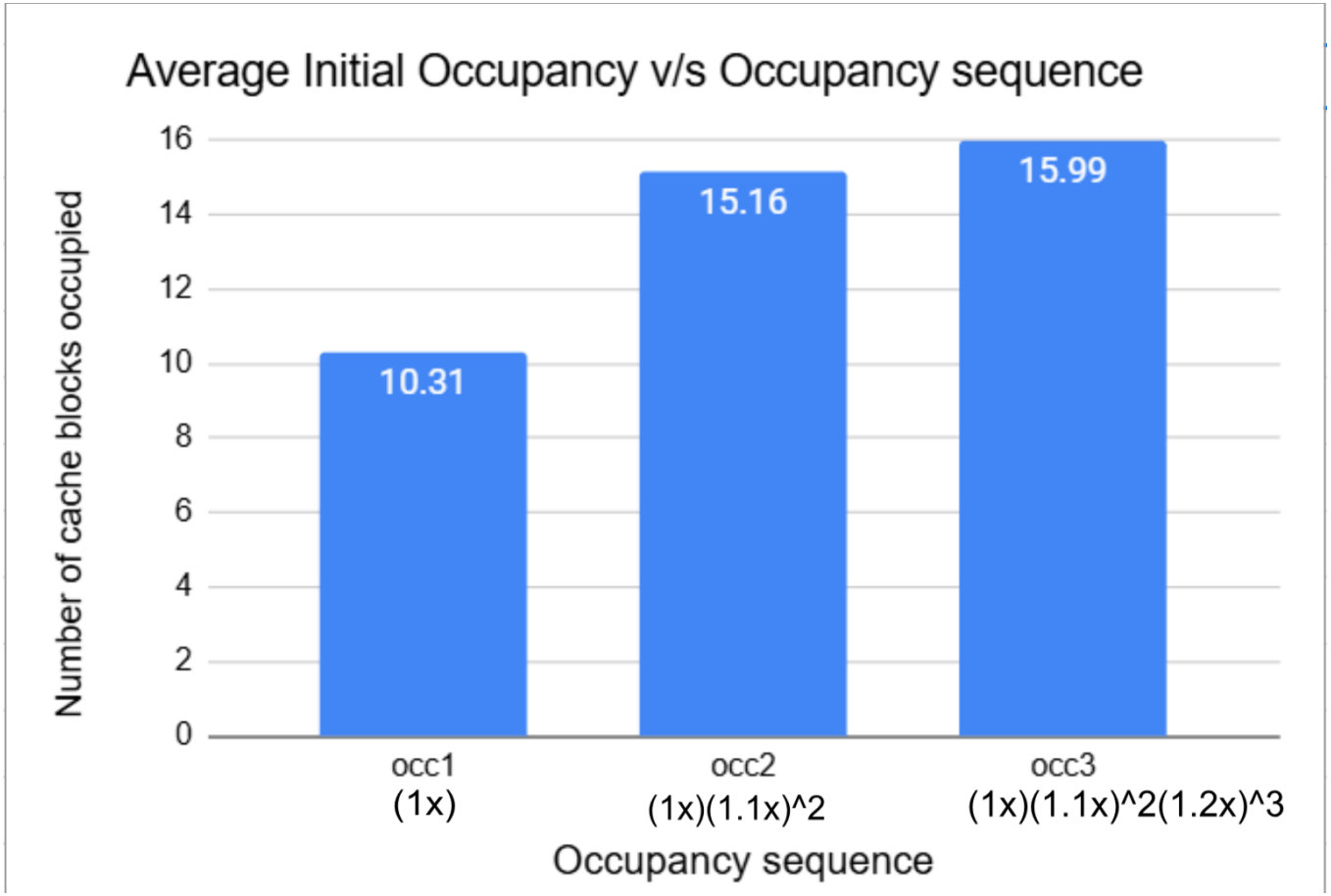


Figure: Receiver FSM: Complete view

Results & Observations

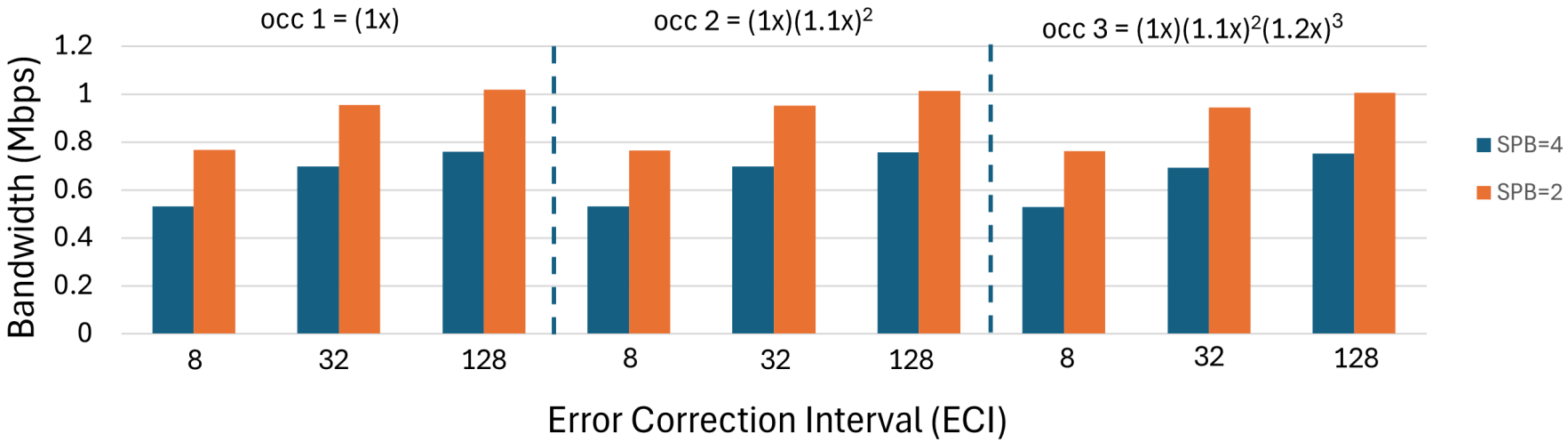
Results & Observations



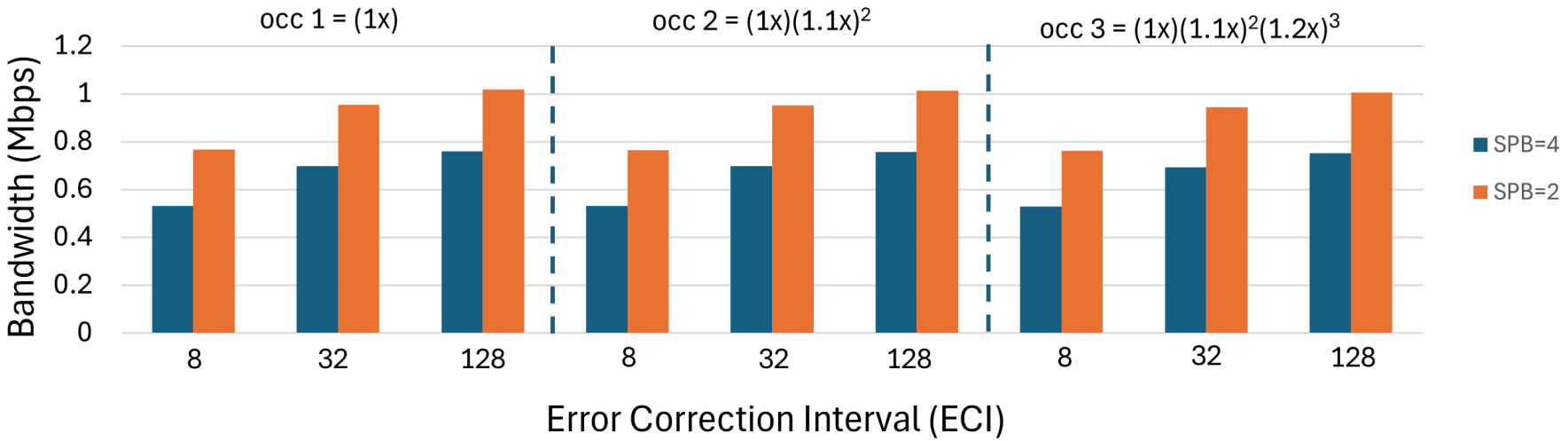
- A better occupancy sequence significantly increases the initial cache occupancy of the receiver.

