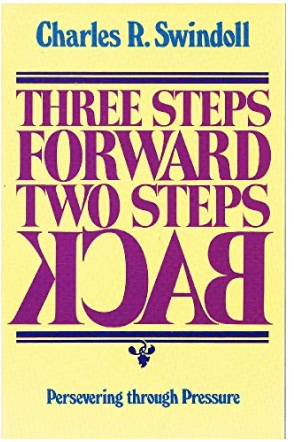
6.S965
Digital Systems Laboratory II

Lecture 5:

Some Intro Digital Signal Processing,

and maybe Drivers and Monitors

Vivado 2025.1



- 2025.1 has indeed fixed some bugs compared to 2024.1*

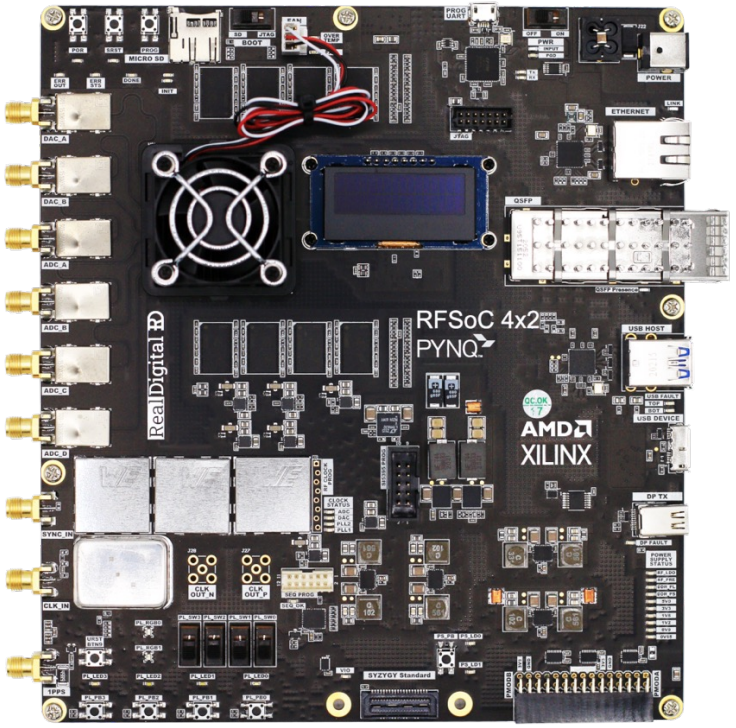
- Current new issue that we've seen to look for:
  - On updating IP, Vivado may possibly not realize the IP has updated (and can't be convinced otherwise)… The fix is: Close and reopen project
  - On updating IP, Vivado causes a core dump. The fix is to just let computer restart and move on.

*which weren't around in 2023.1 or .2

Signal Processing on the FPGA

6.S965 RFSoC

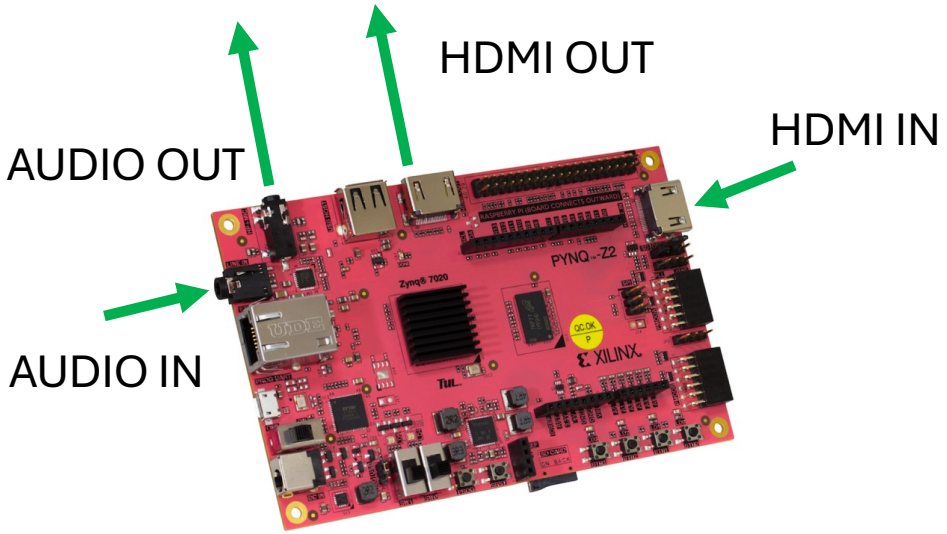
- UltraScale+ ZU48DR:
  - 38 Mb of BRAM
  - +22Mb of UltraRAM
  - 4272 DSP slices
  - 930,000 Logic Cells
  - **Four 5-Gsps 14 bit ADCs**
  - **Two 10-Gsps 14 bit DACs**
  - Four 1.3 GHz ARM 53 processors
  - Two Real-time 533 MHz ARM processors



- Board has 4GB of DDR4 for FPGA portion (”PL”) and 4 GB of DDR4 for processors (“PS”)

https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-rfsoc.html#tabs-b3ecea84f1-item-e96607e53b-tab

Pynq Z2 Board



- Series 7000 XC7Z020:
  - 5.04 Mb of BRAM
  - 220 DSP slices
  - 85K logic cells
  - Two 650 MHz A9 ARM processors
  - High-speed interconnects between two resources
- Board has 512 MB of DDR3

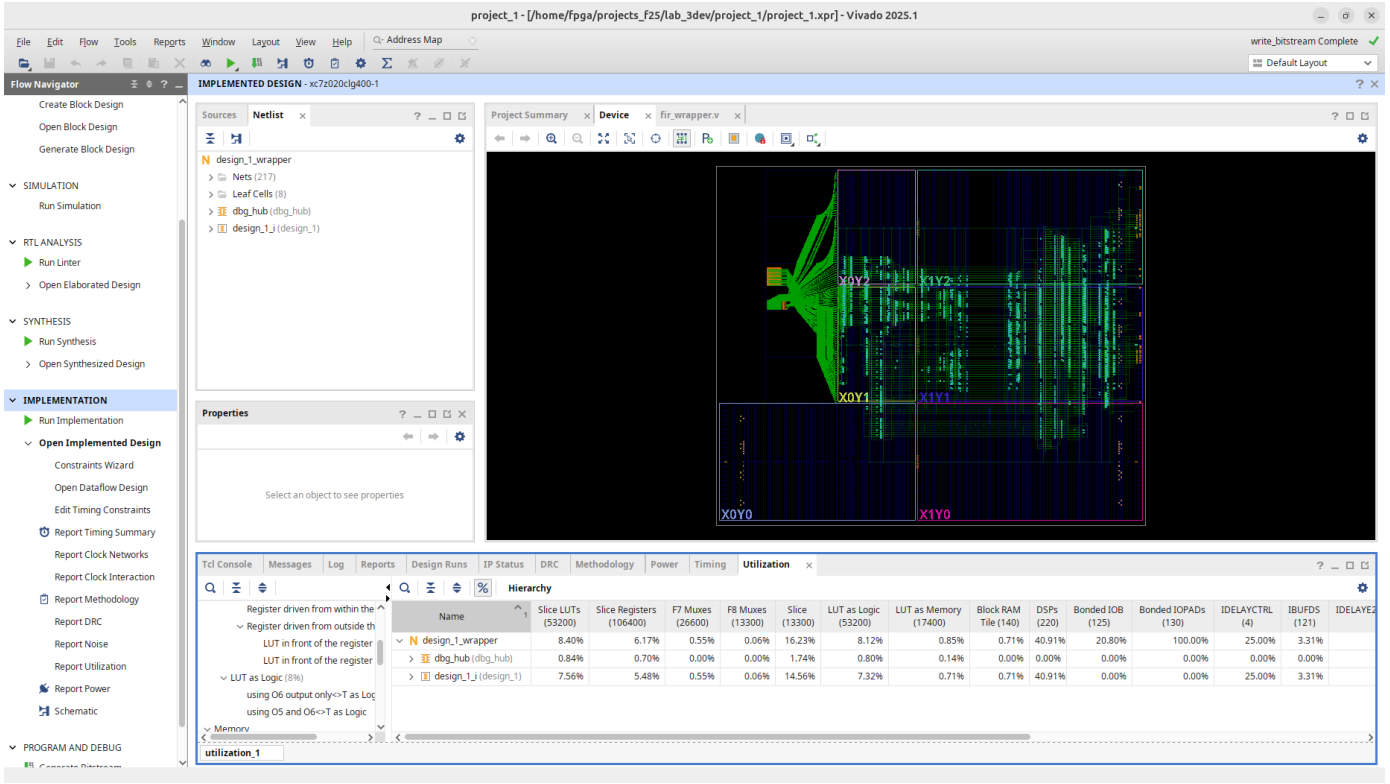
https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-7000.html

Digital Design and DSP

- A lot of signal manipulation and signal processing involves doing huge amounts of math.
- Much of that math is irreducible.
- But also, much of that math exists in algorithms that are “embarrassingly parallel”, meaning it is very easy to split up and do in parralel
- And hardware is very good at that.

