C++ Real-Time Audio Programming with Bela

Dr Andrew McPherson

Centre for Digital Music
School of Electronic Engineering and Computer Science
Queen Mary University of London

Founder and Director, Bela

Course topics

Programming topics

Working in real time
Buffers and arrays
Parameter control
Classes and objects
Analog and digital I/O
Filtering
Timing in real time
Circular buffers
State machines
MIDI
Block-based processing
Threads
Fixed point arithmetic
ARM assembly language

Today

Music/audio topics

Oscillators
Samples
Wavetables
Control voltages
Gates and triggers
Filters
Metronomes and clocks
Delays and delay-based effects
Envelopes
ADSR
MIDI
Additive synthesis
Phase vocoders
Impulse reverb

Lecture 18: Phase vocoder, part 1

What you'll learn today:

Segmenting a real-time signal into windows
Working with overlapping windows (overlap-add)
Multi-threaded audio processing

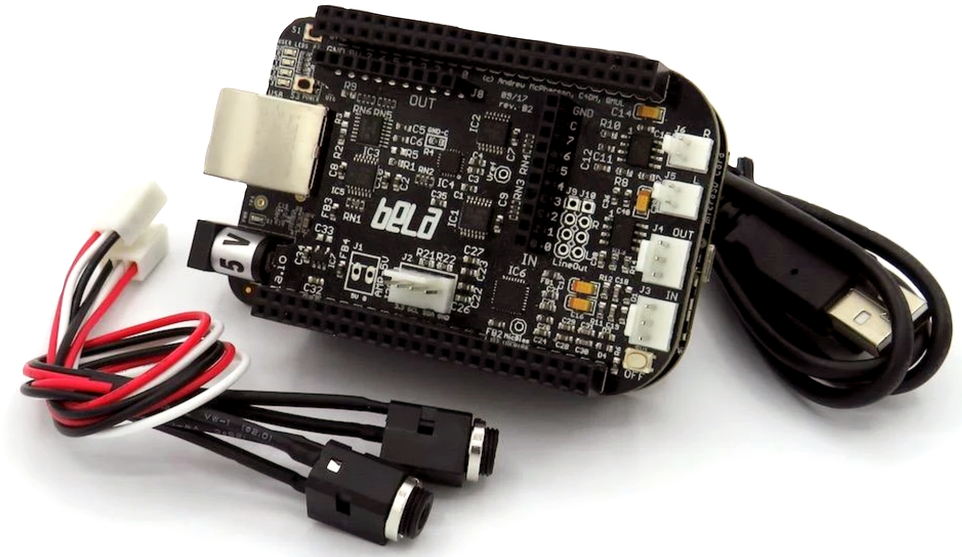
What you'll make today:

A phase vocoder framework with simple effects

Companion materials:

github.com/BelaPlatform/bela-online-course

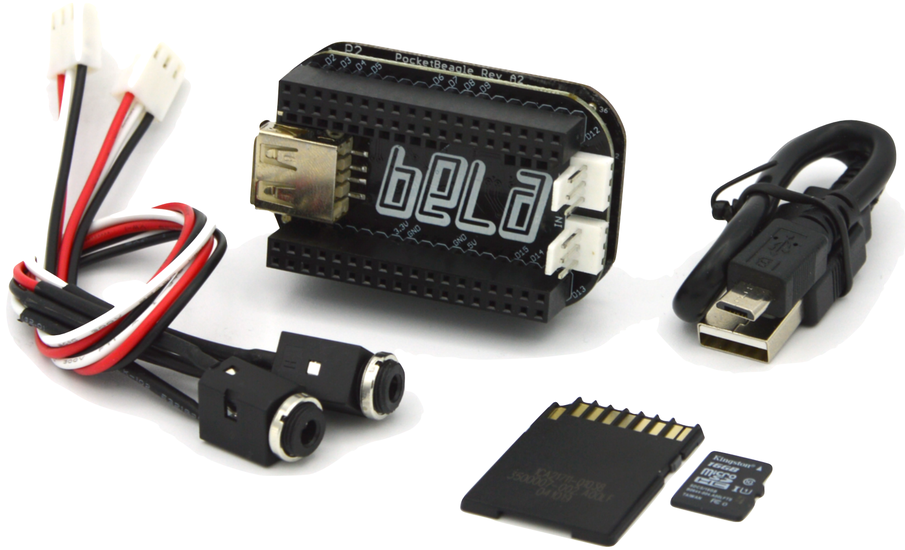
What you'll need



Bela Starter Kit

or

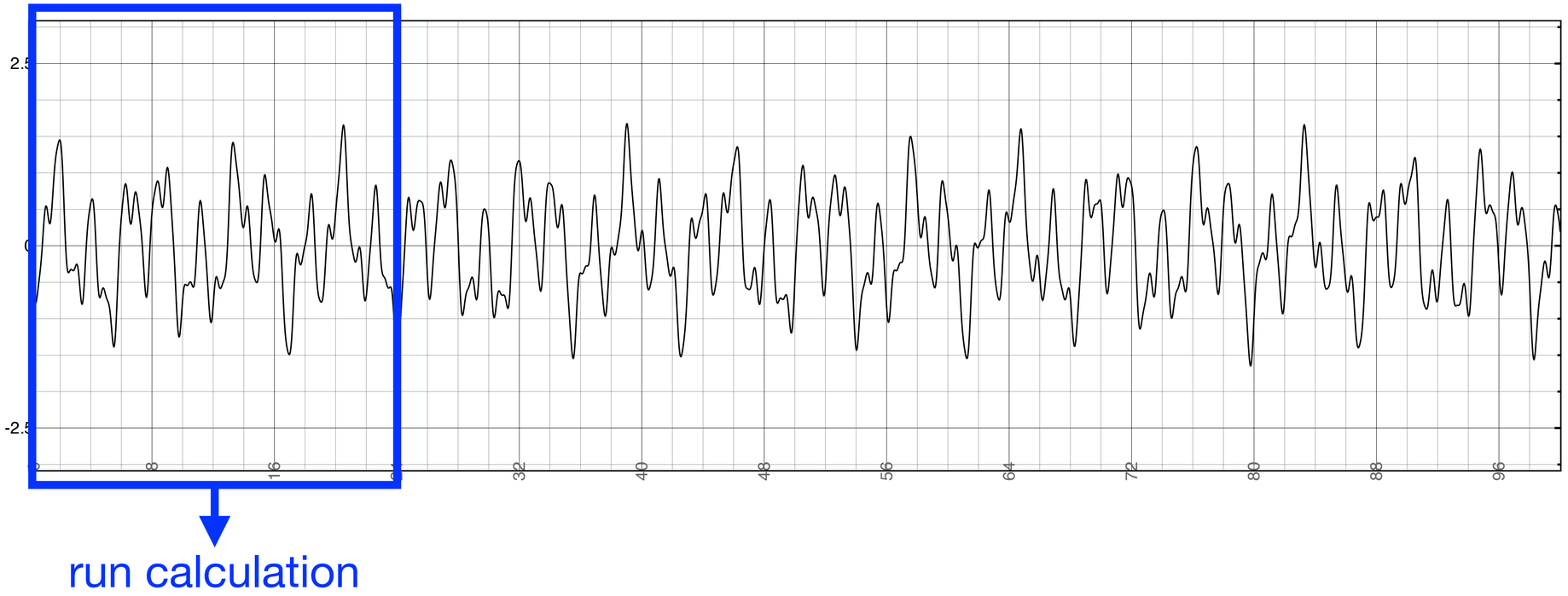
[shop.bela.io]



Bela Mini Starter Kit

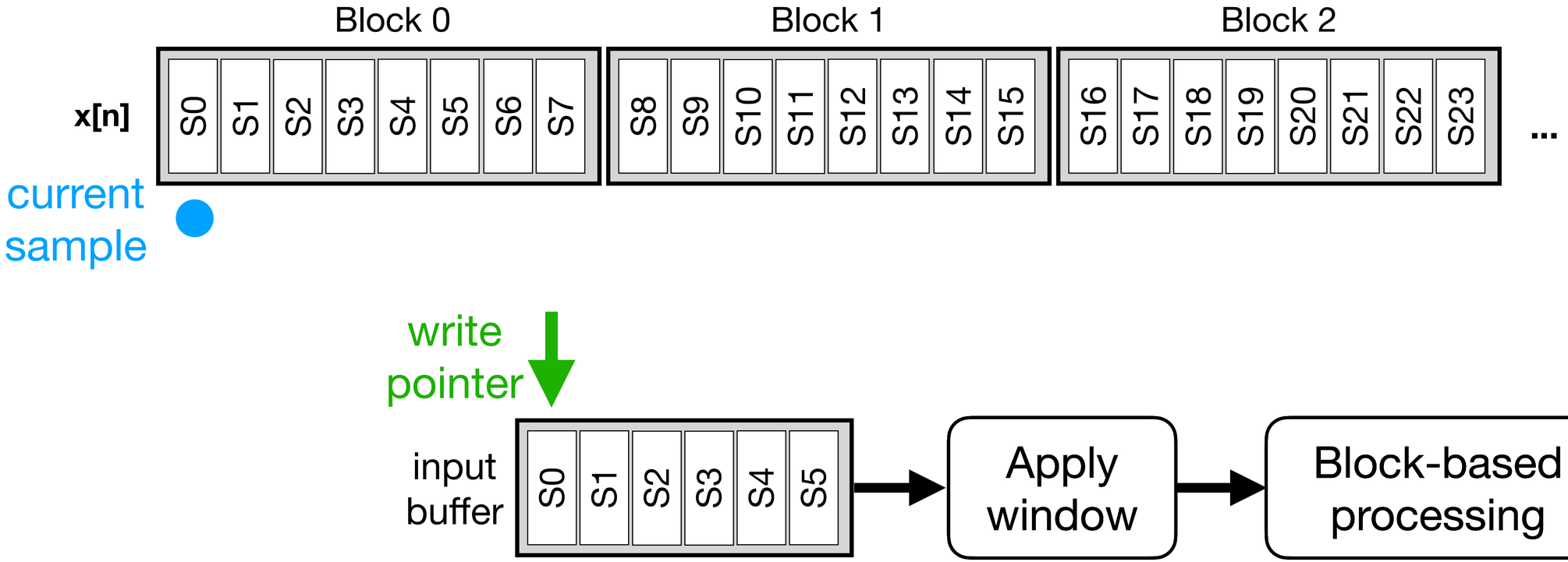
Block-based processing

- Sometimes audio calculations need to run on a block of samples simultaneously
  - For example: wait until 512 samples arrive, then do something with all of them at once



- Compare this to filters, where we calculate a new output every sample
  - With block-based processing, we typically run the calculation periodically, not every sample

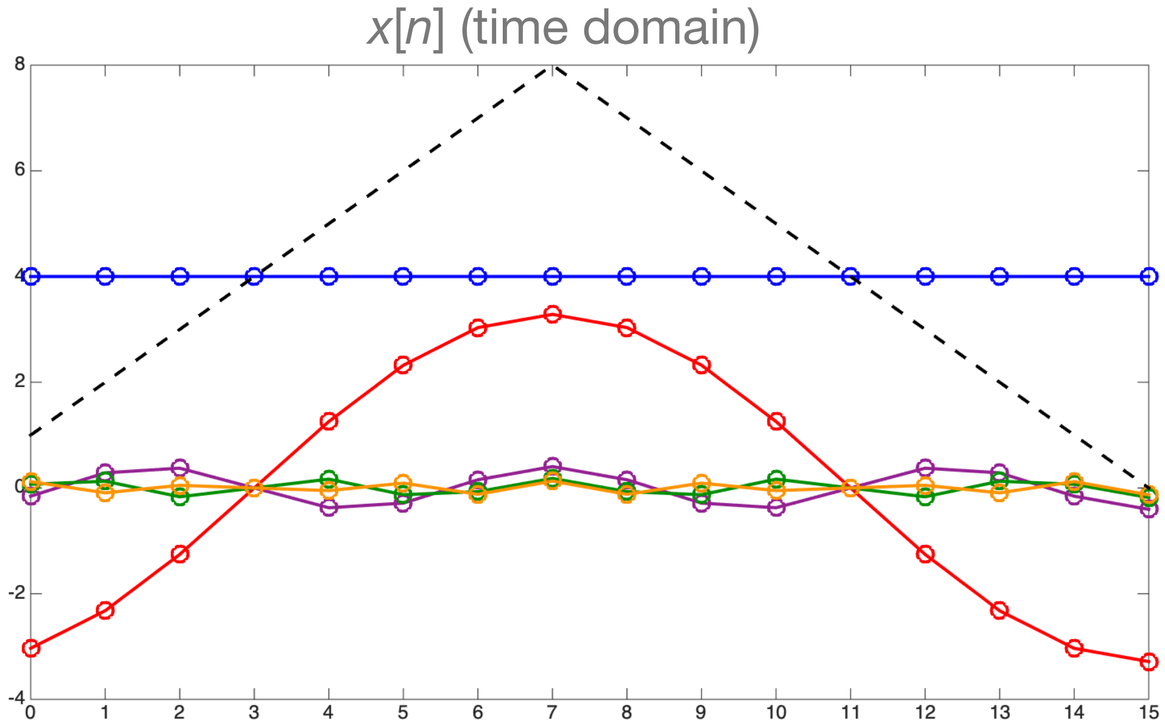
Block-based input



- Start with a buffer of fixed size
  - Fill it up sample by sample
  - When it's full, apply the window function and pass it to the FFT (or other block based process)