Results & Observations

Results & Observations

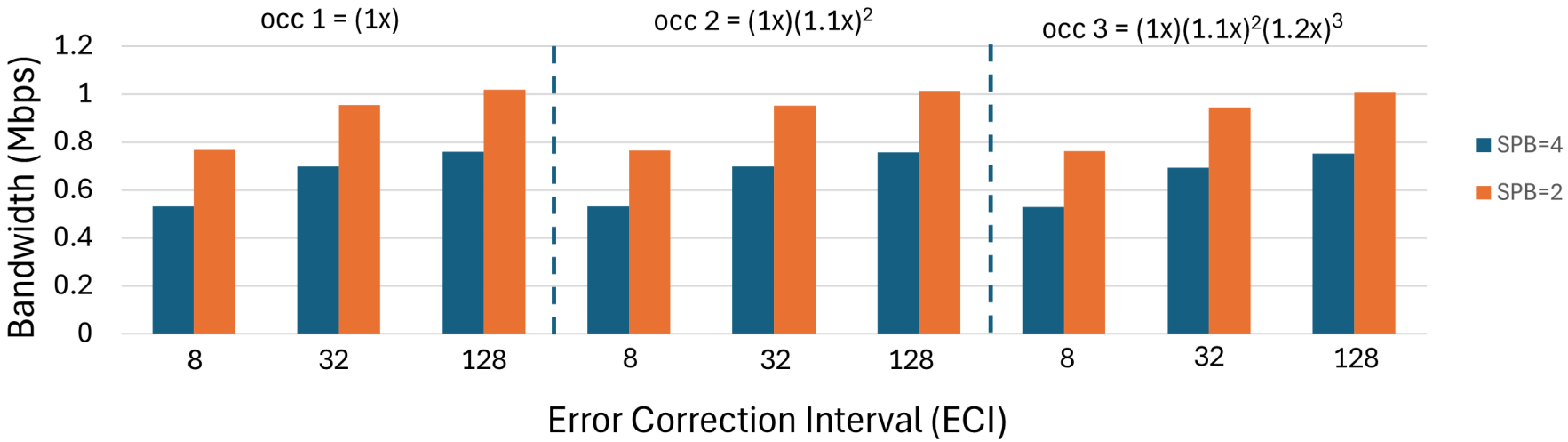


Results & Observations



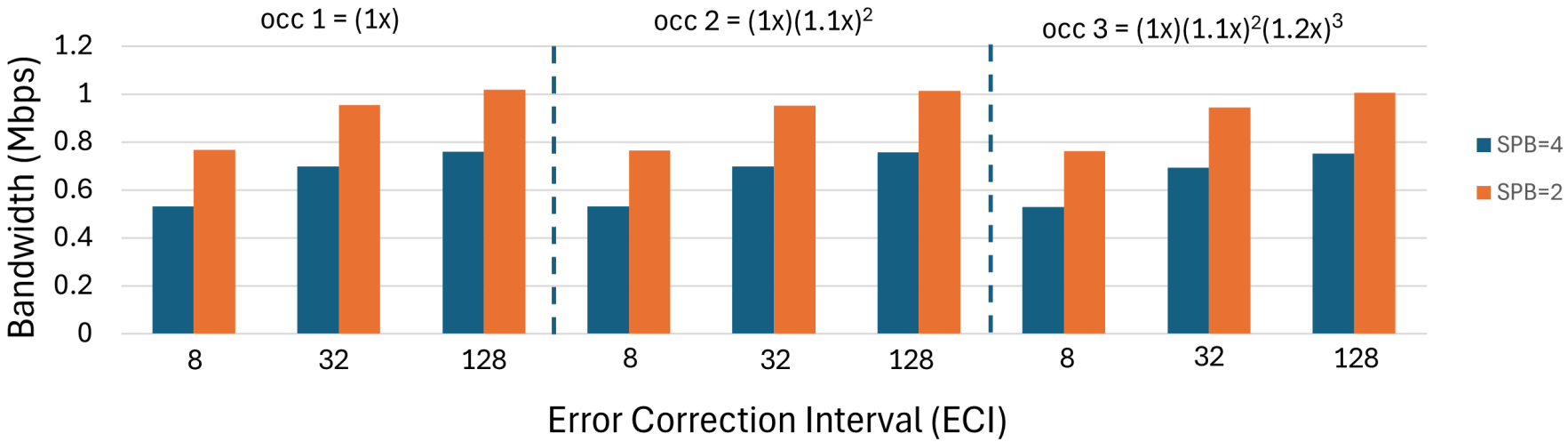
- If ECI ↑, then Bandwidth ↑.

Results & Observations



- If ECI ↑, then Bandwidth ↑.
- If SPB ↓, then Bandwidth ↑.