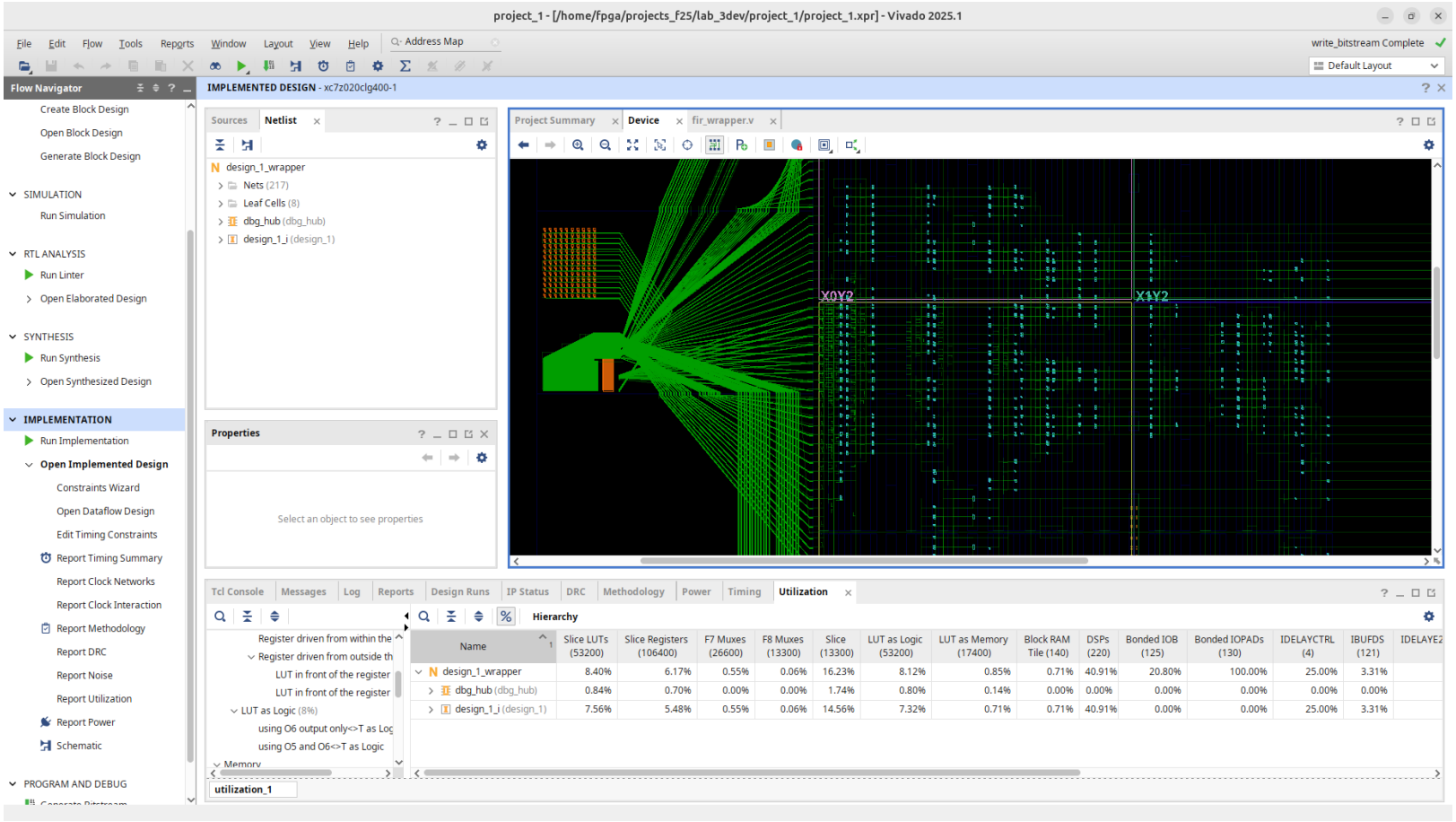
If you accidentally clicked “OK” rather than “cancel” after your bitstream build completes…