The Fast Fourier Transform

- Most common use of block-based processing: the Fast Fourier Transform (FFT)
  - Efficient computational algorithm for calculating the Discrete Fourier Transform (DFT)
  - The DFT is a mathematical formula for calculating the frequency content of a signal
- Any discrete-time signal of N points can be expressed as the sum of N sinusoids:



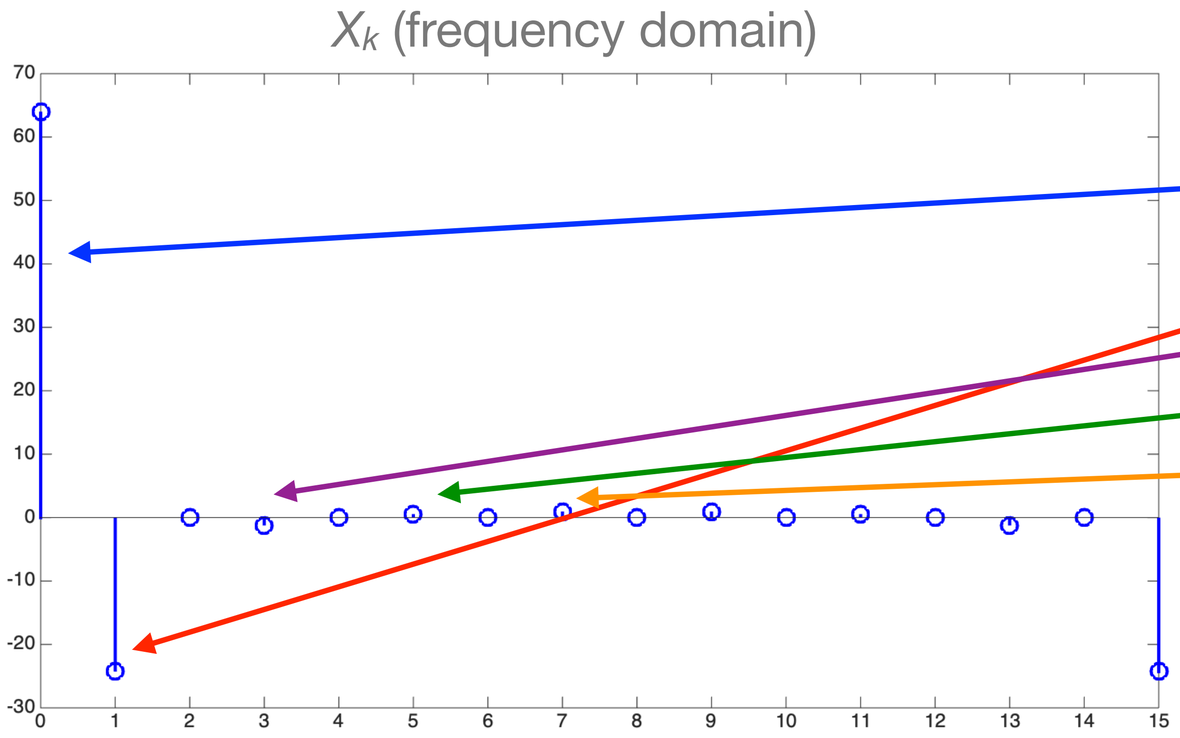
Frequency	Amplitude	Phase
0	4.00	0
π/8	1.64	-.875*π
3π/8	0.20	-.625*π
5π/8	0.09	-.375*π
7π/8	0.65	-.125*π

(other components are 0 or redundant)

The DFT calculates the amplitudes and phases of each frequency component

The Fast Fourier Transform

- Most common use of block-based processing: the Fast Fourier Transform (FFT)
  - Efficient computational algorithm for calculating the Discrete Fourier Transform (DFT)
  - The DFT is a mathematical formula for calculating the frequency content of a signal
- Any discrete-time signal of N points can be expressed as the sum of N sinusoids:



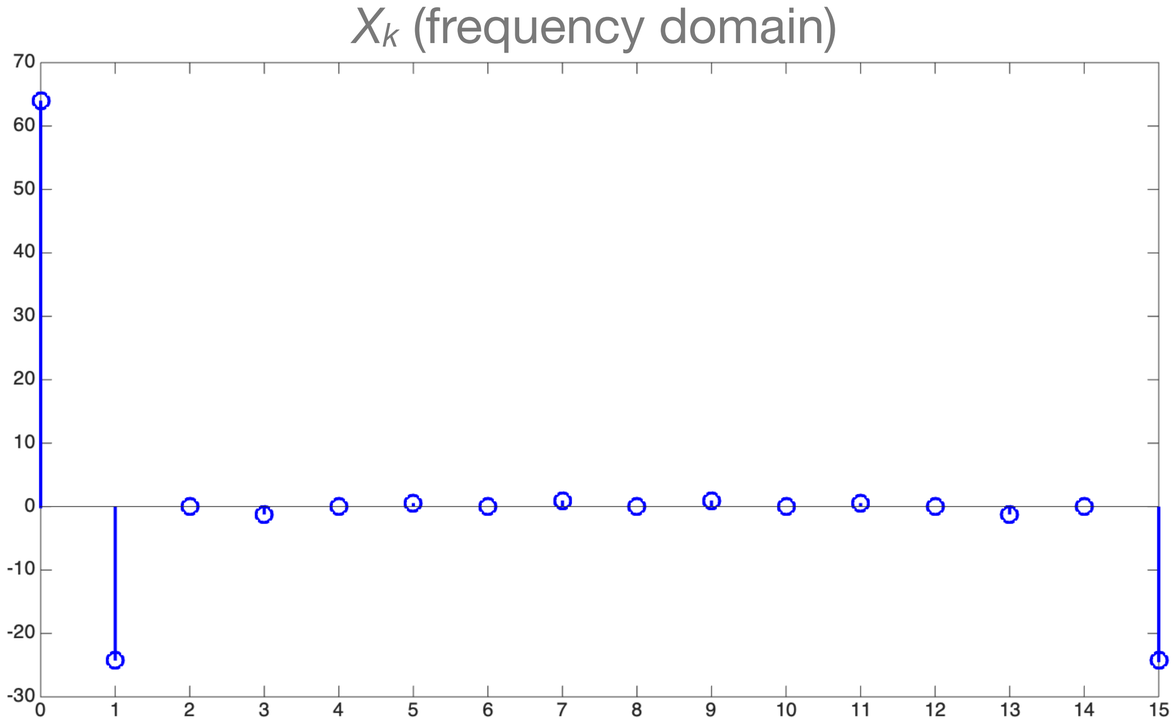
Frequency	Amplitude	Phase
0	4.00	0
π/8	1.64	-.875*π
3π/8	0.20	-.625*π
5π/8	0.09	-.375*π
7π/8	0.65	-.125*π

(other components are 0 or redundant)

The DFT calculates the amplitudes and phases of each frequency component

The Discrete Fourier Transform

- The DFT calculates the amplitudes and phases of each sinusoidal component
  - The frequencies are linearly spaced between 0 and 2π: $\frac{2\pi k}{N}$ for $0 \leq k < N$
  - Each amplitude/phase measurement is called a bin
  - Collectively, the N bins are called the frequency domain representation of the signal



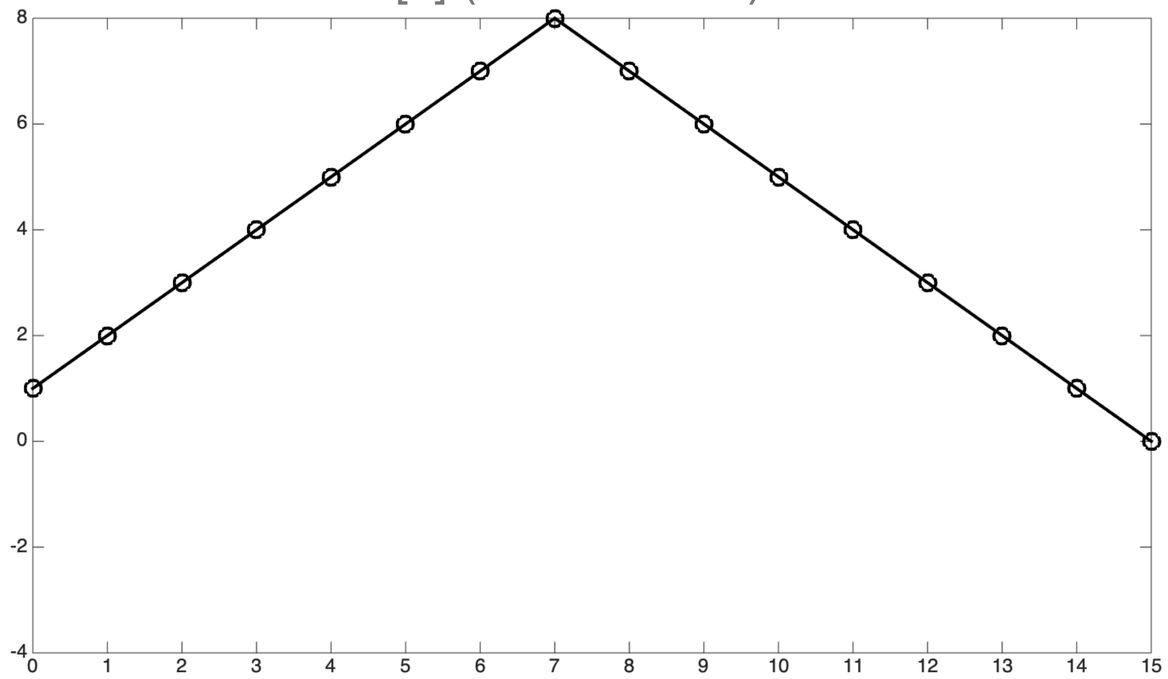
Frequency	Amplitude	Phase
0	4.00	0
π/8	1.64	-.875*π
3π/8	0.20	-.625*π
5π/8	0.09	-.375*π
7π/8	0.65	-.125*π

(other components are 0 or redundant)

The Discrete Fourier Transform

- Key point: N time samples ↔ N frequency samples
  - Two different mathematical perspectives on the same signal
  - The Inverse Discrete Fourier Transform (IDFT) converts from frequency domain to time domain
  - The DFT+IDFT is an exact reconstruction as long as signal length $M \leq$ DFT length N