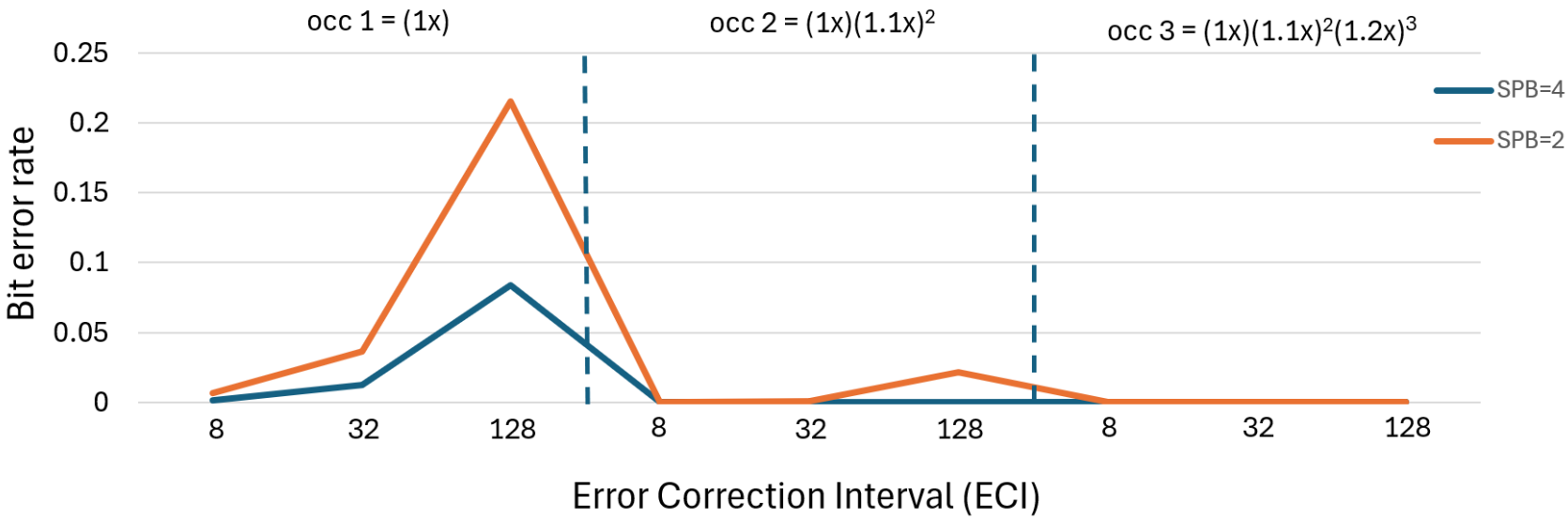
Results & Observations



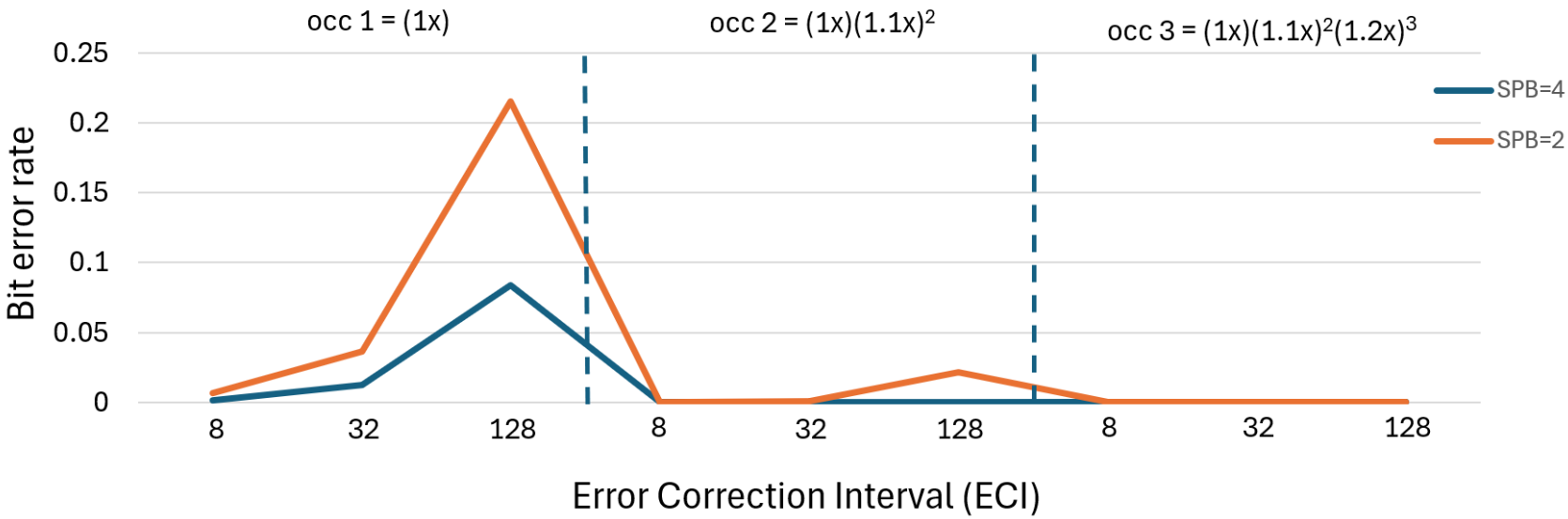
- If ECI ↑, then Bandwidth ↑.
- If SPB ↓, then Bandwidth ↑.
- Bandwidth Overhead for a larger occupancy sequence is negligible.