- It’ll open up your design…you can see the entire circuit

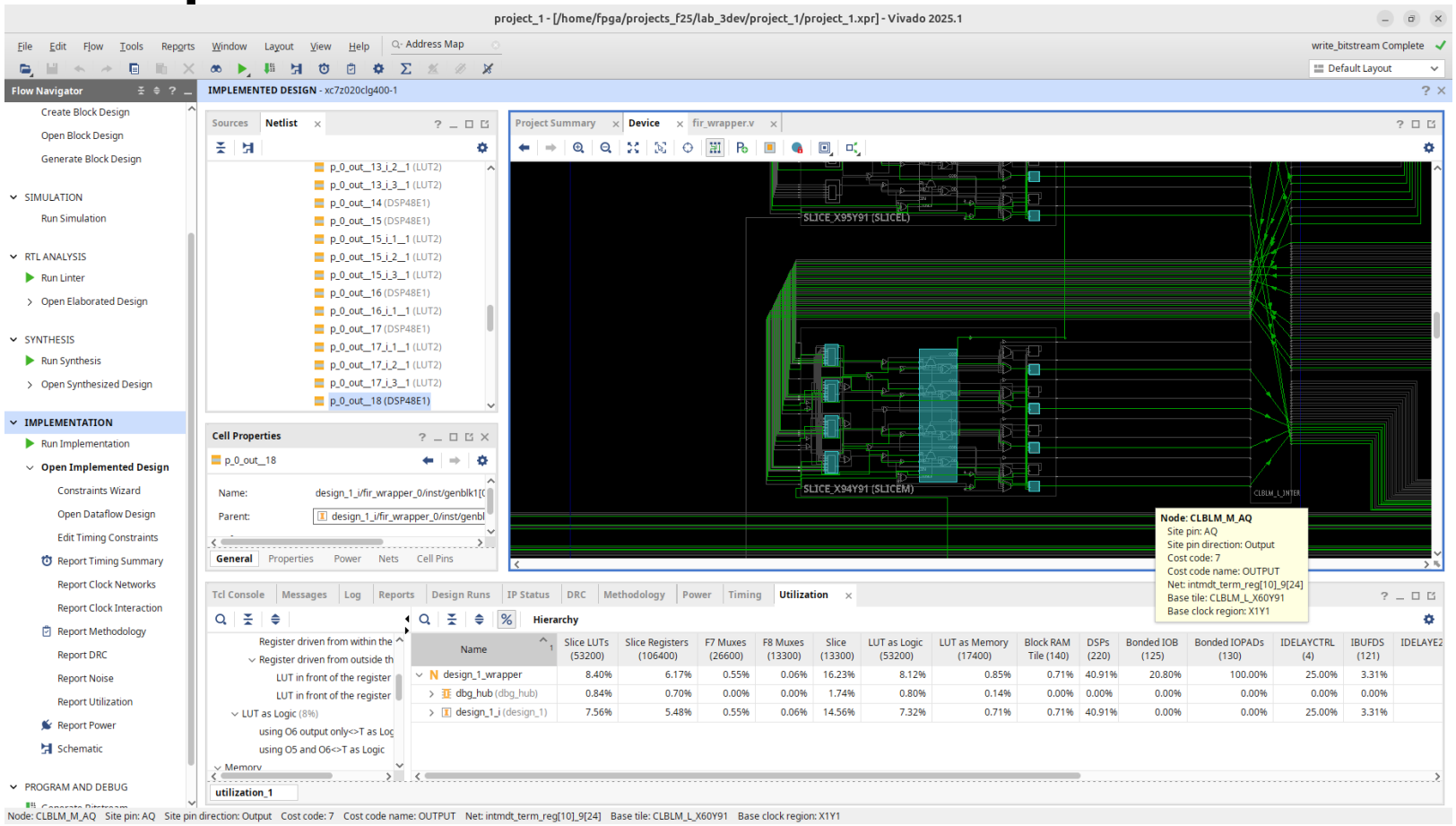


Keep Zooming…

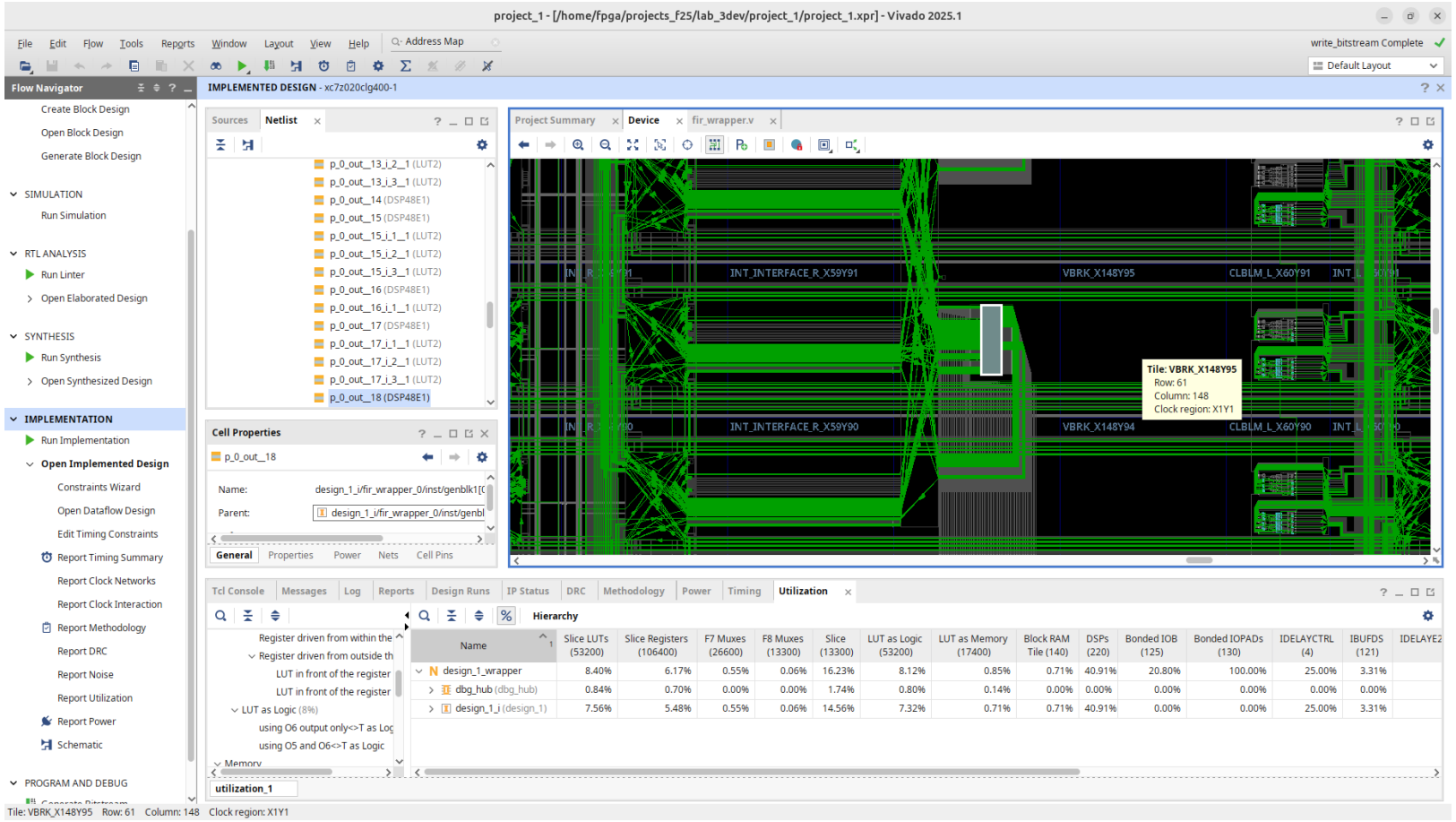
- Shows you every wire… And every component in your design



You can go all the way down to the flip-flops



Logic, BRAM, also DSP blocks



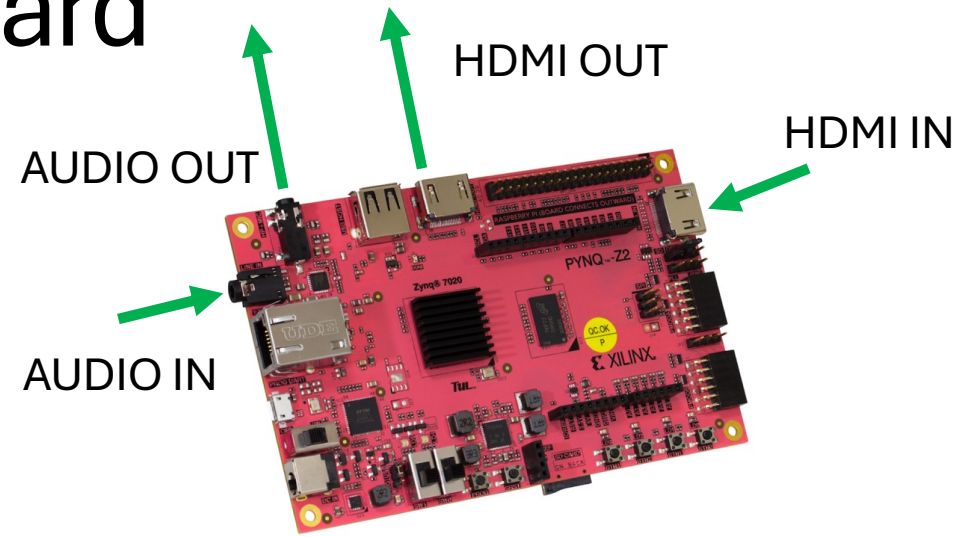
DSP Blocks???

- Logic is for Logicking
- Memory is for Remembering
- What are DSP blocks for?

DSP Blocks

- Multiply-then-add is a common operation chain in many things, particularly Digital Signal Processing
- FPGA has dedicated hardware modules called DSP48 blocks on it
  - Capable of single-cycle multiplies
- Can get inferred from using * in your Verilog that isn't a power of 2:
  - x*y, for example, will likely will result in DSP getting used
  - May take a full clock cycle so would need to budget timing accordingly

6.S965 Pynq Z2 Board

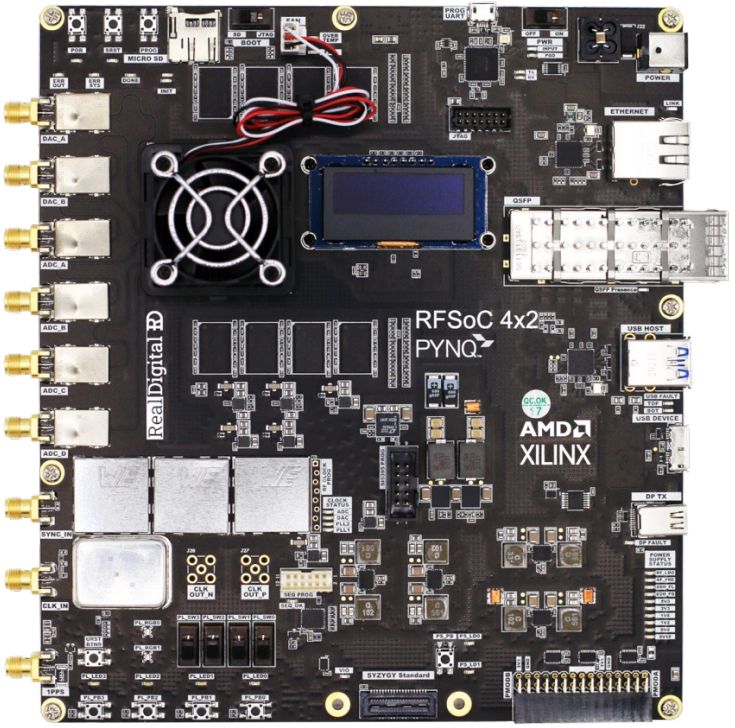


- Series 7000 XC7Z020:
  - 5.04 Mb of BRAM
  - ***220 DSP slices***
  - 85K logic cells
  - Two 650 MHz A9 ARM processors
  - High-speed interconnects between two resources
- Board has 512 MB of DDR3

https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-7000.html

6.S965 RFSoC

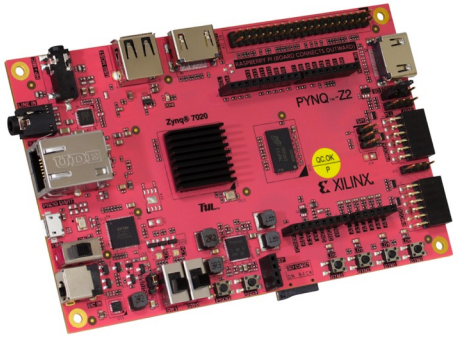
- UltraScale+ ZU48DR:
  - 38 Mb of BRAM
  - +22Mb of UltraRAM
  - ***4272 DSP slices***
  - 930,000 Logic Cells
  - Four 5-Gsps 14 bit ADCs
  - Two 10-Gsps 14 bit DACs
  - Four 1.3 GHz ARM 53 processors
  - Two Real-time 533 MHz ARM processors



- Board has 4GB of DDR4 for FPGA portion (”PL”) and 4 GB of DDR4 for processors (“PS”)

https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-rfsoc.html#tabs-b3ecea84f1-item-e96607e53b-tab

DSP48 Slice (High Level)



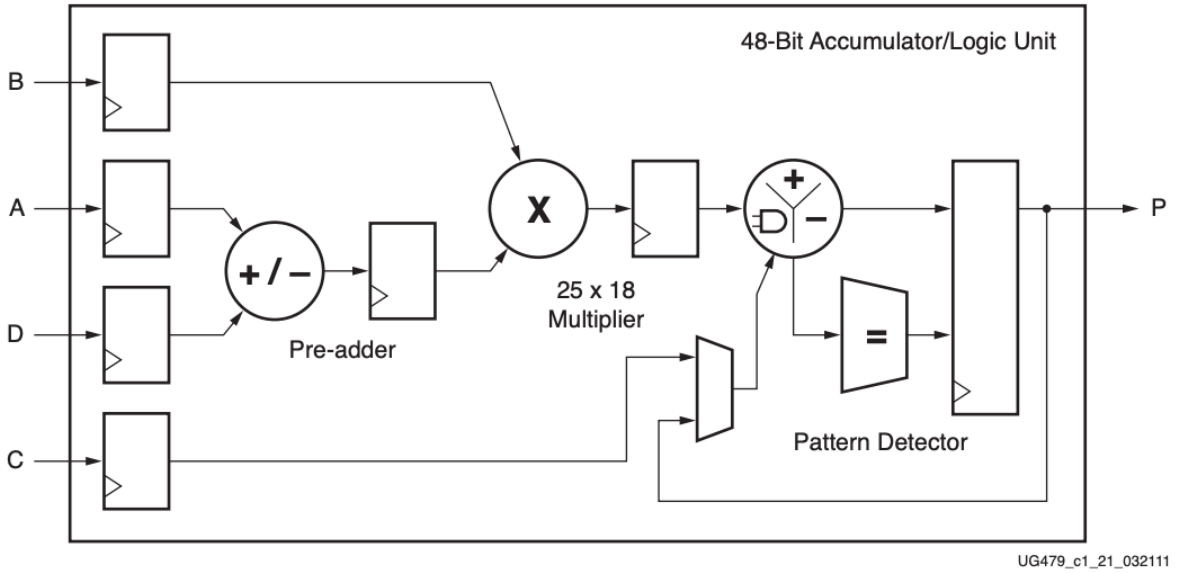
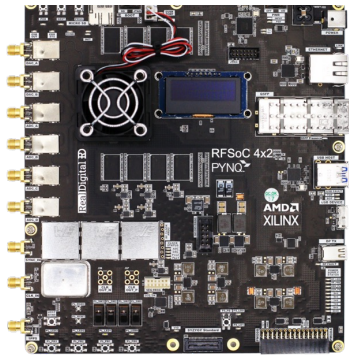