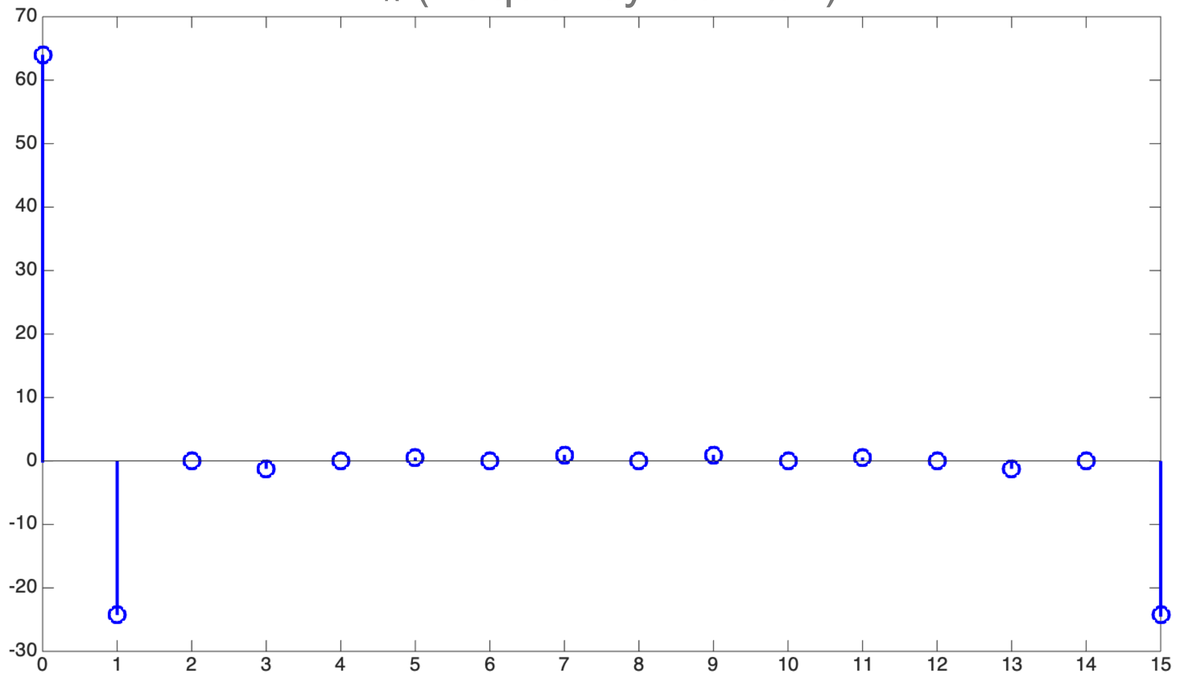
$x[n]$ (time domain)



DFT

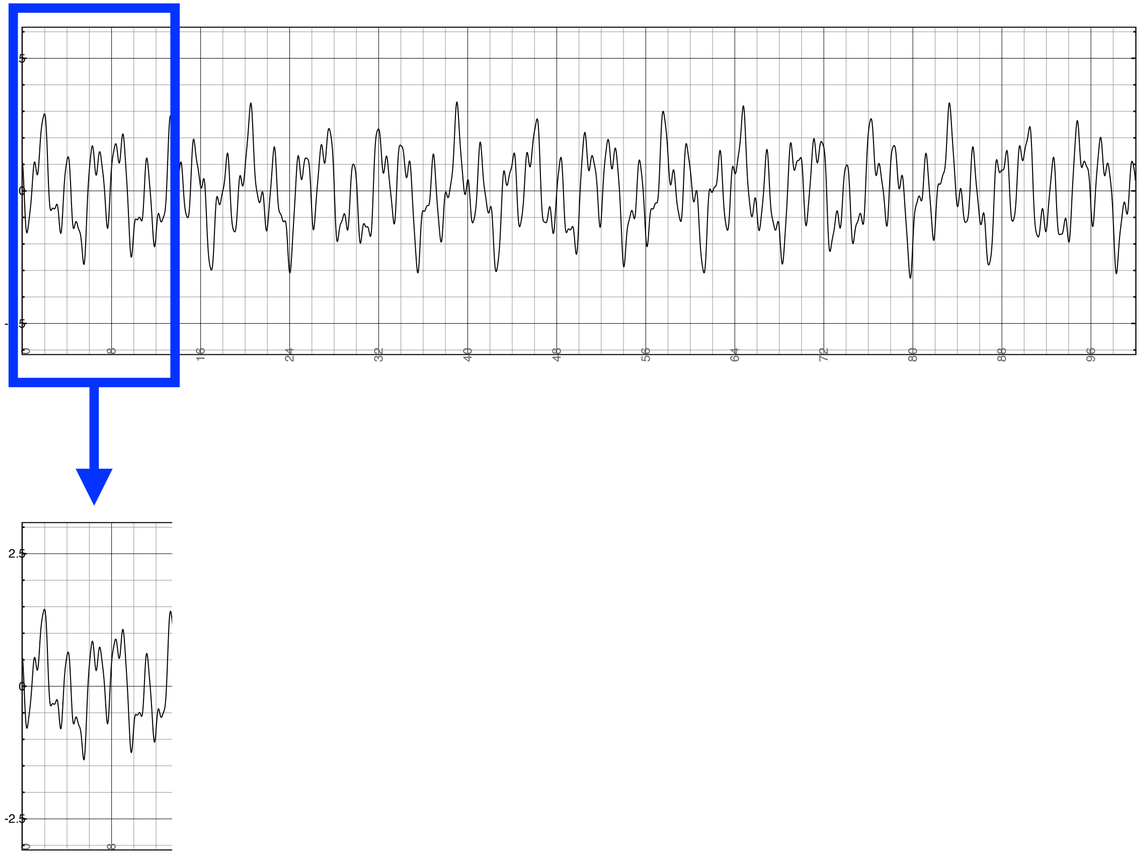
IDFT

X_k (frequency domain)



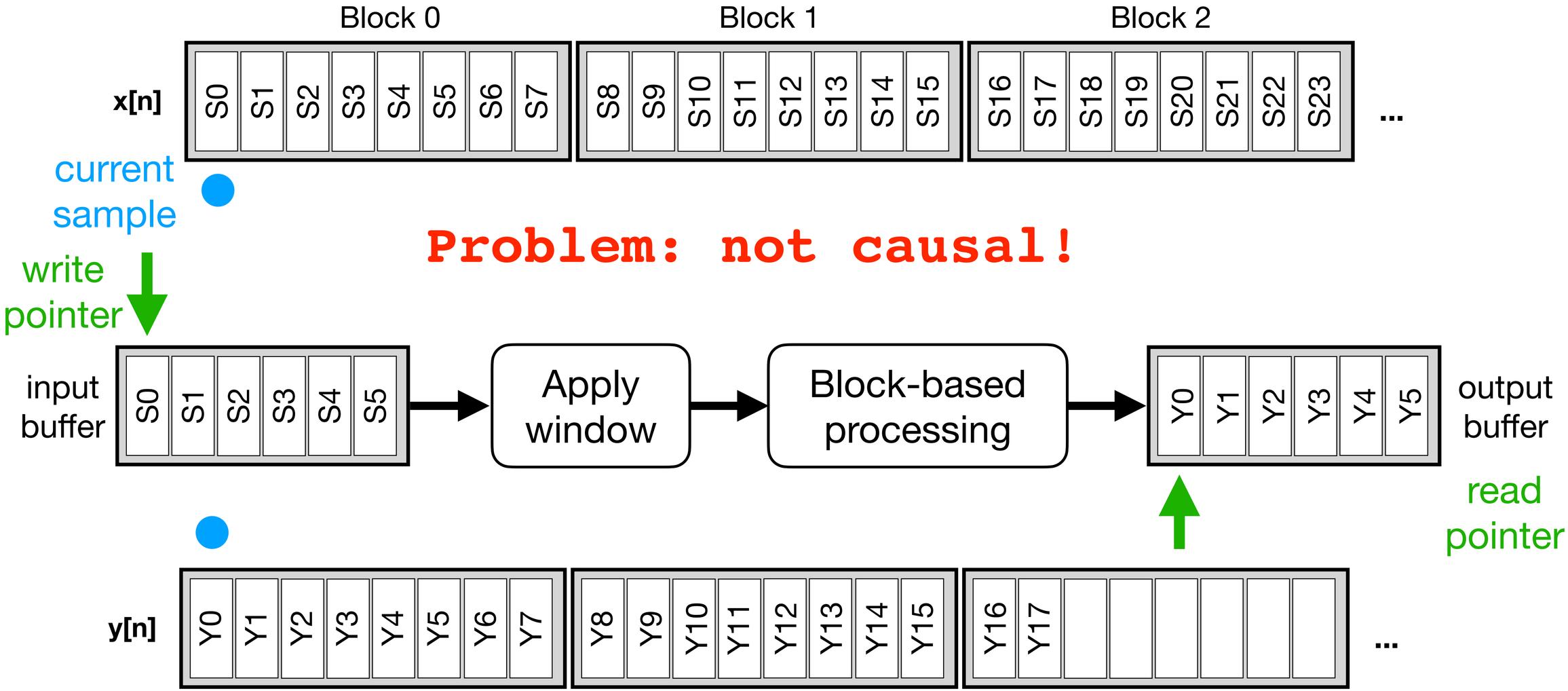
Block-based input and output

- After we divide our signal into windows, we often want to reconstruct it
  - Do some processing on each block
  - Then stitch the processed blocks back together into a new signal
  - In the ideal case, if we don't do any processing on each block, we should be able to reconstruct the signal exactly
- Application: phase vocoder
  - Take an FFT on each block
  - Do processing in the frequency domain using the phase of the bins for a more accurate frequency estimate (more soon!)
  - Take the IFFT on each block



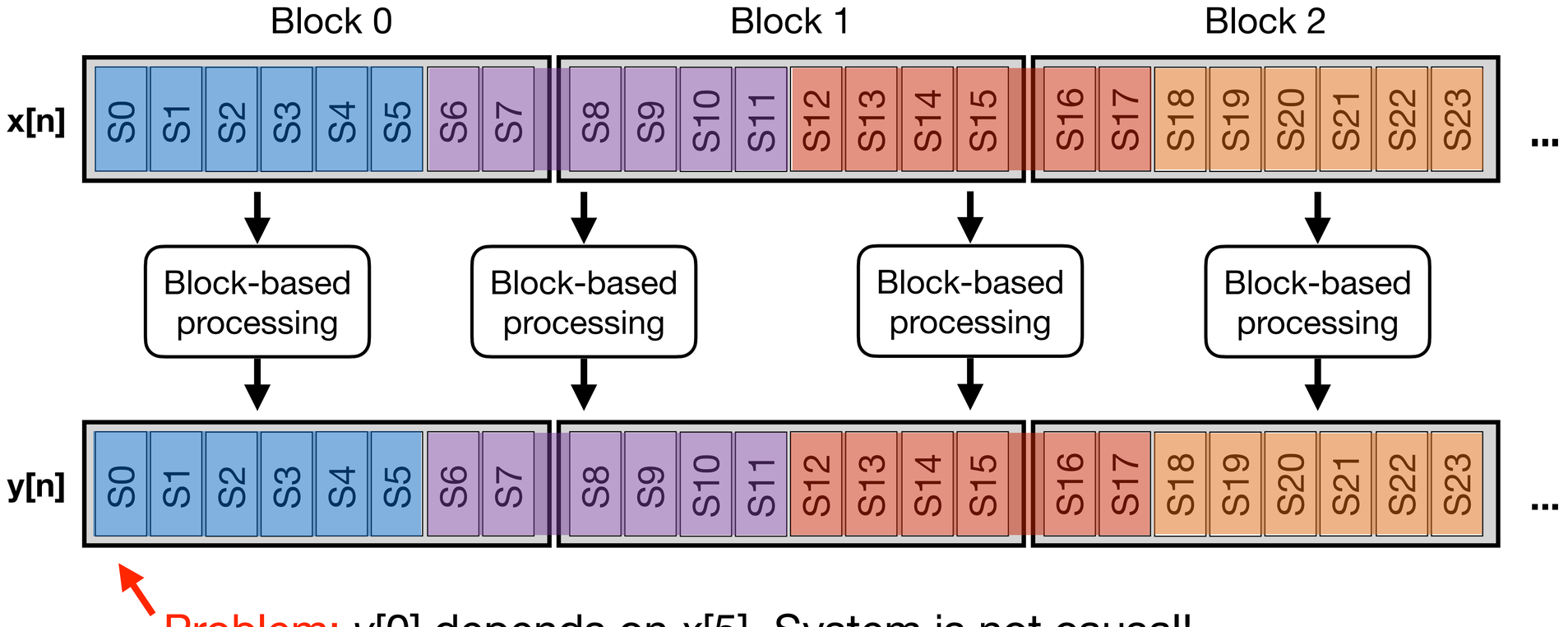
Block-based input and output

- Our ideal block-based processing system might work something like this:



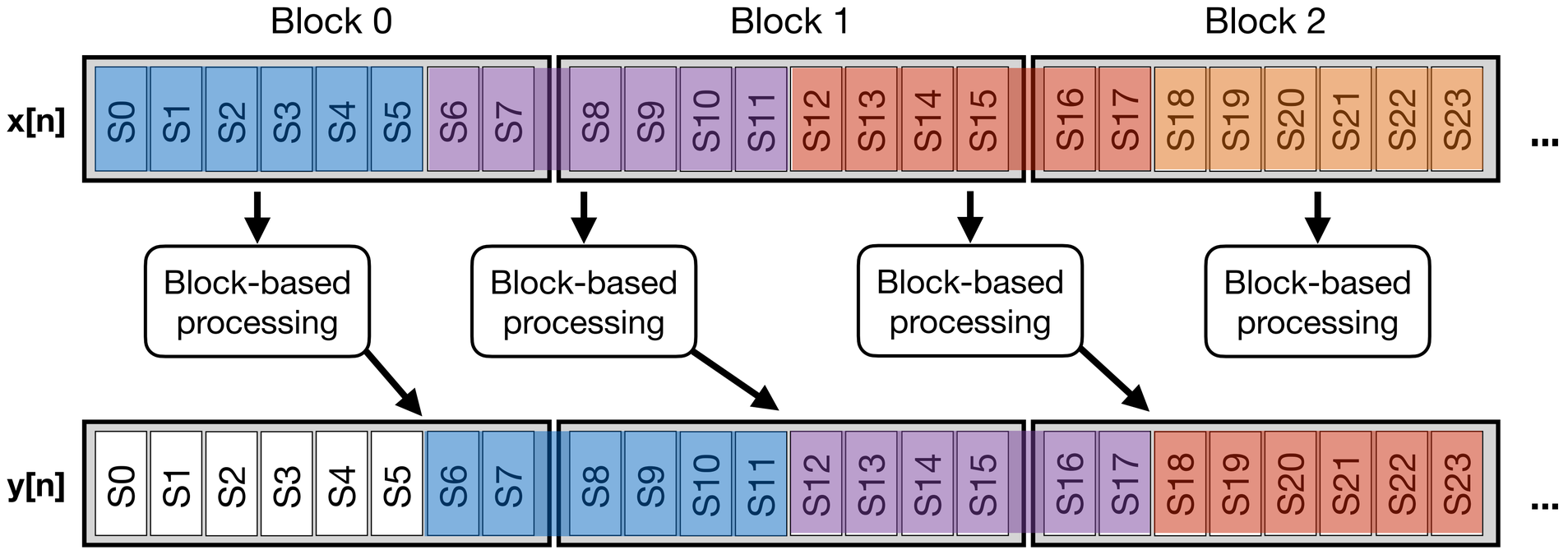
Block-based output: causality

- To have a causal system, an output cannot depend on future inputs



Block-based output: causality

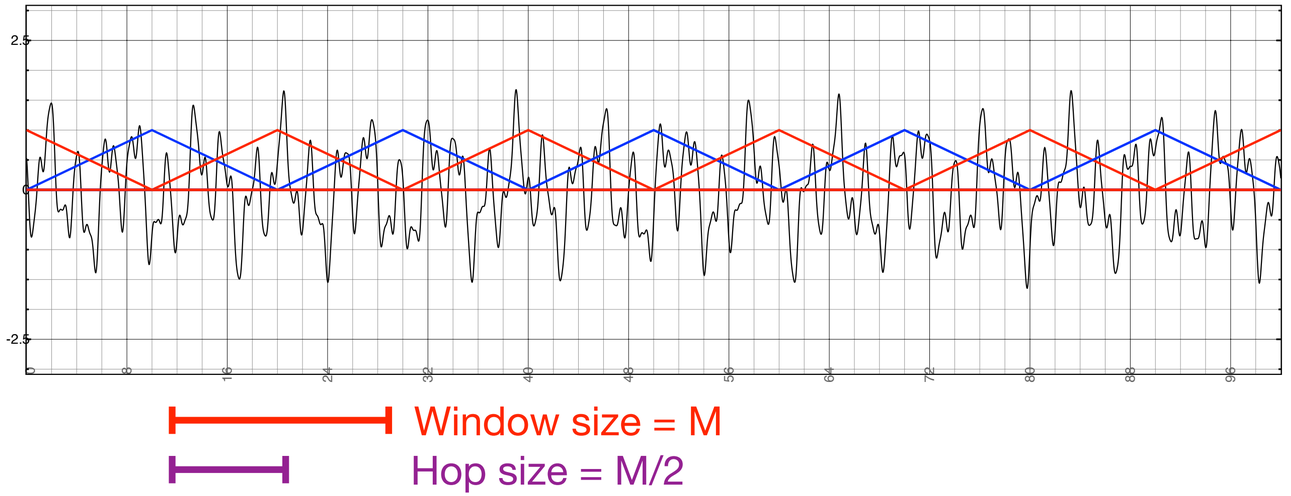
- To have a causal system, an output cannot depend on future inputs



- Partial solution: deliberately add a 1-window latency to the output
  - As we gather input samples for the next FFT calculation, we are copying samples from the last FFT into the output

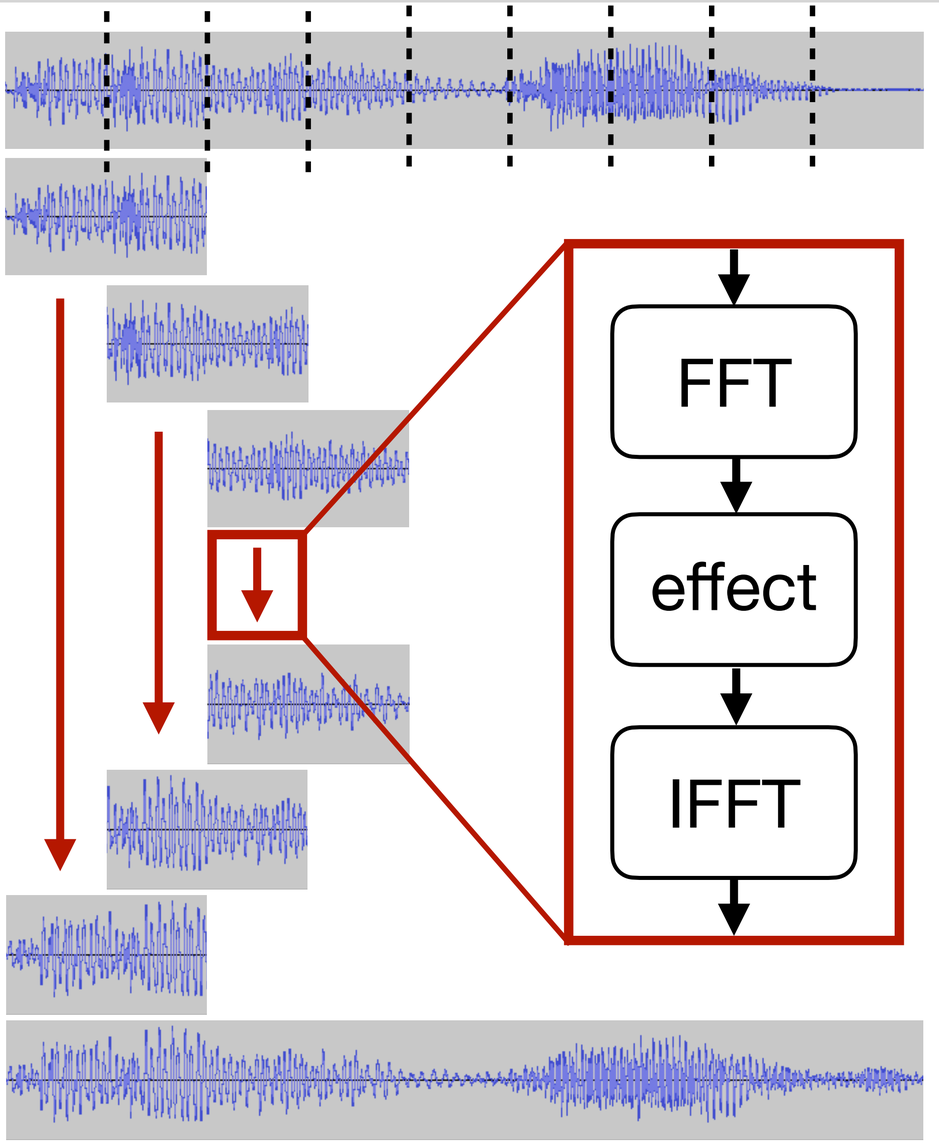
Overlapping blocks

- The previous example performed a block calculation at regular intervals
  - Once a full block of samples was assembled, it was processed (e.g. using an FFT)
  - The larger the window (block) size, the less frequent the calculations
  - In that example, each input sample was part of exactly one block
- It can be useful to have blocks that overlap one another
  - For example, use larger FFTs but calculate them more often
  - The number of samples between the start of consecutive blocks is called the hop size
  - This is often a fraction of the block size (or window size)
    - For example, FFT size 1024, hop size 512
- We might apply a window function before calculating the FFT
  - In this example, a triangular window

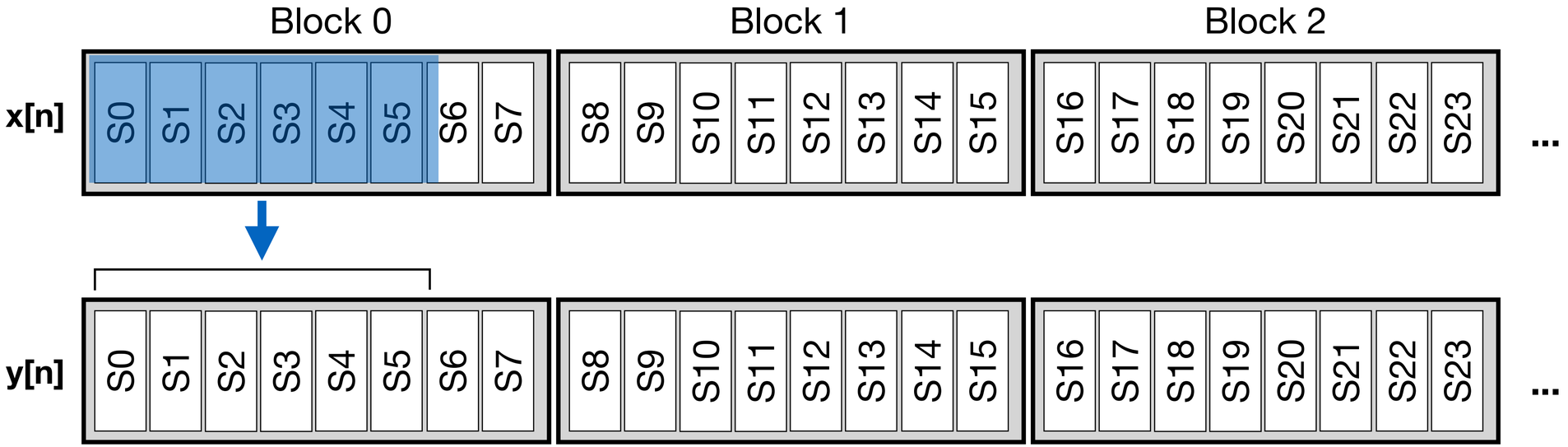


Overlap-Add

- A standard approach for block-based processing with overlapping blocks is called overlap-add:
  1. Isolate a block of length M using a windowing function
  2. Take an FFT of length N ≥ M of the segment
     - If N > M, zero-pad the block (add zeros to end of the window)
  3. Do something interesting to frequency-domain data
  4. Take an IFFT to get a new time domain segment
  5. Add the segment to an output buffer which also contains the preceding segments
     - As we'll see, we can't write the segment directly to the audio output
  6. Advance by the hop size (H) to the next frame and repeat
     - In real time, count samples until H more have arrived
     - Hop size H is less than window size M, hence the **overlap**