Results & Observations

Results & Observations

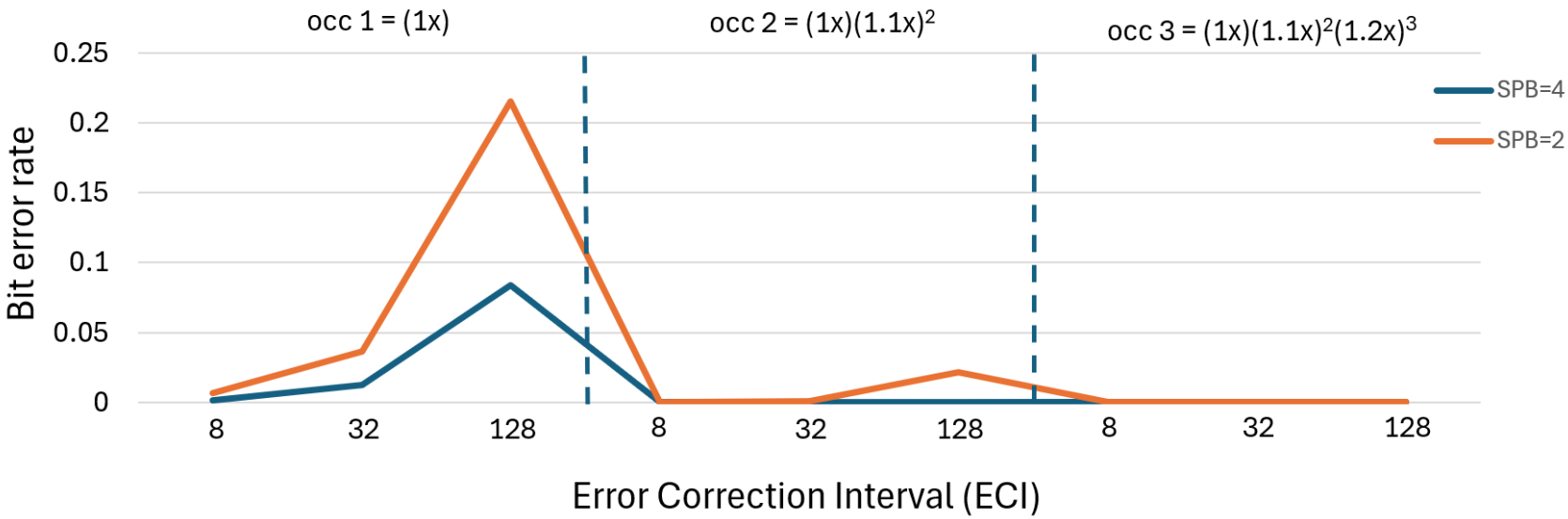


Results & Observations



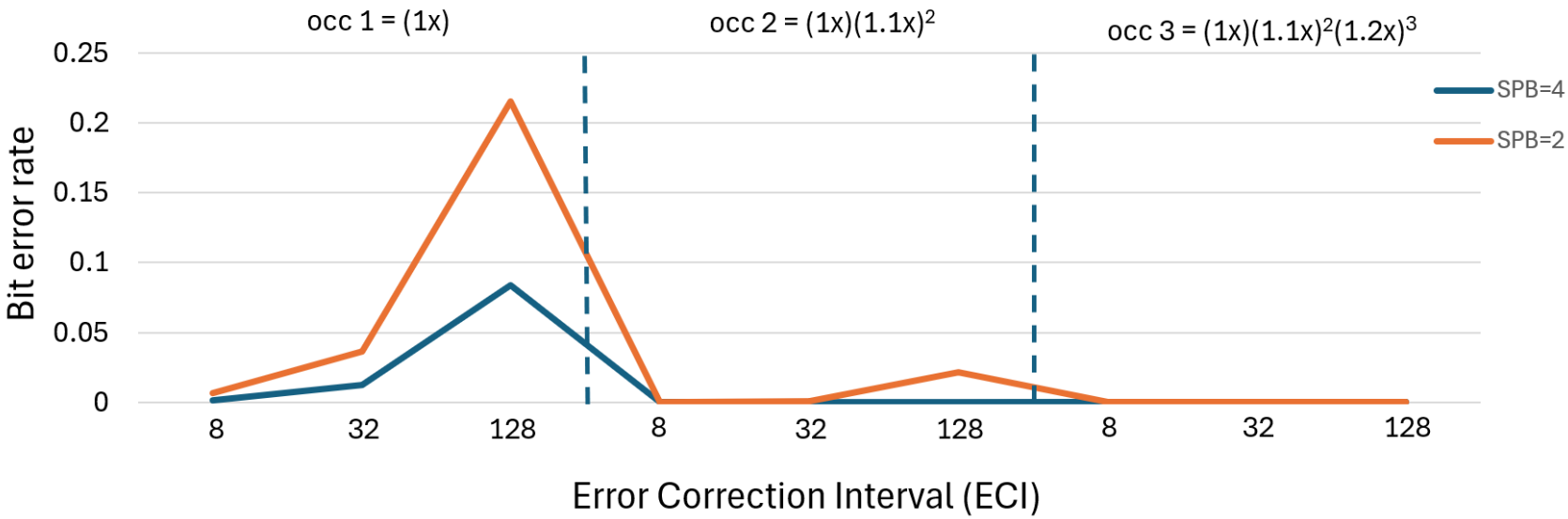
- If ECI ↓, then BER ↓.

Results & Observations



- If ECI ↓, then BER ↓.
- If SPB ↑, then BER ↓.

Results & Observations



- If ECI ↓, then BER ↓.
- If SPB ↑, then BER ↓.
- A better occupancy sequence reduces the errors significantly.