Figure 1-1: **Basic DSP48E1 Slice Functionality**

https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

DSP48E2 (Ultrascale +)



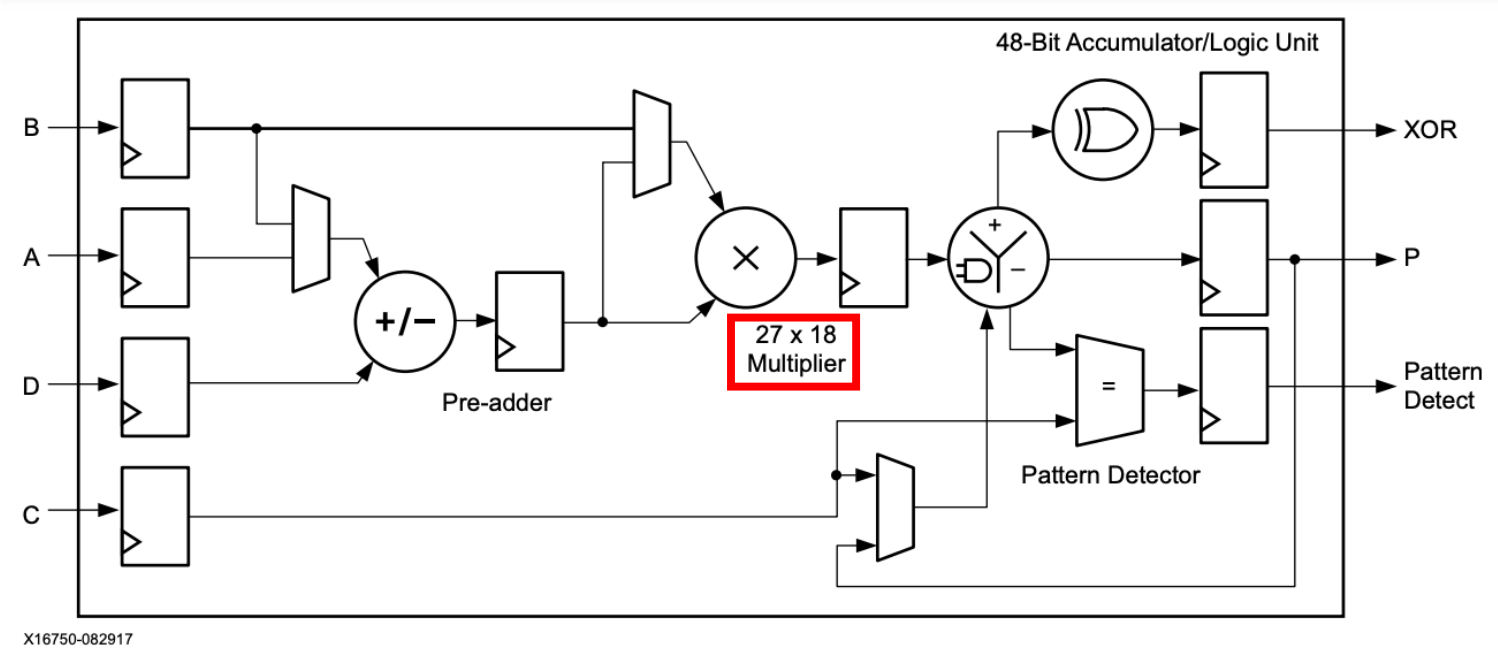
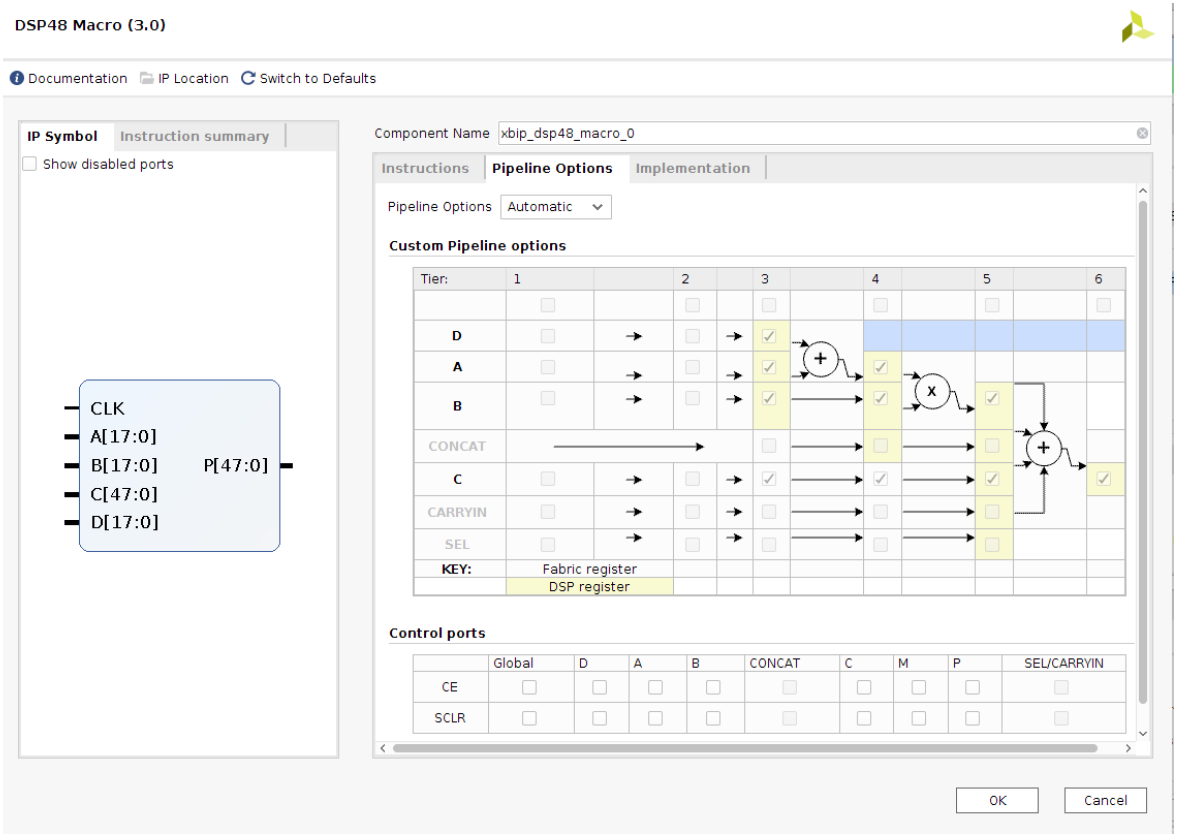