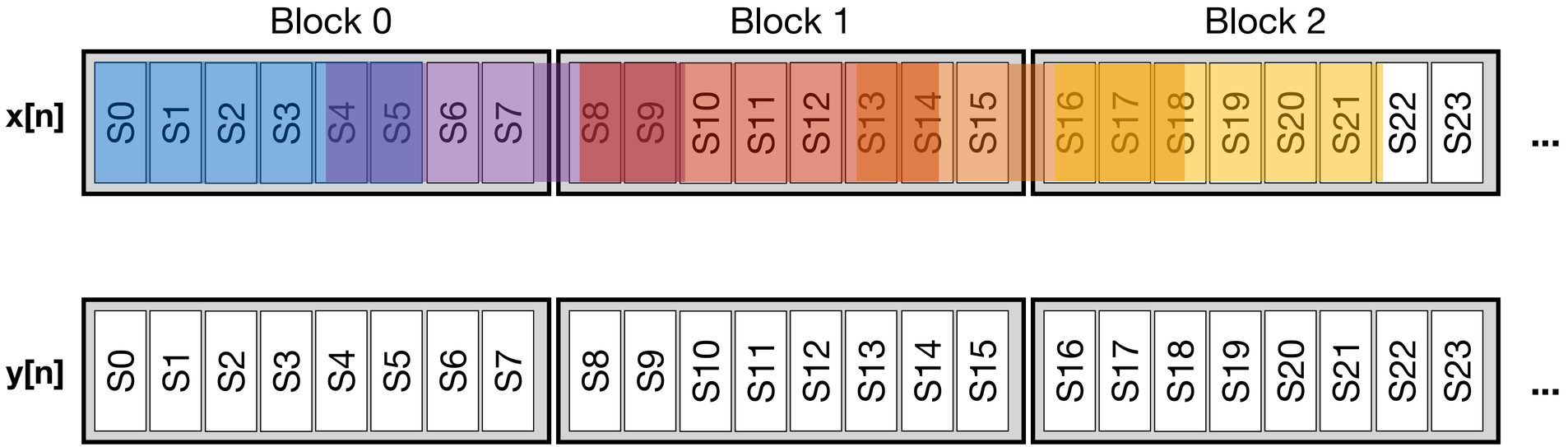


Review: overlapping blocks (input side)



- The goal is to work with overlapping blocks in real time
  - This means block *k* shares some samples with block *k*-1
  - How do we adapt our windowing code to handle this?
- Easiest approach: keep a running history of the input samples
  - i.e. a buffer which always has the last M samples
  - What kind of structure have seen that does this? Circular buffer
  - At each hop, pass M samples from circular buffer to FFT

Review: overlapping blocks (input side)

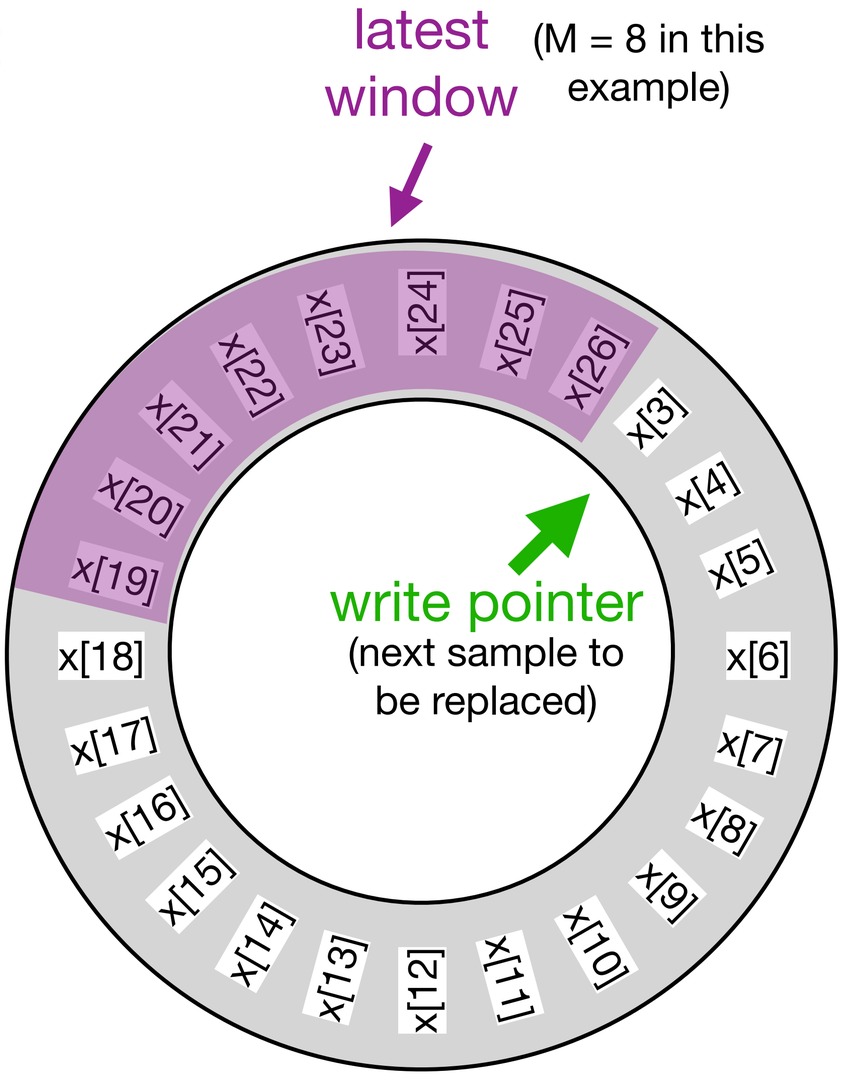


- The goal is to work with overlapping blocks in real time
  - This means block *k* shares some samples with block *k*-1
  - How do we adapt our windowing code to handle this?
- Easiest approach: keep a running history of the input samples
  - i.e. a buffer which always has the last M samples
  - What kind of structure have seen that does this? Circular buffer
  - At each hop, pass M samples from circular buffer to FFT

Review: overlap-add with a circular buffer

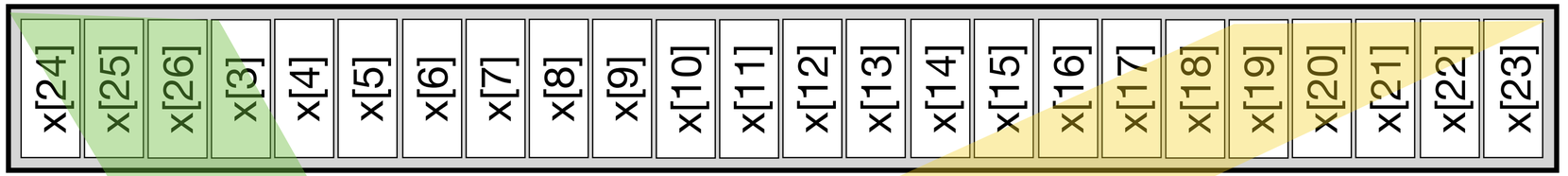
- We use a circular buffer to keep track of the input
- Necessary characteristics:
  - The buffer always holds (at least) the last M input samples (where M is the window size)
  - If using multiple channels, one buffer per channel
- Using the circular buffer:
  - Each iteration of the `for()` loop:
    - store input in buffer
    - increment the write pointer (wrapping as necessary)
    - increment the total count of samples stored
  - When the count reaches the hop size:
    - take one window from buffer, unwrap it, and pass it to FFT
    - in other words: copy it to a new buffer such that the oldest sample appears at index 0 of the new buffer

Important: this is not the same as when the write pointer wraps around!

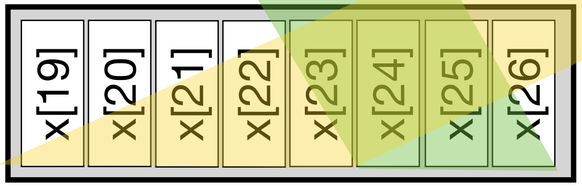


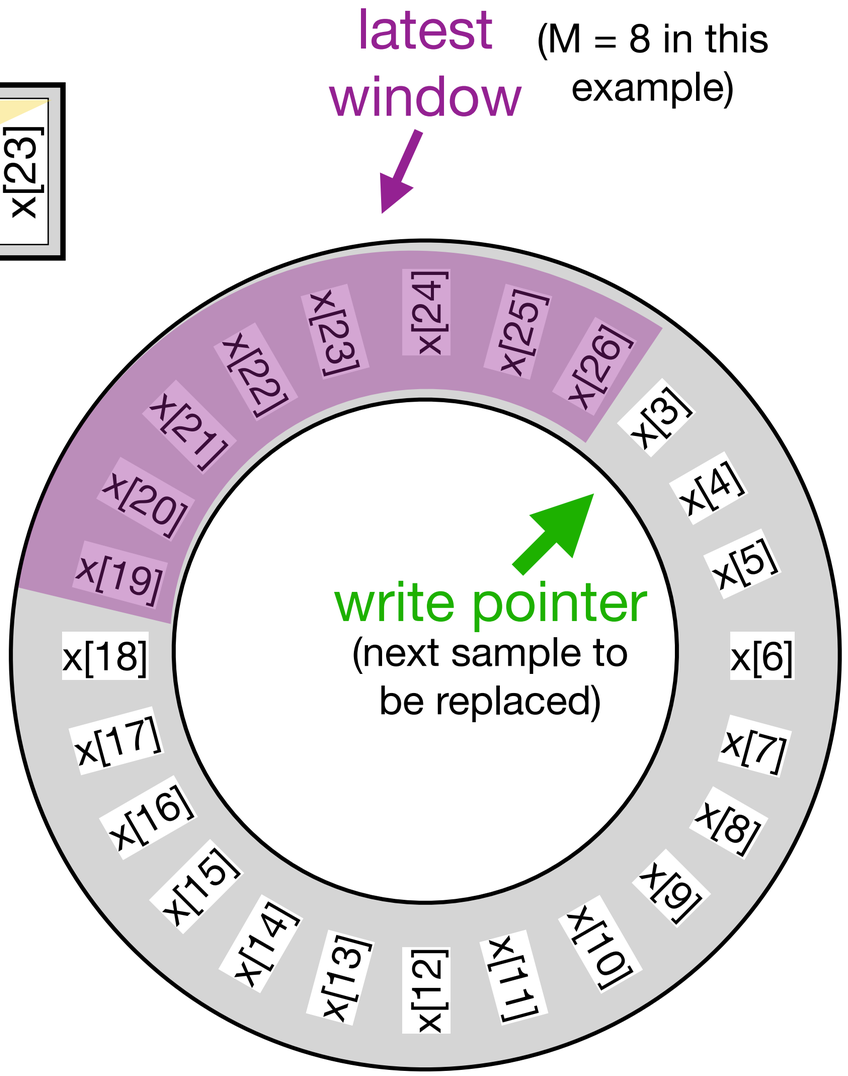
Review: unwrapping the circular buffer

How the circular buffer is actually stored in memory:



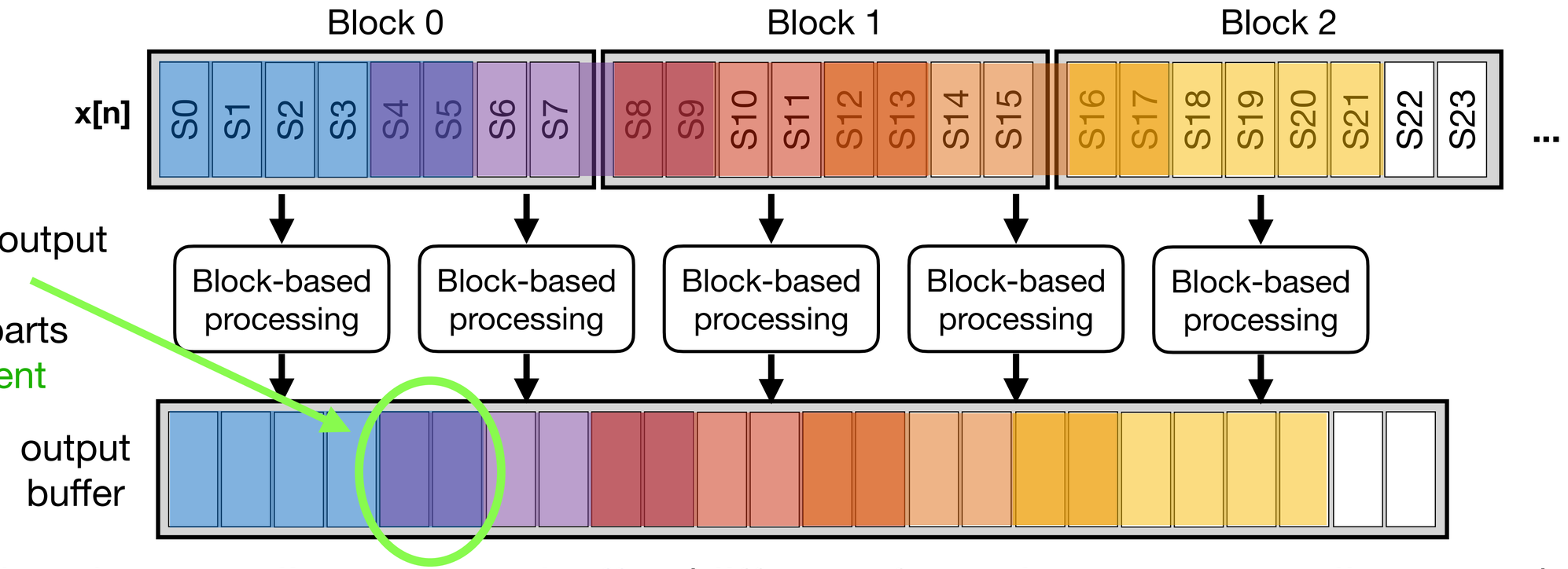
What we need to calculate the FFT (the latest 8 samples):