Demonstration

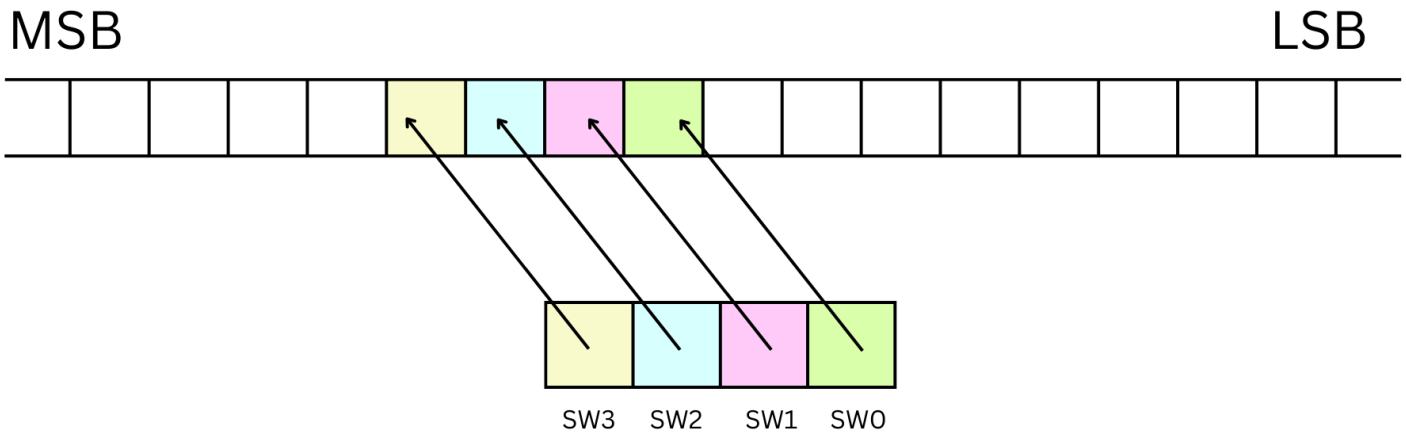


Figure: Input Convention

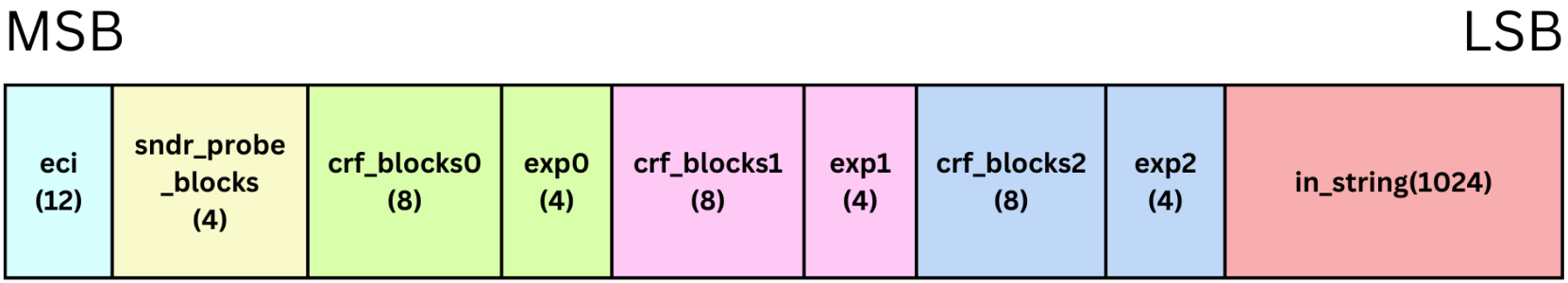


Figure: Sequence of Input Data

Future Improvements

Future Improvements

- Increase the cache size.

Future Improvements

- Increase the cache size.
- Introduce multiple levels in the cache hierarchy.

Future Improvements

- Increase the cache size.
- Introduce multiple levels in the cache hierarchy.
- Introduce DRAM support.

Future Improvements

- Increase the cache size.
- Introduce multiple levels in the cache hierarchy.
- Introduce DRAM support.
- Make the cache non-blocking. This will make miss and hit latencies variable.

Future Improvements

- Increase the cache size.
- Introduce multiple levels in the cache hierarchy.
- Introduce DRAM support.
- Make the cache non-blocking. This will make miss and hit latencies variable.
- Make the sender and receiver software processes.

Future Improvements

- Increase the cache size.
- Introduce multiple levels in the cache hierarchy.
- Introduce DRAM support.
- Make the cache non-blocking. This will make miss and hit latencies variable.
- Make the sender and receiver software processes.
- Use a better pseudo-random generator.

Thank you for Attending!

See you in our next UGP presentation :)

Questions and Answers!

Bandwidth Calculation

- **Parameter values:**
  - No. of cache blocks(num) = 16
  - SPB = 2
  - ECI = 128
  - Occupancy sequence = $(1x)(1.1x)^2(1.2x)^3 \implies (16)(18)^2(19)^3$.
  - CRProbe blocks (crp_blocks)= (16+18+19) = 53
  - String length (len) = 1024
  - No. of marker bits (mb) = 7
  - Miss Latency (ML) = 4
  - Hit Latency (HL) = 2

Bandwidth Calculation

- **CRFill time:** $(16 + (18 * 2) + (19 * 3)) * ML = (109) * 4 = 436$
- **CRProbe time:**
  $53(ML + HL) + (num * ML) = 53 * 6 + (16 * 4) + 2 = 384$
- **Comm (per bit):**
  $SPB(ML+HL)+SPB*ML+HL(num-SPB) = 2*6+2*4+14*2 = 48$
- **EDC (per m-bit):**
  $Comm + SPB(ML + HL + HL(num - 1) + ML + ML) =$
  $Comm + SPB(3ML + num * HL) = 48 + 2(12 + 32) = 136$
- **Total Comm cycles:** $48 * 1024 = 49152$
- **Total EDC cycles:** $136 * 7 = 952$
- **Total Cycles(end to end):** $436 + 384 + 49152 + 952 = 50924$
- **Clock Cycle Time:** $20ns$
- **Total time taken for 1024 bits:**
  $50924 * 20ns = 1018480ns = 1018.48us$
- **Bandwidth:** $1024/1018.48 * 10^6 = 1.01Mbps$

LFSR

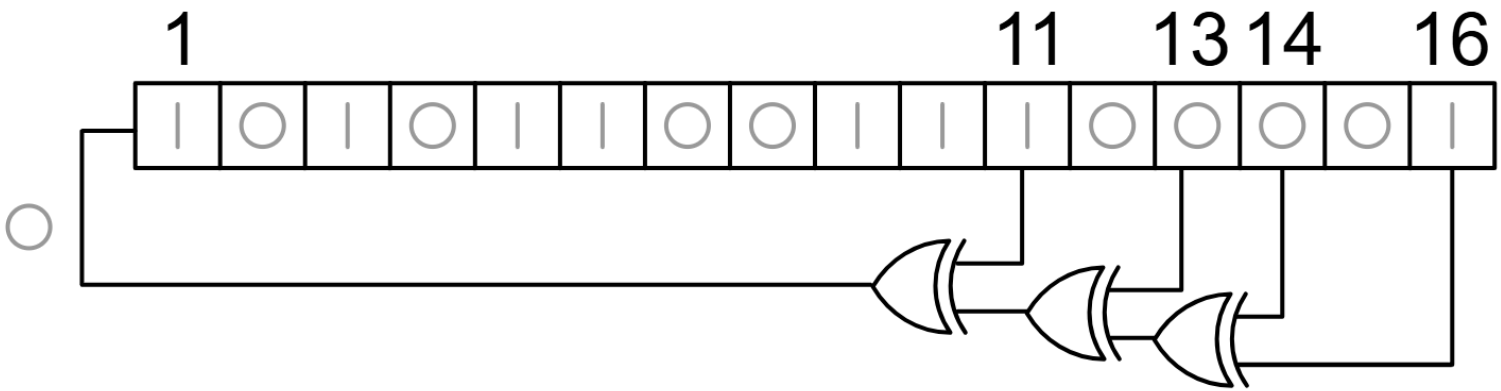


Figure: LFSR[2]