Figure 1-1: **Basic DSP48E2 Functionality**

DSP Blocks



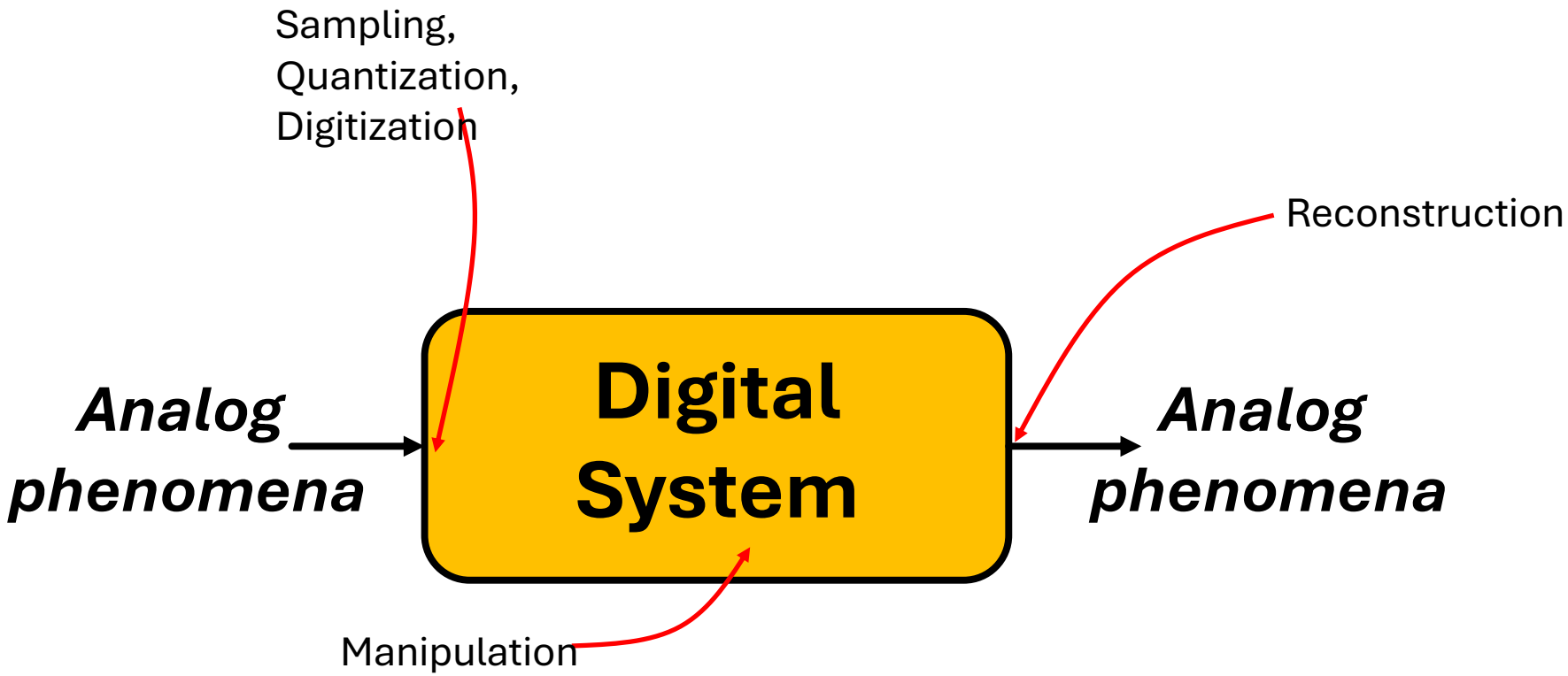
- Can manually instantiate them
- Or you can have their usage come from inference
- Or you can use IP which has already laid them out efficiently (for example an FFT block).

The need to Multiply-then-Add…

- Is pervasive in DSP applications, hence their name
- We’ll see why in a bit.

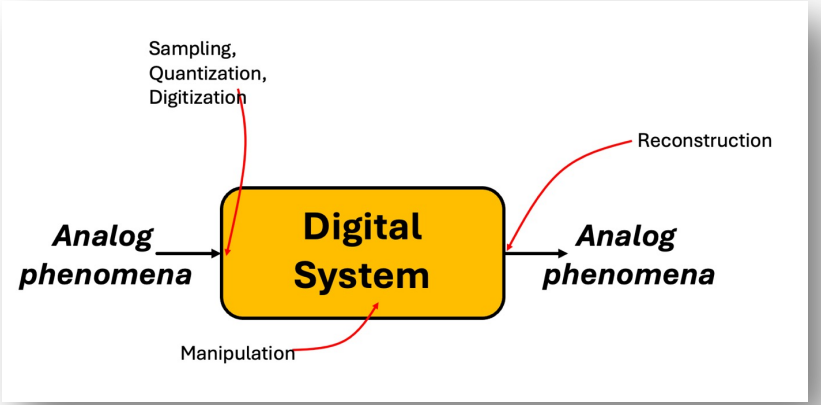
A Digital System in an Analog World

- Many physical phenomena (sound, light, physics in general) are best-described as continuous entities



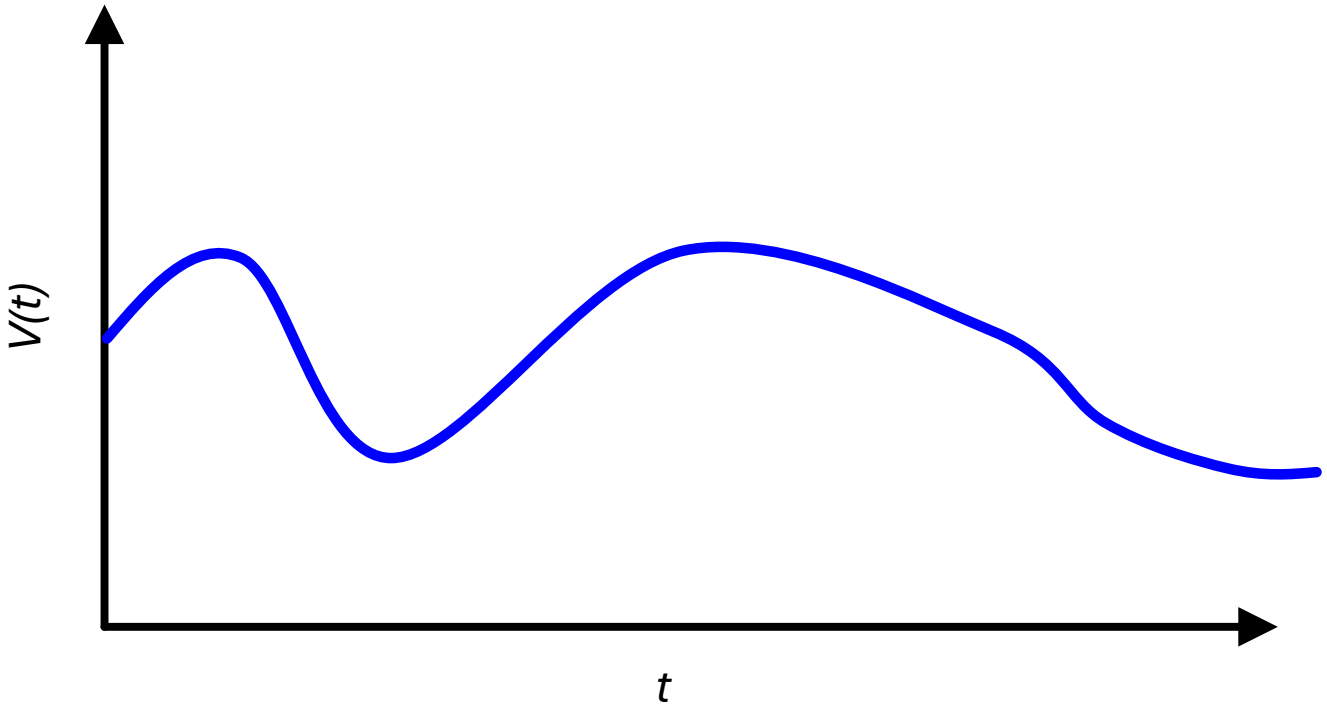
The “System” can be very large now.

- In the case of me watching Cat TV on youtube, the signals:
  - depart the analog realm in Cornwall, England where it is recorded.
  - Largely stays in digital format (with some transmission exceptions) until it exits my phone or computer display and hits my cat’s or my eyes in Boston.