In other words: having assembled the samples into a circular buffer, on each hop it is helpful to copy them into a linear (standard) buffer to pass to the FFT library

Block-based output: overlap

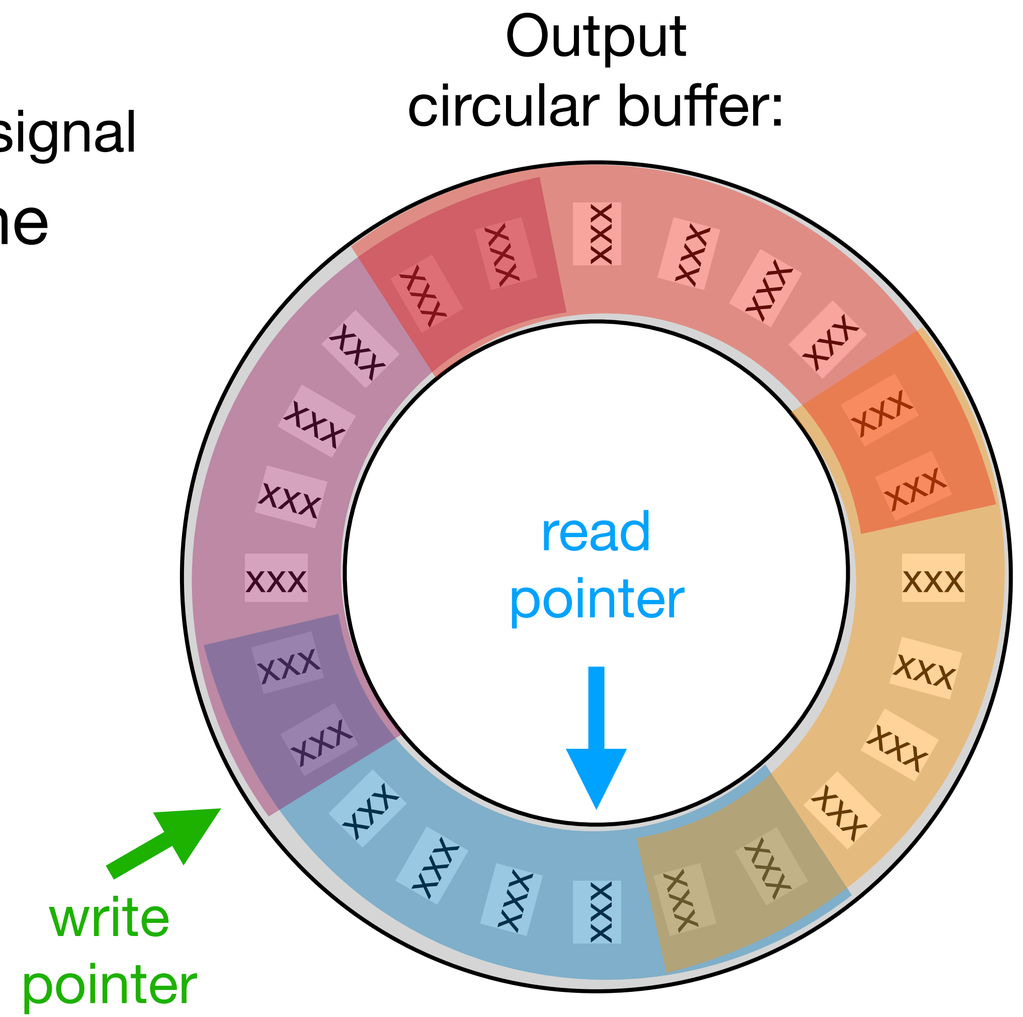


Note: these output samples are the sum of parts of two different blocks

- We need an intermediate output buffer (different from the system audio output) to hold the results of our block-based calculation
  - Each window, the new output should be added into that buffer
  - A sample can be sent from there to the system output as soon as all the blocks it depends on have been calculated (this implies a latency of at least 1 window)

Output circular buffer

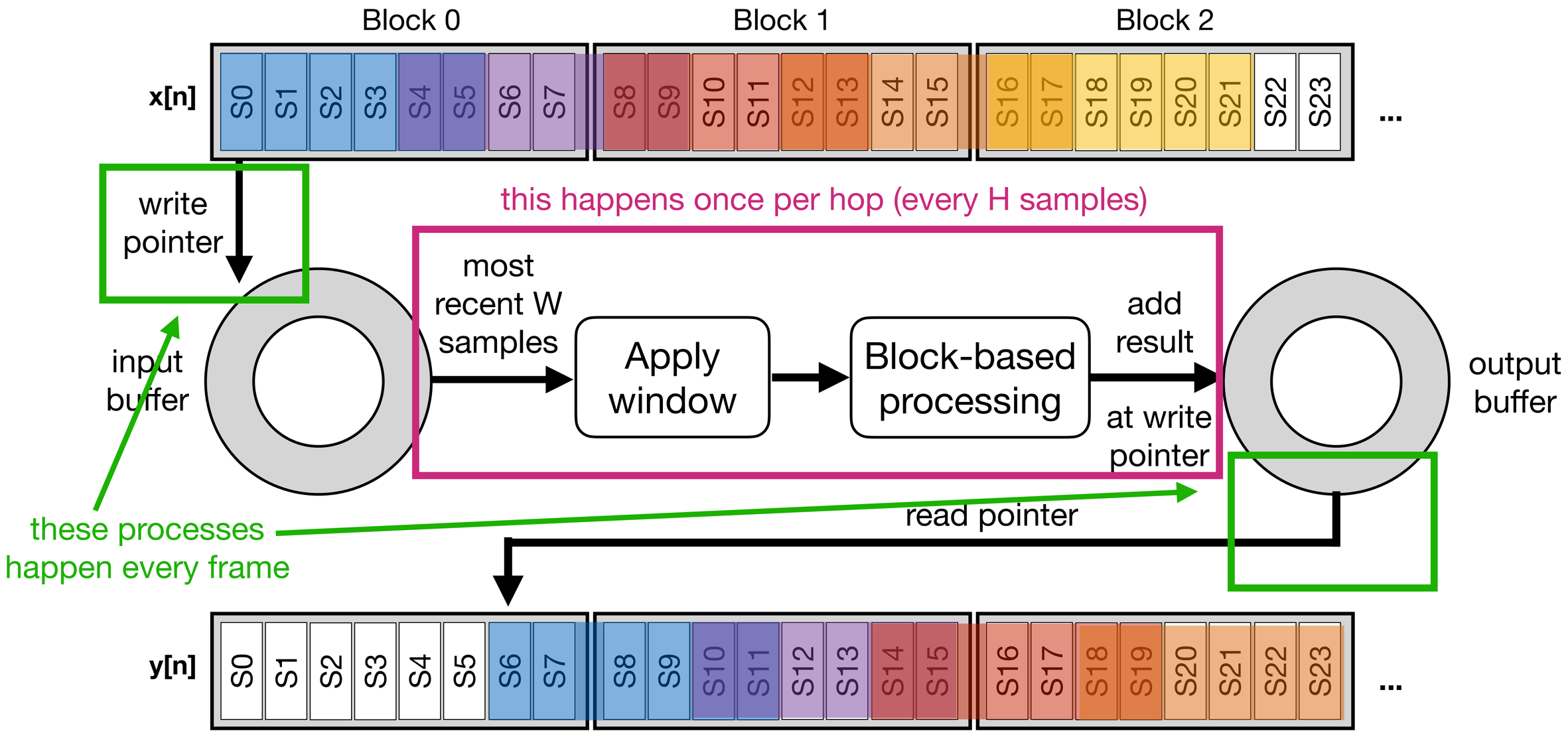
- We will use another circular buffer to gather output from the overlap-add
  - Not the same as the circular buffer of the input signal
- We need to keep track of two pointers in the circular output buffer
  - Write pointer: where does the next block get written once it's calculated?
    - This pointer moves in jumps of the hop size
  - Read pointer: where does `render()` get the next samples for the system output?
    - This pointer moves continuously frame by frame
  - The write pointer should always remain ahead of the read pointer
    - Otherwise we've run out of data!



Output circular buffer

- Necessary characteristics of the output buffer:
  - Should hold at least W + H samples
  - Where W is the window size, H is the hop size
  - One buffer per channel of signal (like before)
  - Needs an array, plus a write pointer (from FFT) and read pointer (to system output)
- Using the output buffer:
  - When you finish the inverse FFT, add the result into the output buffer, starting at the write pointer
    - To reiterate: don't replace the contents, add the new result onto the existing content of the buffer
  - In the main `for()` loop in `render()`, copy samples from the output buffer into the audio output at the read pointer
  - Then (important!) set the sample you just copied to 0
  - Finally, increment the read pointer