[2]Source : https://en.wikipedia.org/wiki/File:LFSR-F16.svg

Port Connections for Input

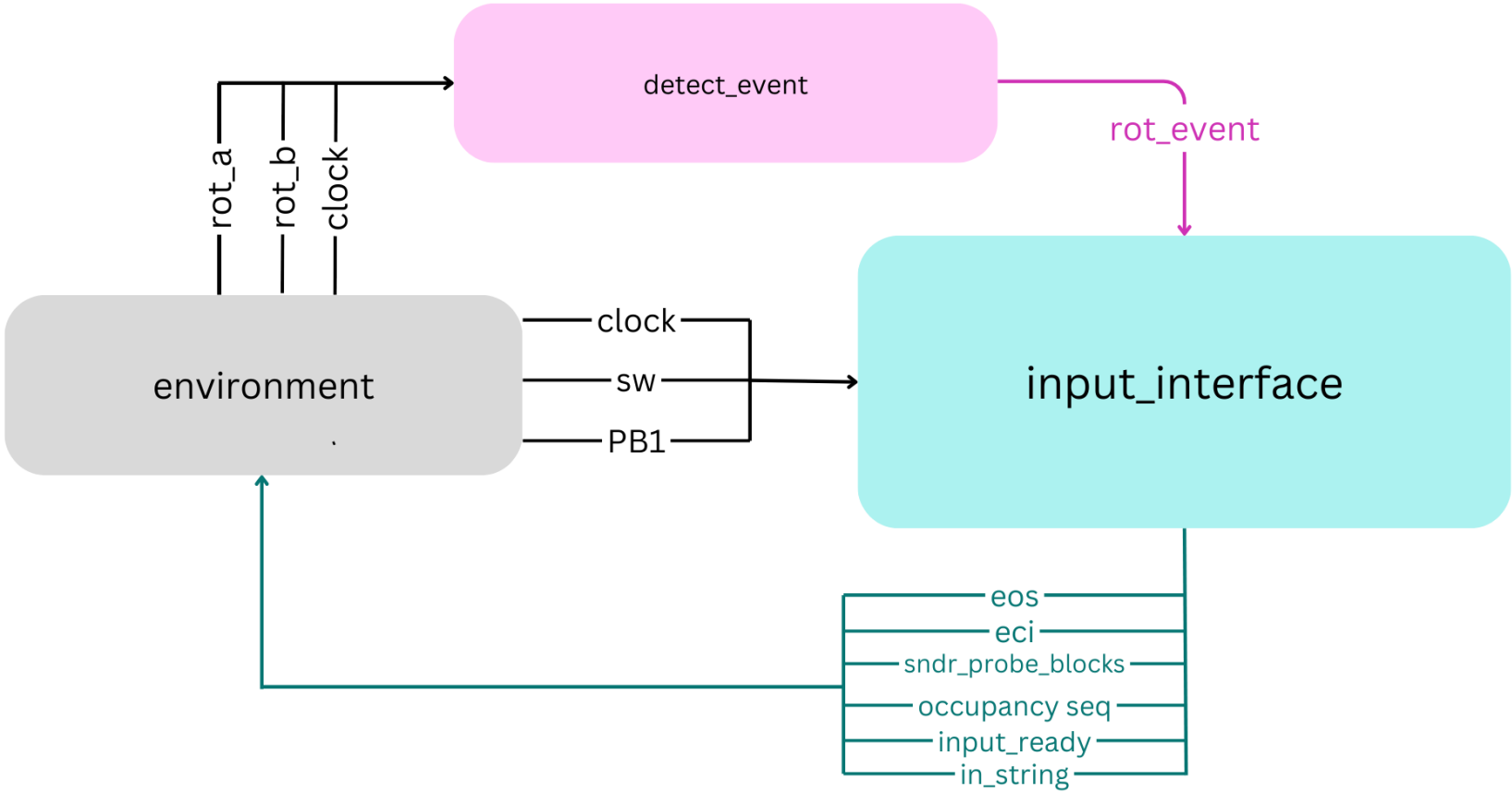


Figure: Input Port Connections

Port Connections for Controller