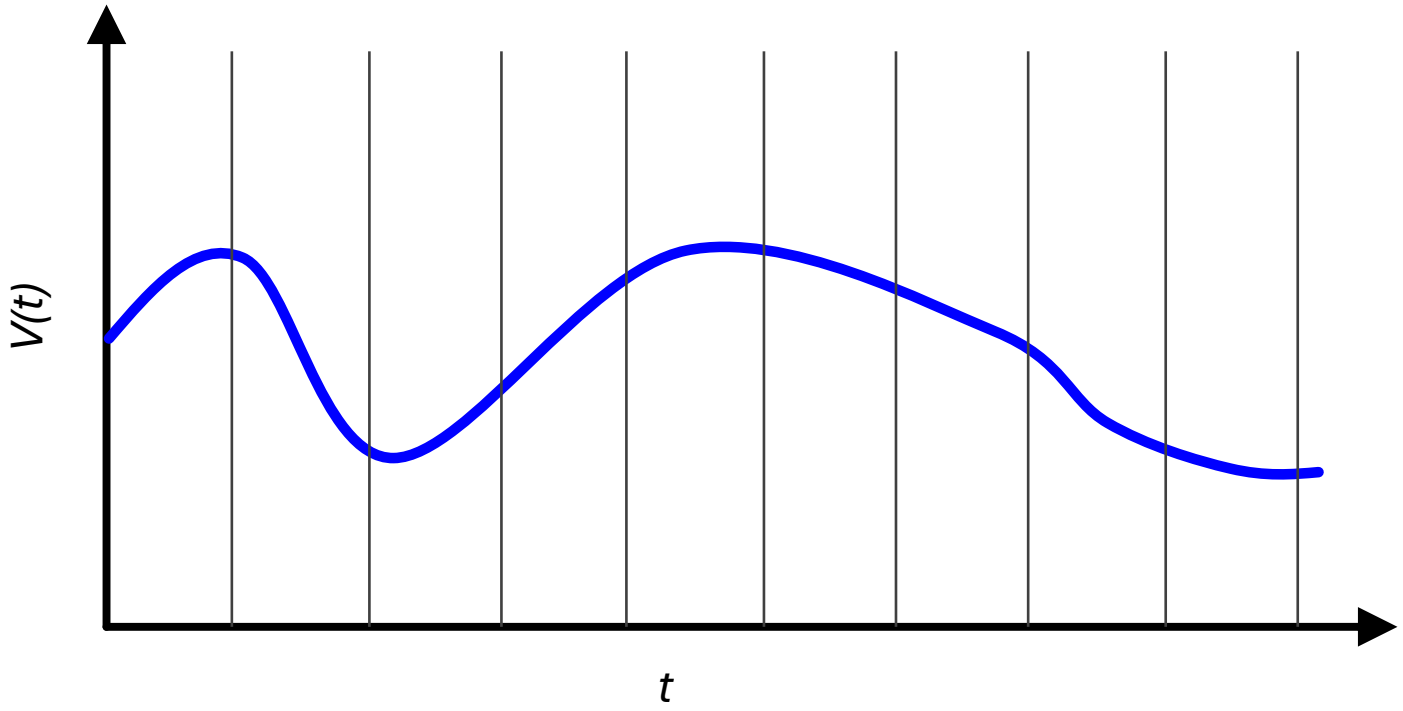


Visualizing Sampling

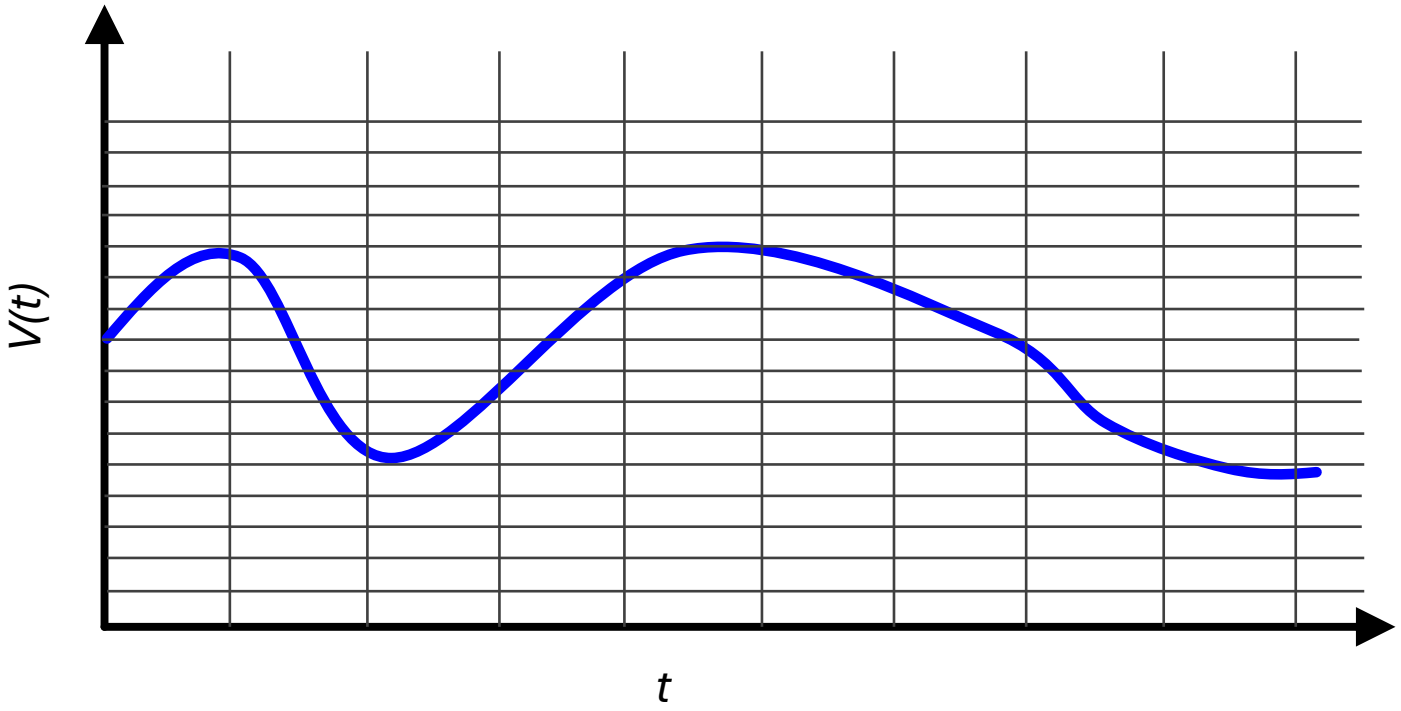
Continuous in Value and in Time



Discretization in Time

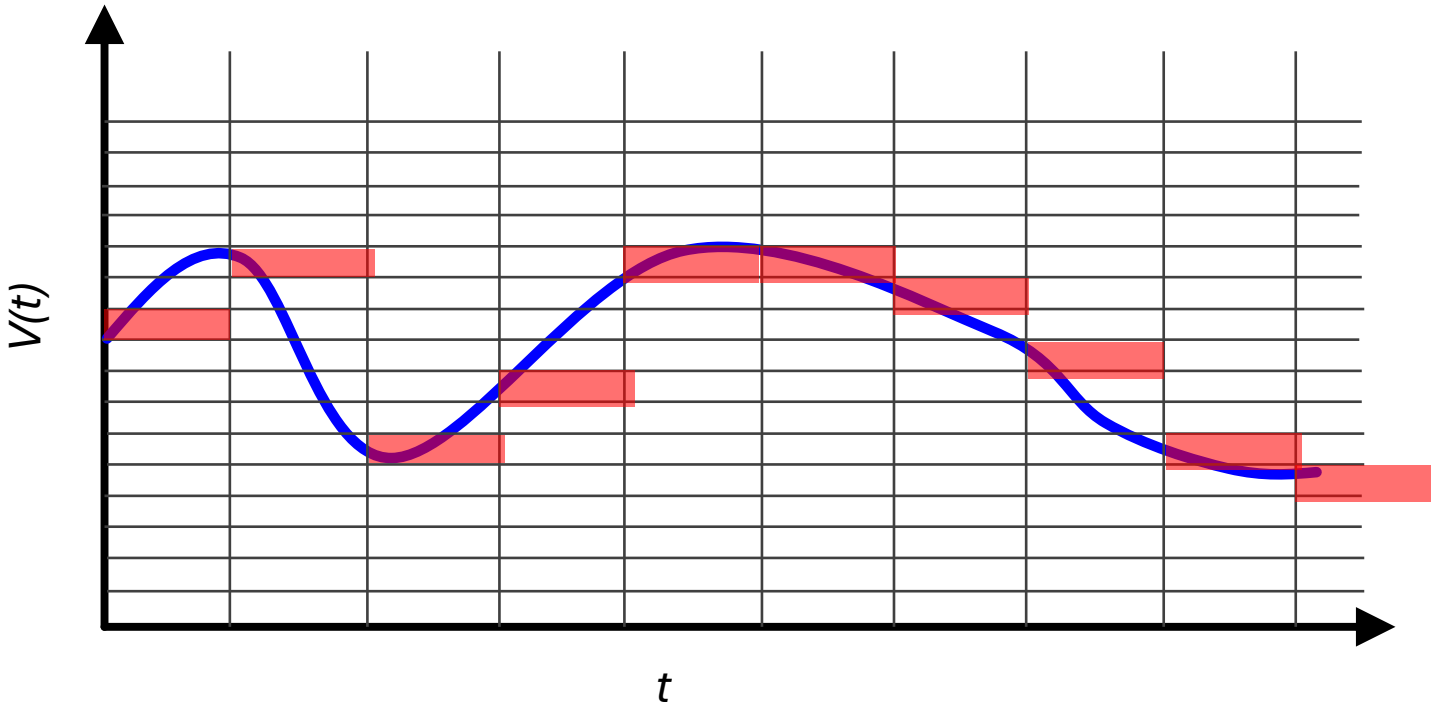


Discretization in Time and **Quantization** in Value



4 bit value encoding

Discretization in Time and **Quantization** in Value



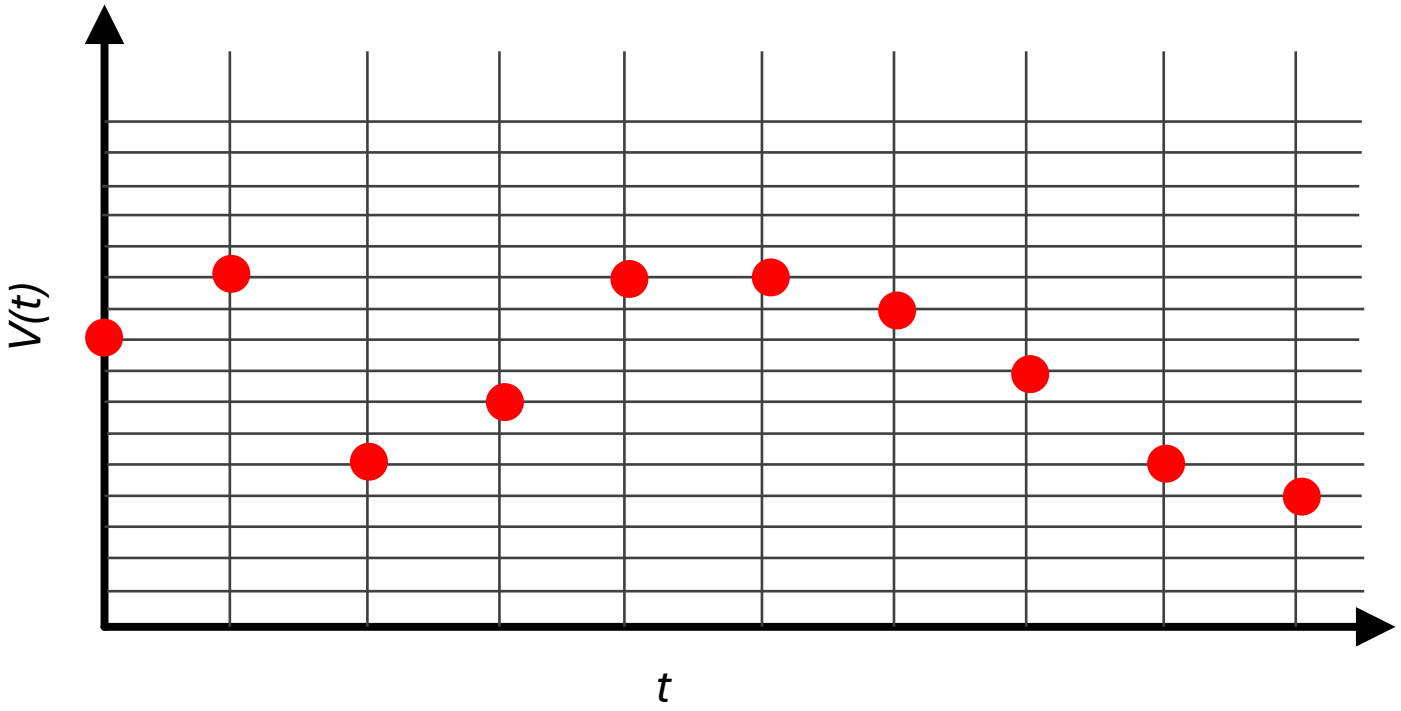
v[n] = [9,11,5,7,11,11,10,8,5,4,]