The full signal chain

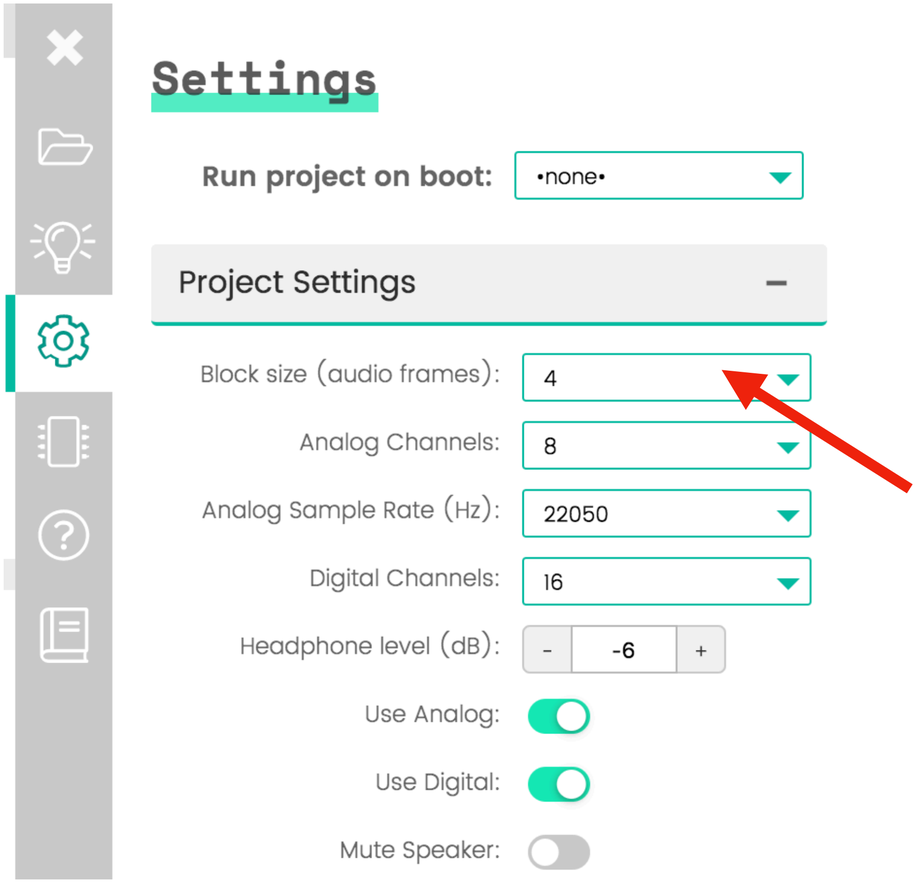


Overlap-add task

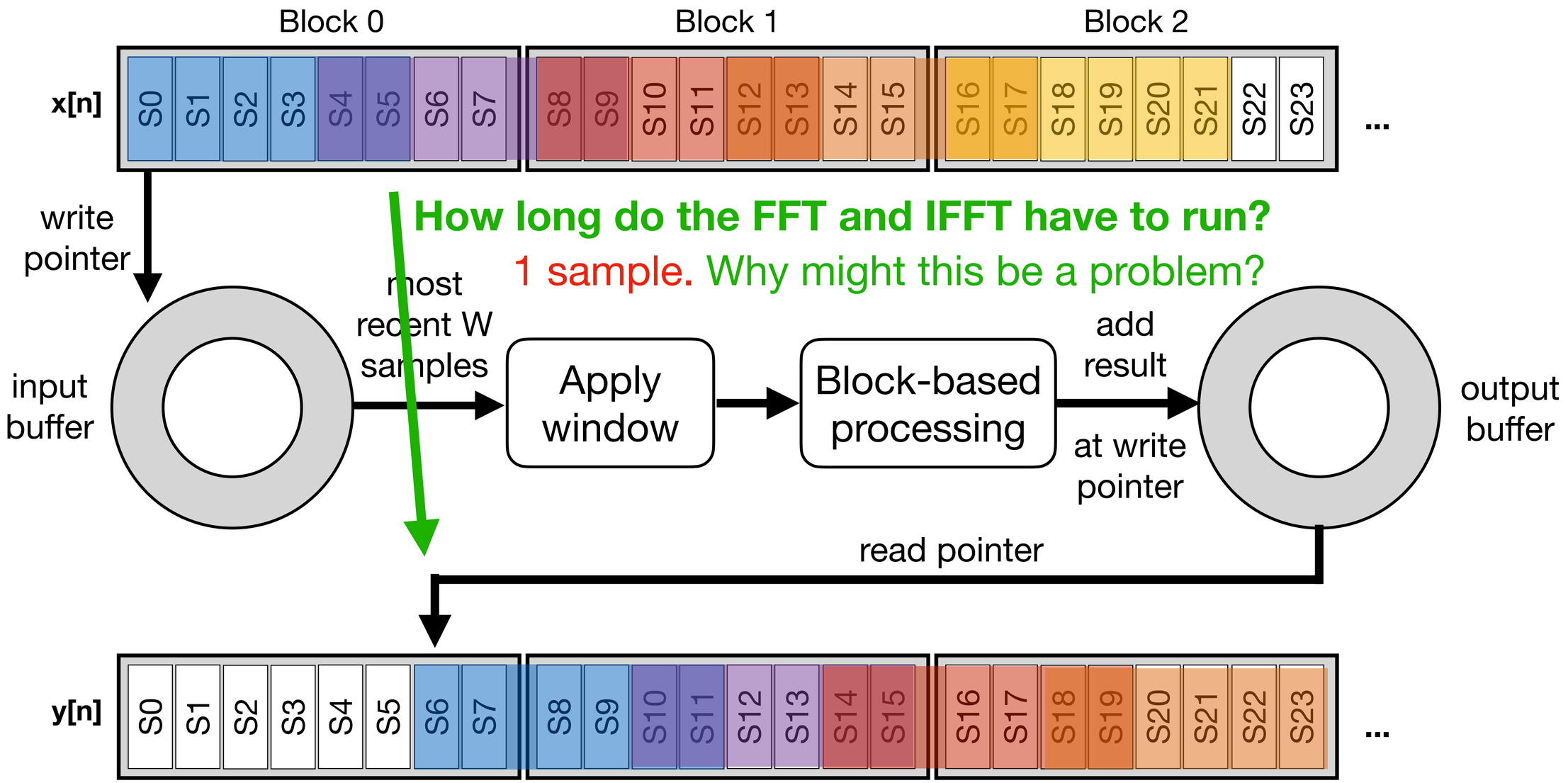
- Task: using fft-overlap-add project, implement the output circular buffer
  - Variables are already declared for you at the top of the file
  - Write pointer is for keeping track of where the next block is written
  - Read pointer keeps track of where to copy samples to the Bela audio output
  - Notice that `process_fft()` takes four arguments:
    - input buffer, input pointer, output buffer, output write pointer
  - Look at bottom of `process_fft()` to see how output is copied to the buffer
- Add your code in `render()`:
  - Copy samples from the output buffer to the audio output (use the variable `out`)
  - After you copy the output, set that sample in the output buffer to 0
  - Increment the output circular buffer pointers in the relevant places
    - Which one happens every frame? Which one advances once per hop?
- Once it works, try uncommenting the robotise code in `process_fft()`

Block-based processing performance

- In the Settings tab of the Bela IDE, try changing the block size to 4 samples
- What happens to the output? Why?

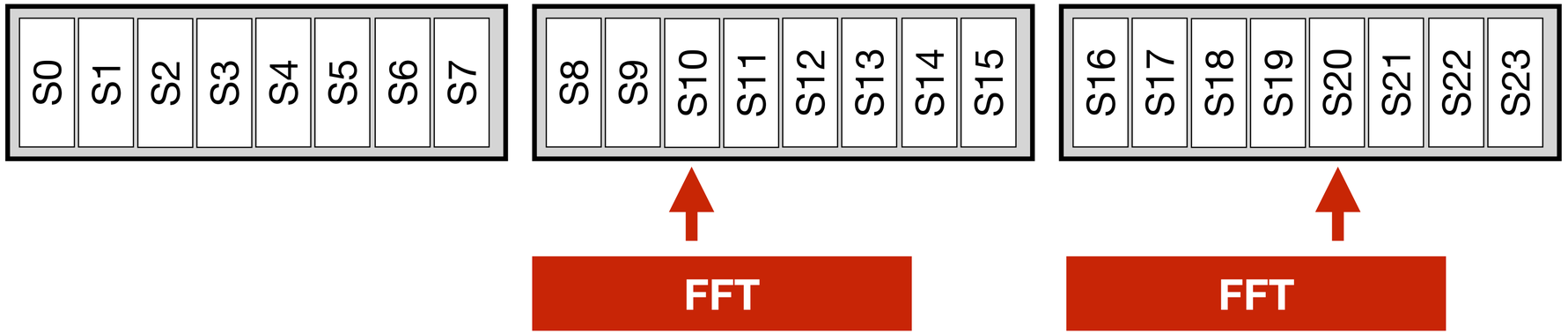


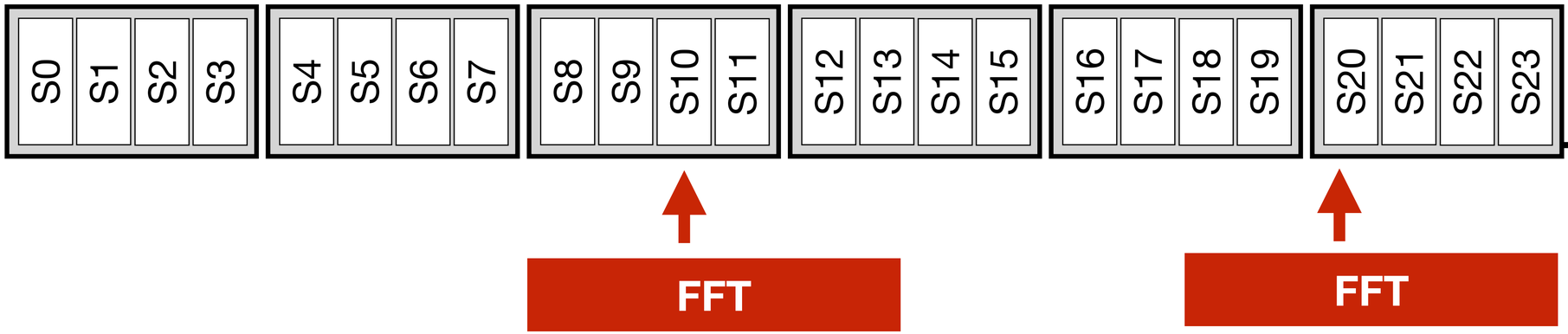
The full signal chain: performance



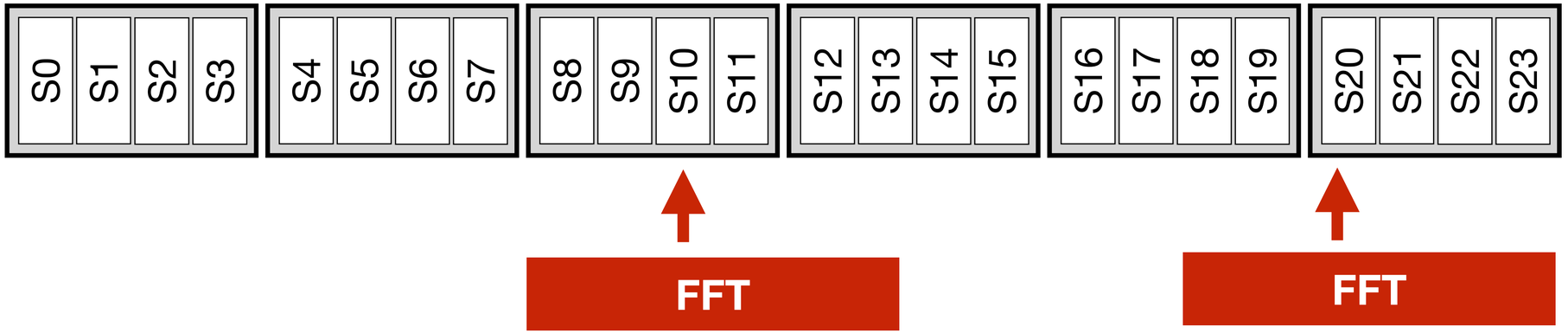
Intermittent CPU load

- We've been assuming the CPU load is roughly constant over time
  - Even if the average load is within limits, the instantaneous load might not be!
- Example: say we do an FFT every 10 samples
  - Let's assume only the FFT takes any time
  - With a buffer size of 8, we're fine:
  - Calculation takes less than one buffer length
  - But with a buffer size of 4, we're not...
  - Calculation takes longer than a call to `render()`