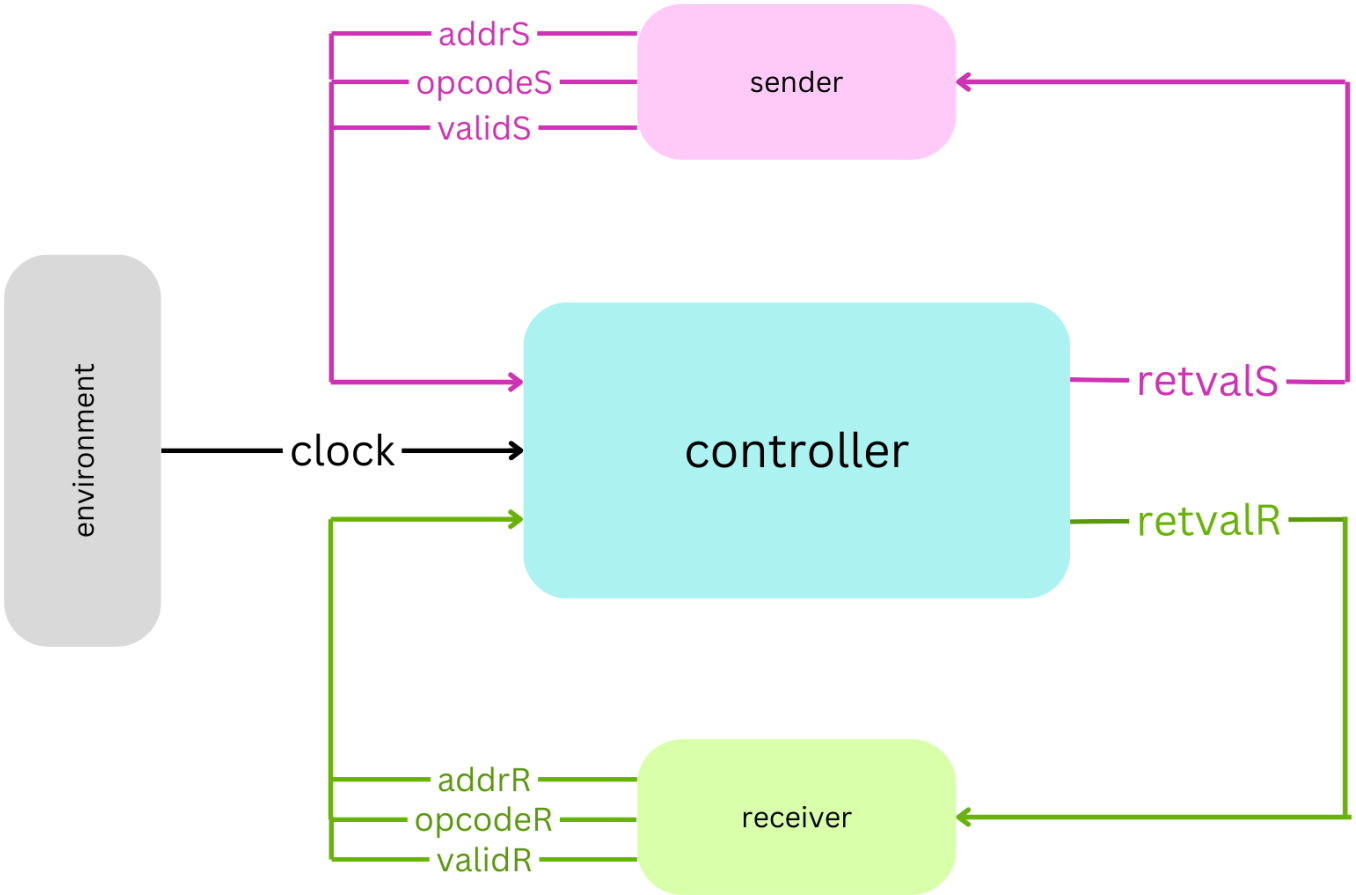


Figure: Cache Controller Port Connections

Port Connections for Sender

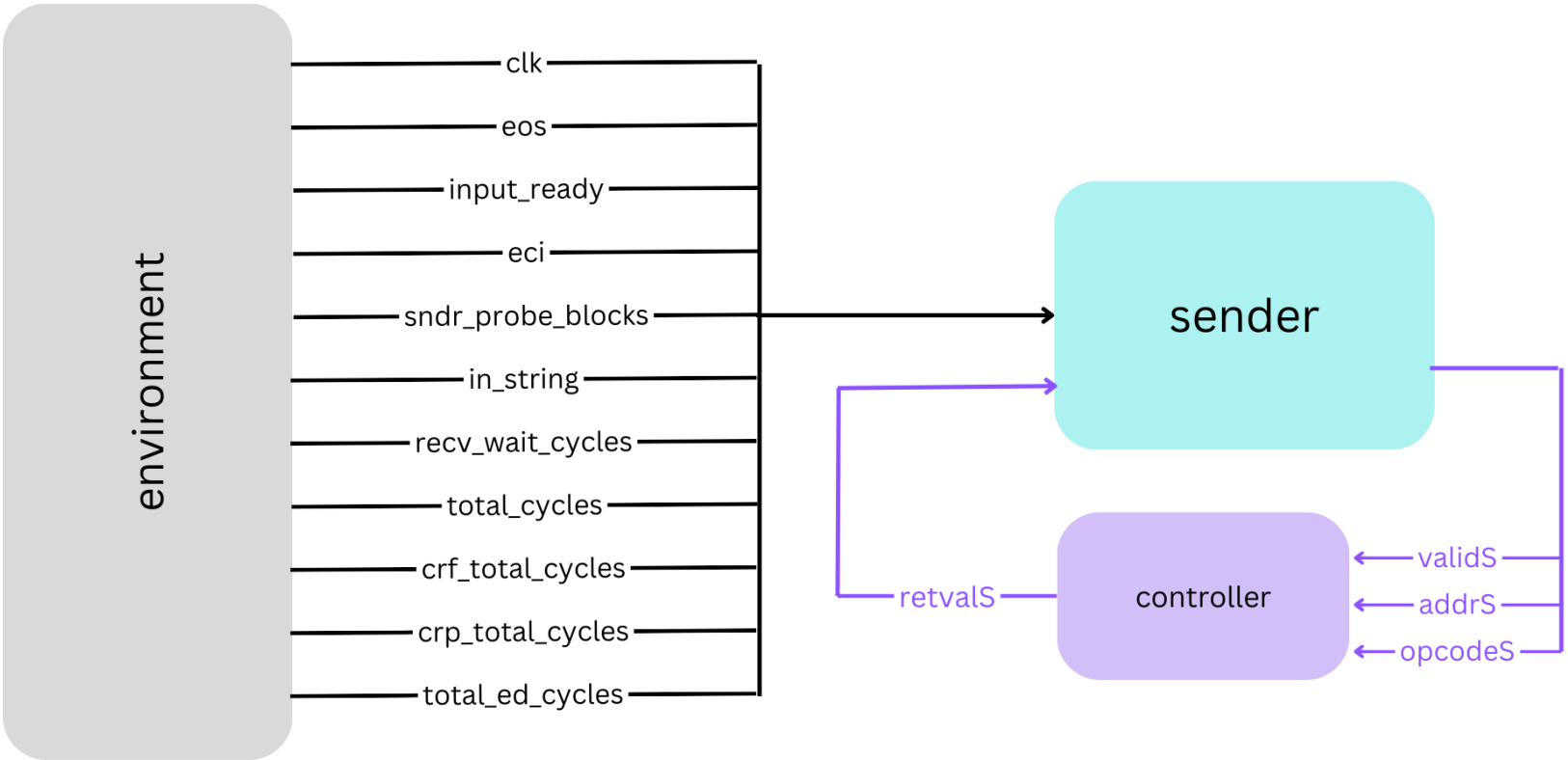


Figure: Sender Port Connections