4 bit value encoding

Store in memory

- v[n] = [9,11,5,7,11,11,10,8,5,4,]
- 10 4-bit values: need 40 bits to represent!

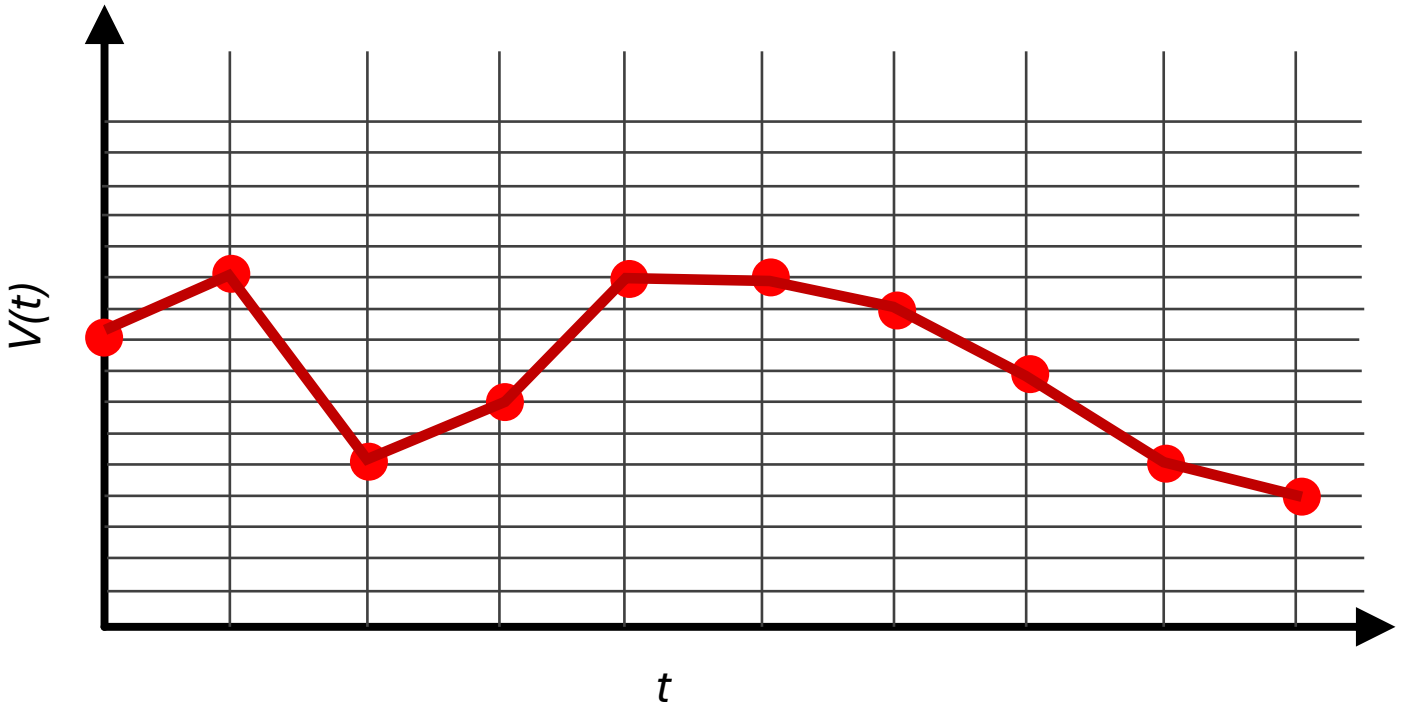
Reconstruction of Signal



v[n] = [9,11,5,7,11,11,10,8,5,4,]

4 bit value encoding

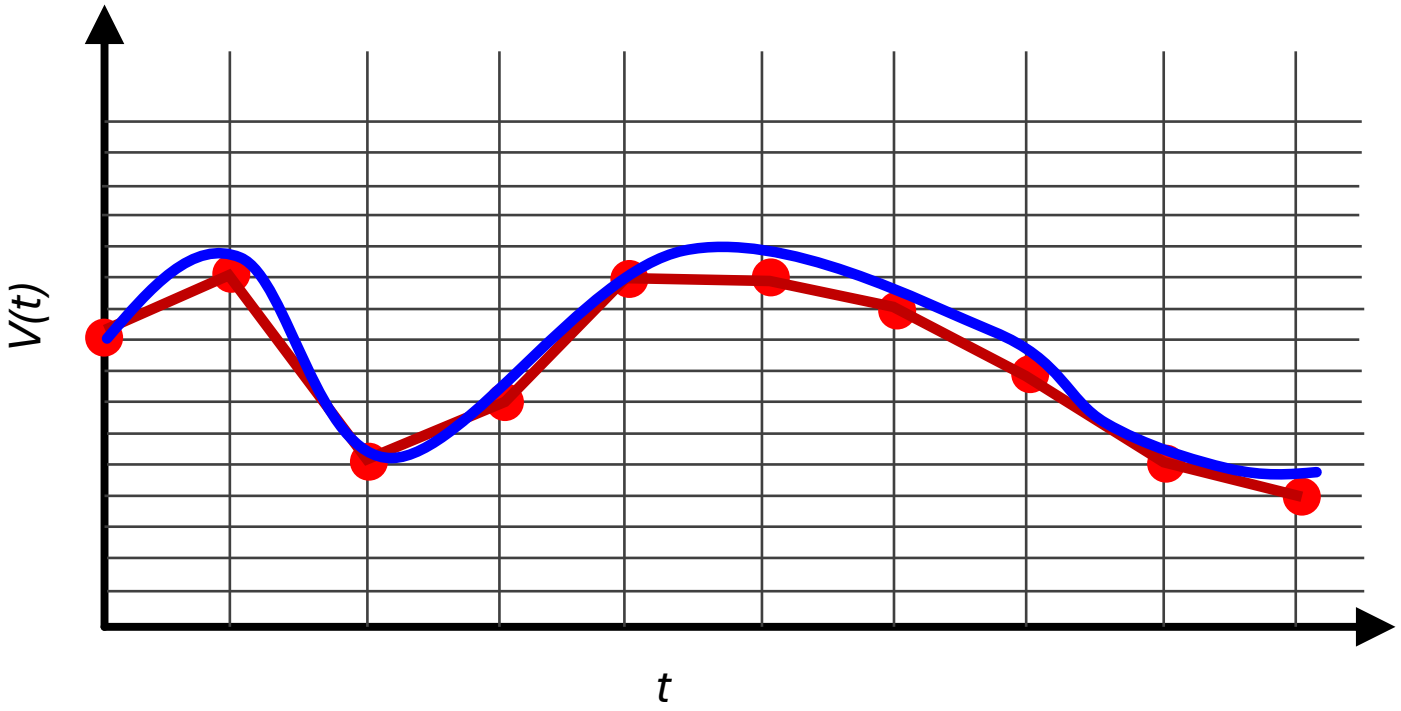
Reconstruction (with first-order hold interpolation)



v[n] = [9,11,5,7,11,11,10,8,5,4,]

4 bit value encoding

Compare to original… not bad



v[n] = [9,11,5,7,11,11,10,8,5,4,]