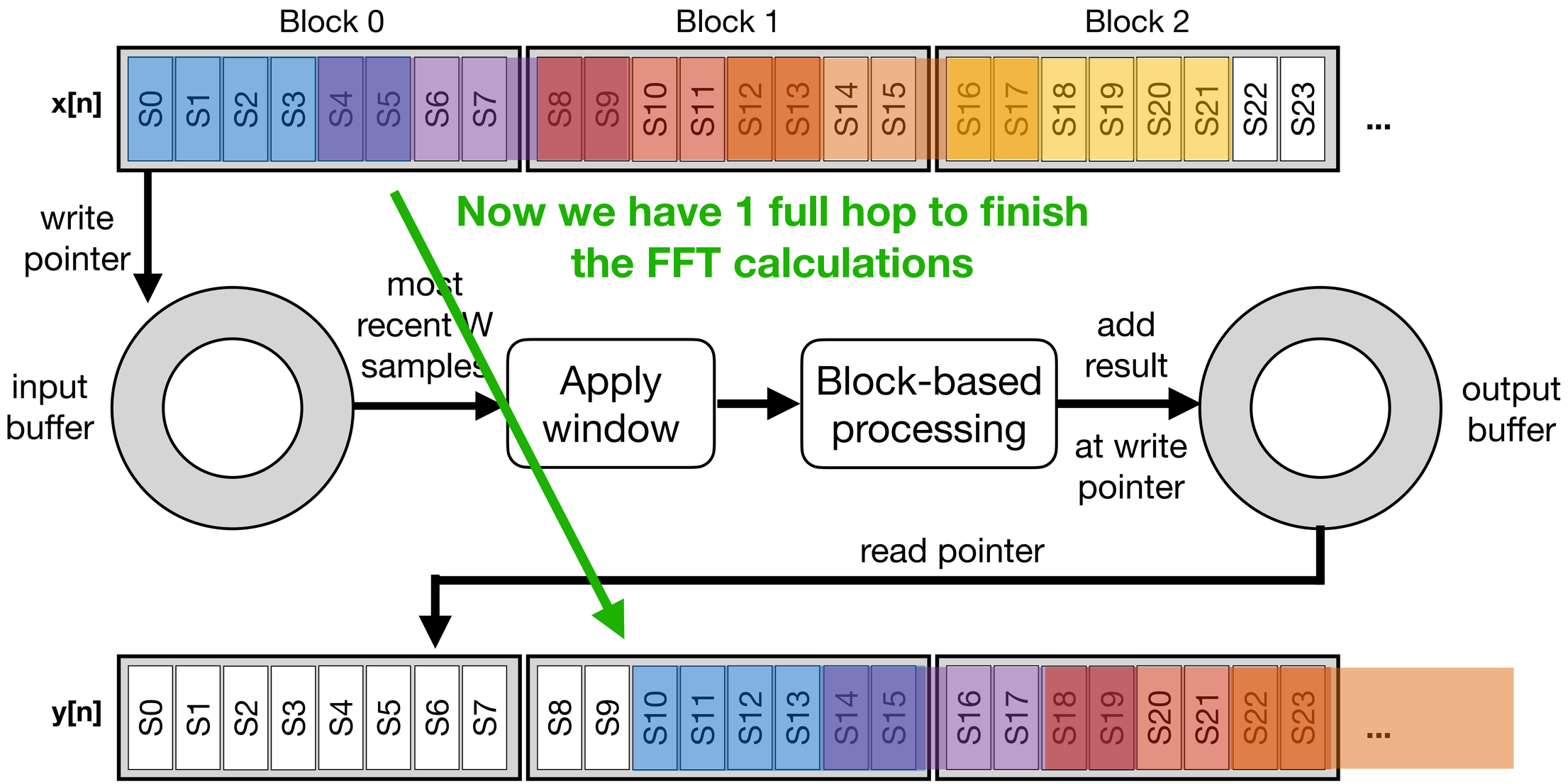


Handling intermittent load



- What can we do about occasional, heavy load?
  - Most calls to `render()` do almost nothing; some calls can't finish on time...
- We should add latency so the result doesn't need to be ready in 1 sample
  - But that still doesn't solve the problem if a calculation spans multiple calls to `render()`
- Solution: move the expensive process to another thread
  - Run the FFT on a thread with lower priority than `render()`
  - The FFT thread will use whatever processing is left over when `render()` is not running

The full signal chain: added latency

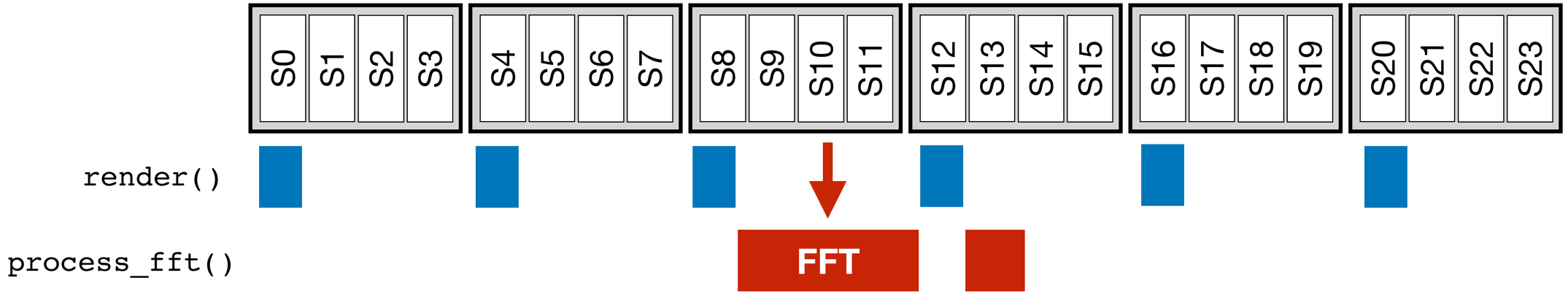


Multi-threaded processing

- Bela / Xenomai maintain a strict priority system for real-time threads
  - The thread with the highest priority always runs first, until it's finished
  - Then execution passes to the thread with the next highest priority
- Regular Linux threads (secondary mode) has a lower priority
  - These threads run when there are no Xenomai real-time threads available to run
  - The Linux scheduler decides which Linux threads will run at what time
    - This is a more complex question that we will not address in this class
- Upshot for block-based processing: use two threads
  - `render()`: high priority, runs every callback, finishes quickly
  - `process_fft()`: lower priority, runs occasionally, takes longer to finish
    - This will run whenever `render()` is not running
    - Still higher priority than the rest of Linux

Multi-threaded processing

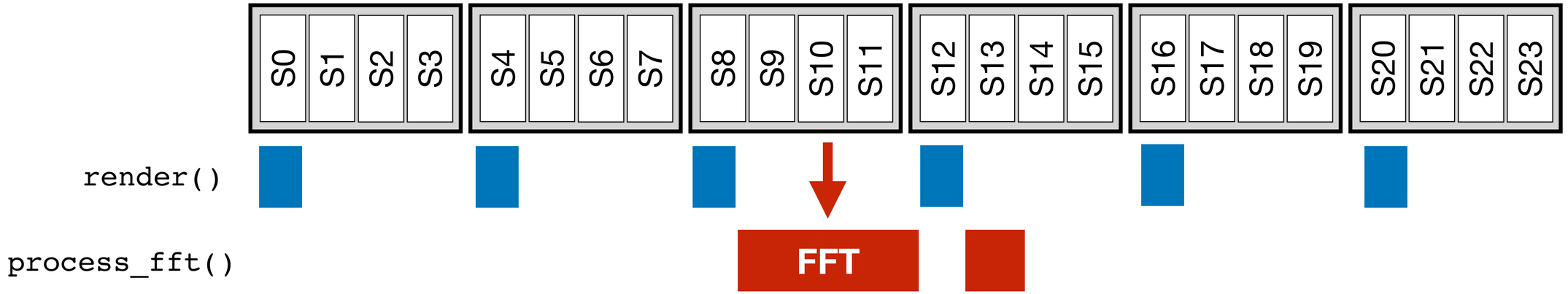
- With 2 threads, `render()` always runs first
  - The rest of the time is spent on the FFT, which could span multiple callbacks



- FFT thread should sleep and wait for a signal to run
  - `render()` should send the signal when a new hop has elapsed
  - `render()` also needs to tell the FFT thread where to find the data (locations of the buffers and pointers it needs)
  - FFT thread should run calculation, store the output in the buffer, then return

Multi-threaded processing

- With 2 threads, `render()` always runs first
  - The rest of the time is spent on the FFT, which could span multiple callbacks

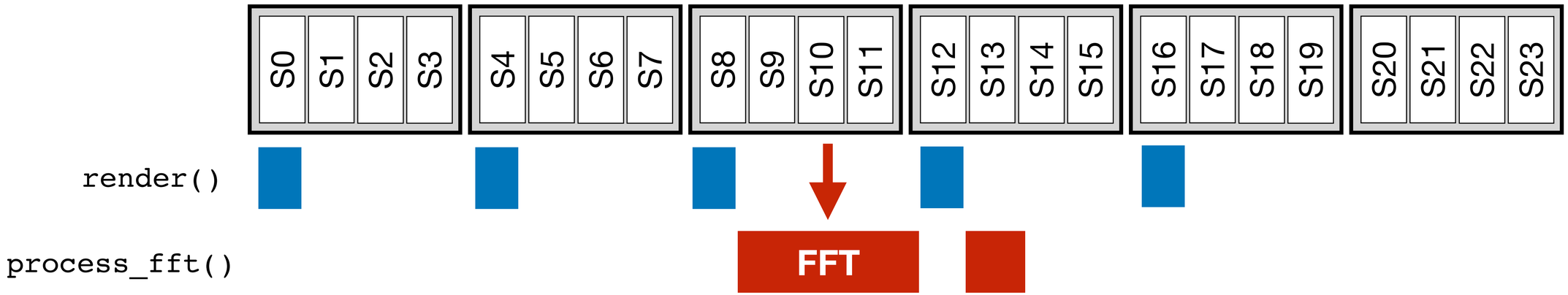


- `process_fft()` is allowed to be interrupted by `render()`
  - That means we need to be careful not to let `render()` modify any variables `process_fft()` was using
  - Main concern here is the input buffer pointer, which will keep changing before `process_fft()` is even allowed to run. Need to cache this value (save a copy that `render()` won't change).
  - Reverse isn't true: `render()` won't be interrupted because it has higher priority

Bela AuxiliaryTask API

- Bela provides a way of making Xenomai real-time threads (tasks)
- Each task requires four pieces of code:
  1. An `AuxiliaryTask` object
  2. A function to call which should be of type: `void function(void *argument);`
     - In practice you don't usually pass anything in the argument
  3. A call to `Bela_createAuxiliaryTask()` inside of `setup()`
     - This will pass in the object, specify which function should be called, and its real-time priority
     - `render()` has a priority of 95; valid levels are 0 to 99
  4. A call to `Bela_scheduleAuxiliaryTask()` whenever you want the task to run
     - Usually, but not always, inside of `render()`
     - Remember that the function won't actually be called until higher priority tasks finish
- You will generally use global variables to communicate between the tasks
  - For example, can't directly pass the `BelaContext` structure to an AuxiliaryTask

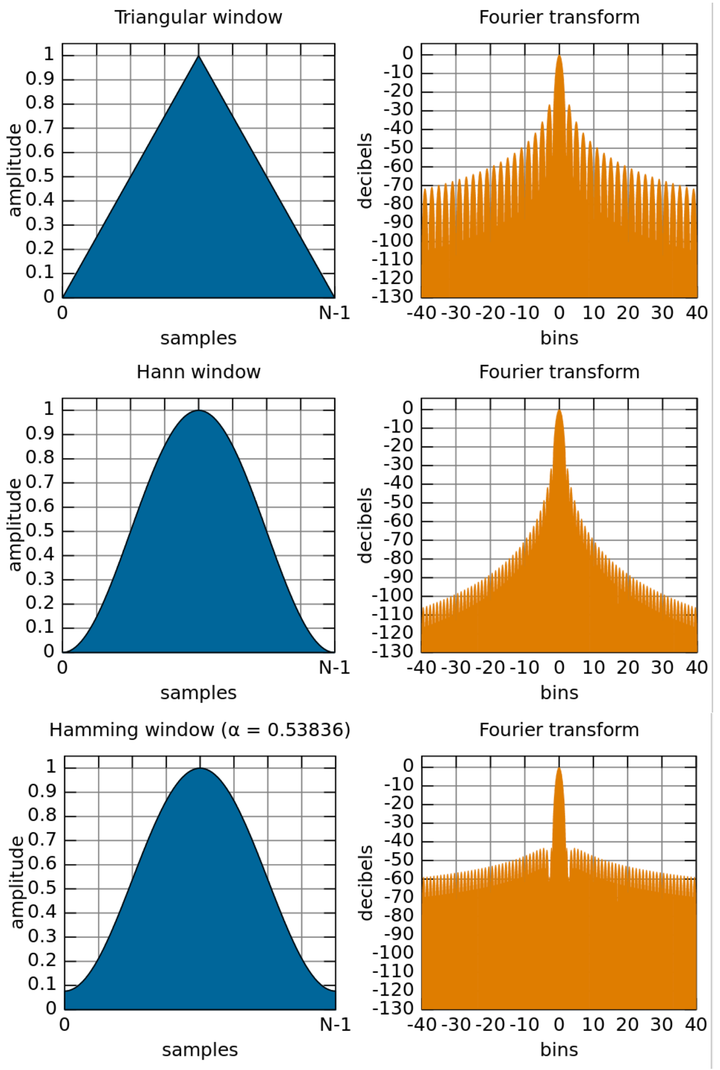
Threading task



- Task: using the fft-overlap-add-threads project
  - Convert the code to run the FFT in a second thread
  - Use the Bela `AuxiliaryTask` API: see `gFftTask` at top of the file
  - The initialisation is done for you in `setup()` - see `Bela_createAuxiliaryTask()`
  - You need to call `Bela_scheduleAuxiliaryTask()` each hop
  - Also finish implementing `process_fft_background()`

Next lecture: Phase vocoder, part 2

- Next time, we will look more at what we can do in the frequency domain
- Window functions: why and how
  - Constant Overlap-Add (COLA) criterion
  - Analysis and synthesis windows
- Reconstructing exact frequency from FFT bin phase
- Phase vocoder effects:
  - Robotisation (phase zeroing)
  - Whisperisation (phase randomisation)
  - Pitch shifting



[images: wikipedia]

Keep in touch!

Social media:

@BelaPlatform

forum.bela.io

blog.bela.io

More resources and contact info at:

learn.bela.io/resources