Port Connections for Receiver

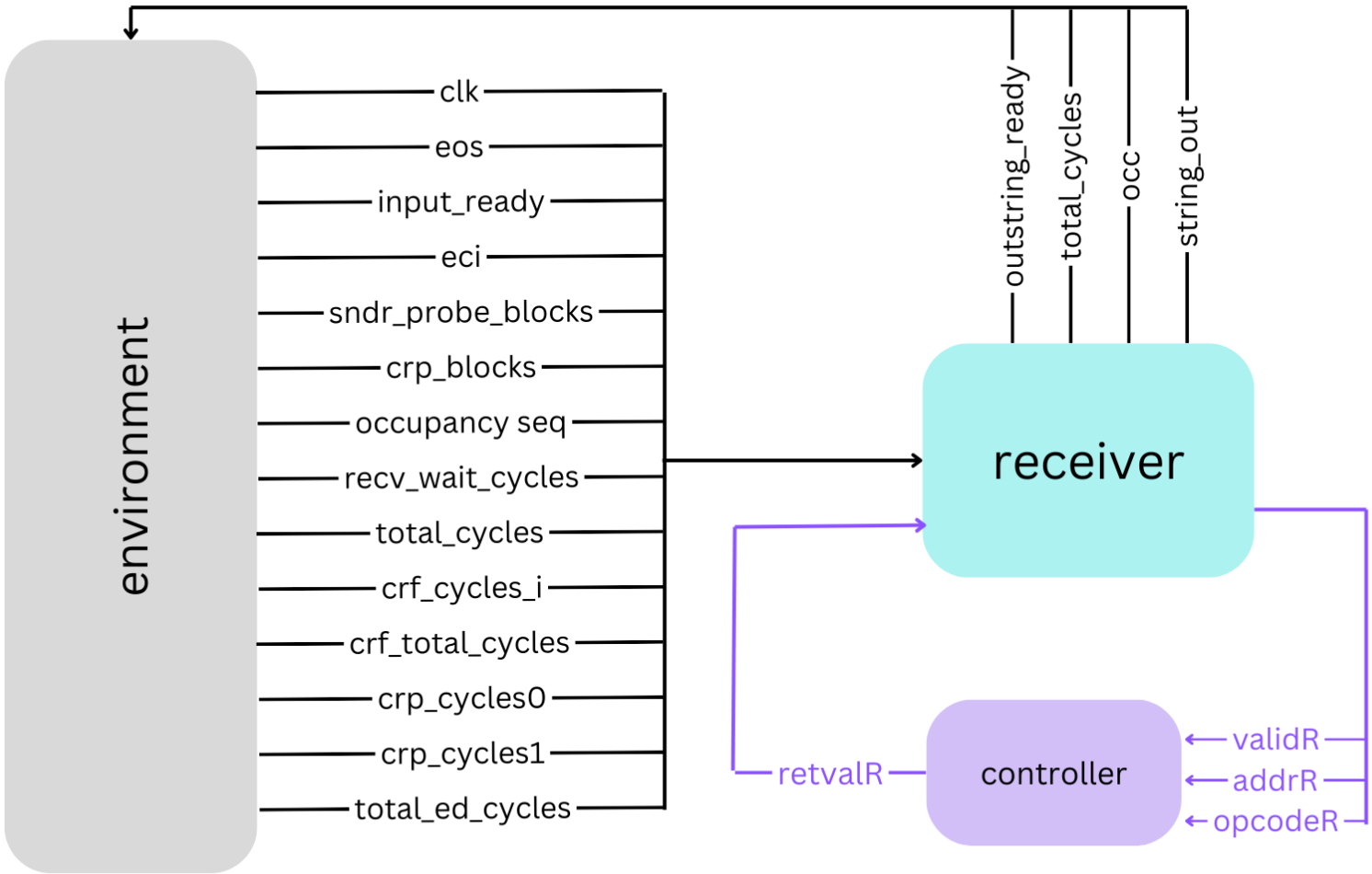


Figure: Receiver Port Connections

Port Connections for Output

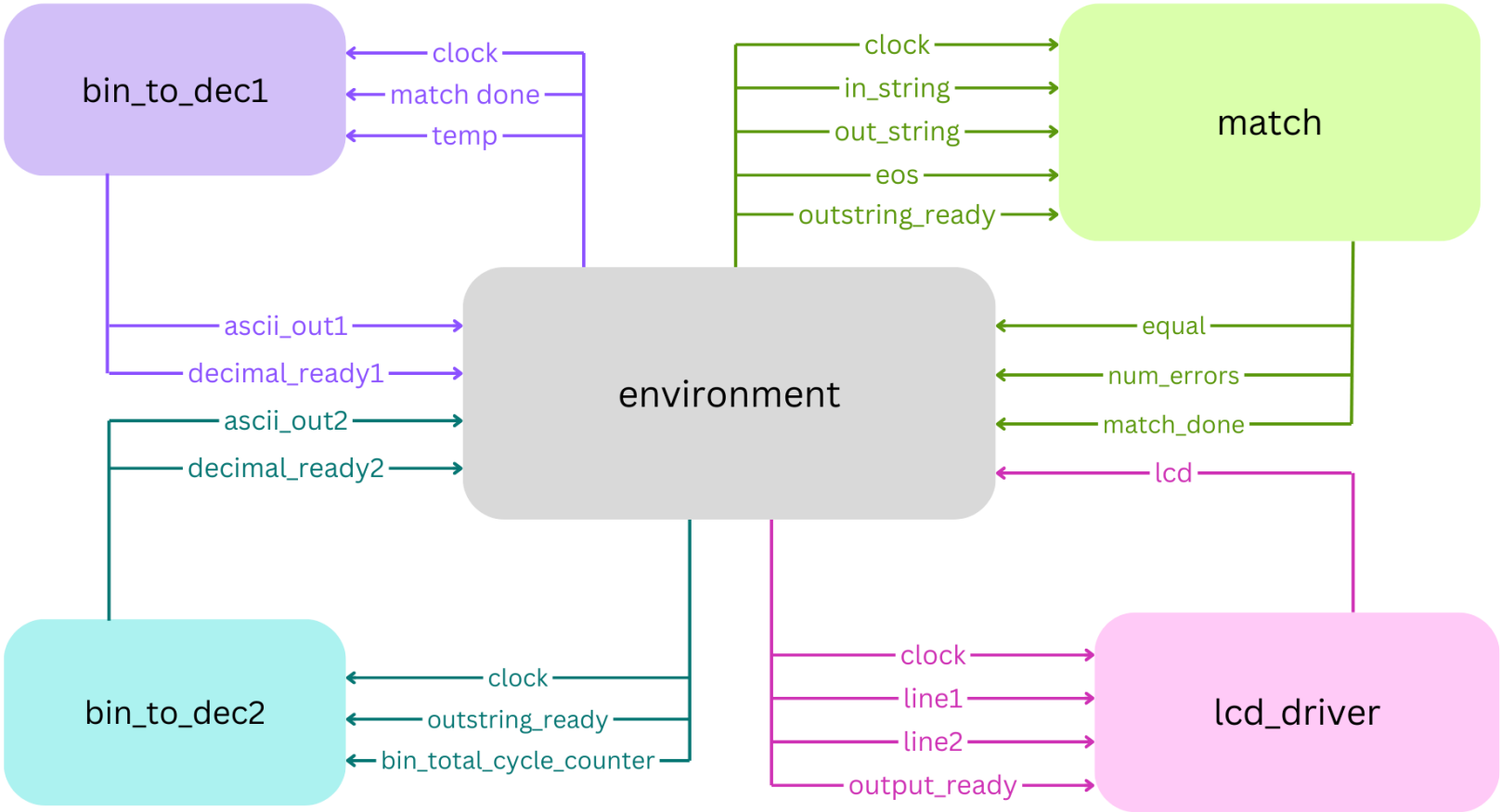


Figure: Output Port Connections