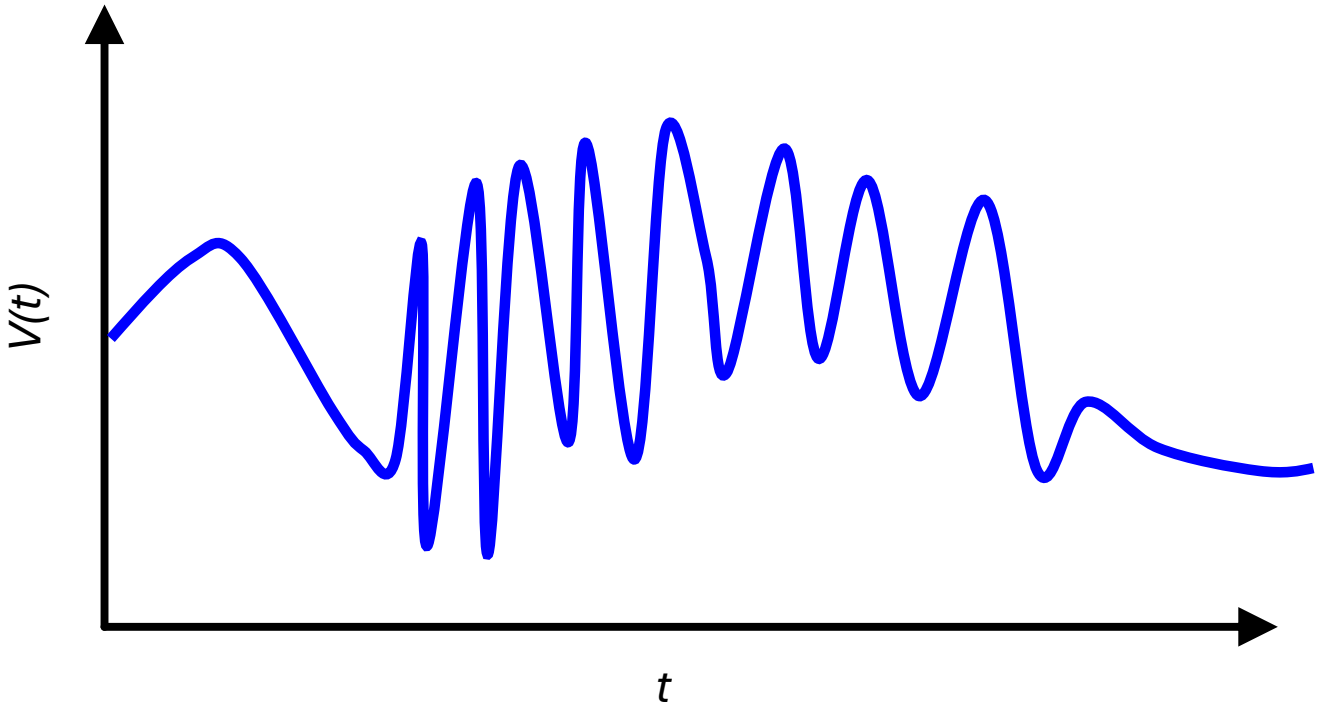
4 bit value encoding

Errors

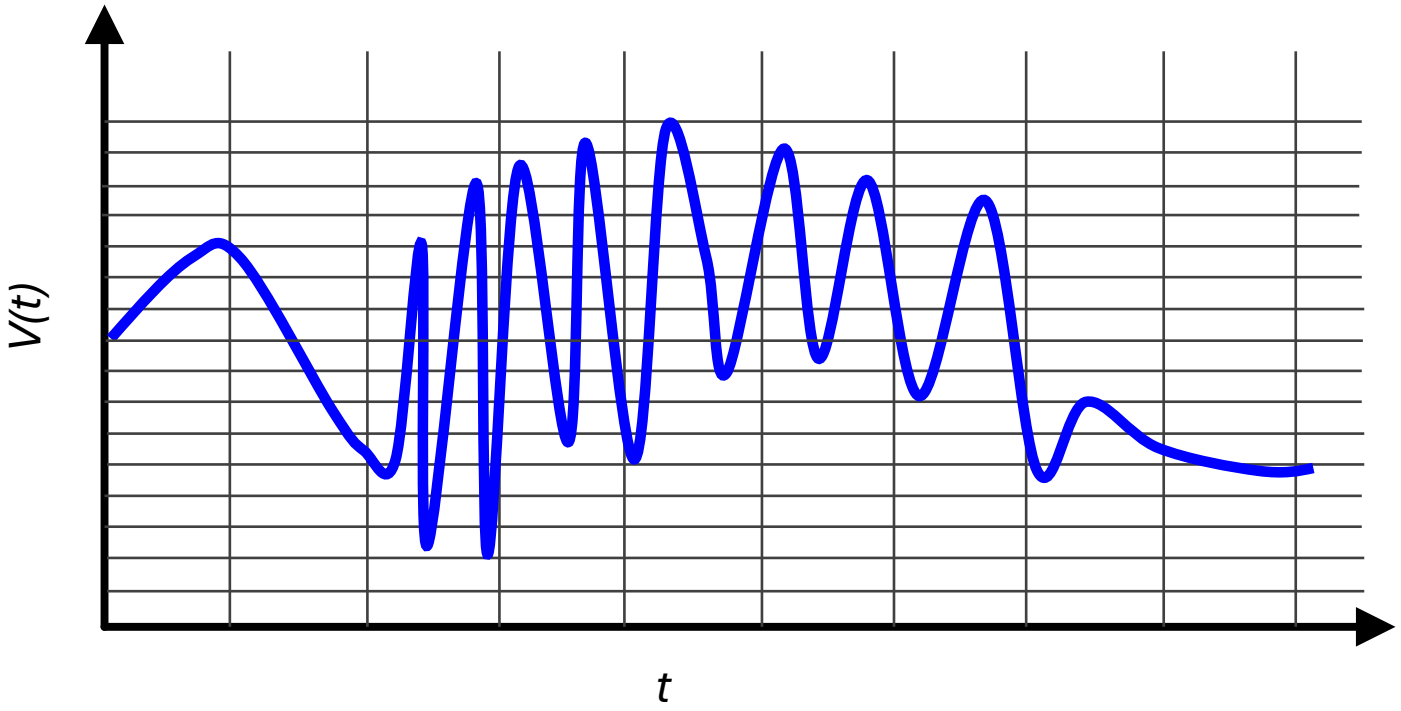
- **Discretization Error:** How ”off” our readings are in time due to sampling at discrete intervals

- **Quantization Error:** How “off” our readings are in reproduced value…if our bin size is 50mV and our signal varies only by 20mV this is going to cause problems

Continuous in Value and in Time

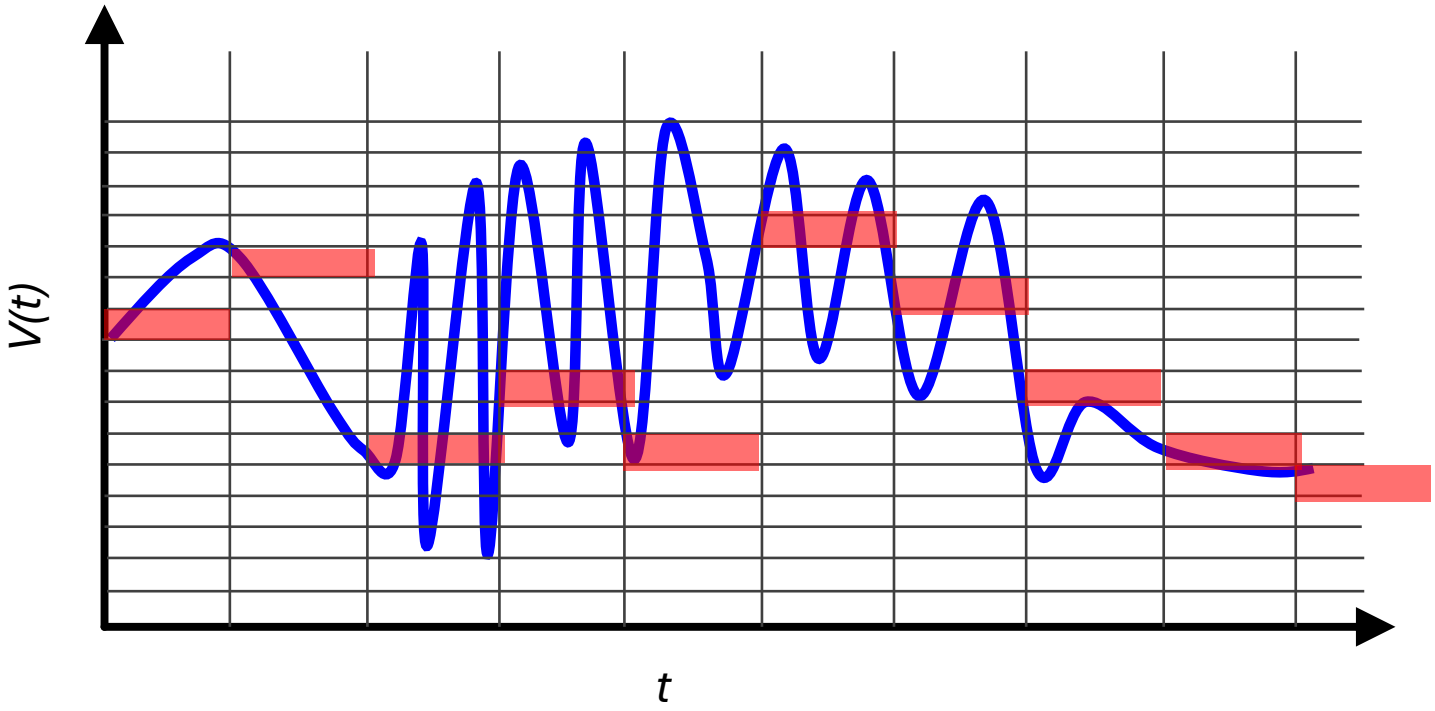


Discretization in Time and **Quantization** in Value



4 bit value encoding

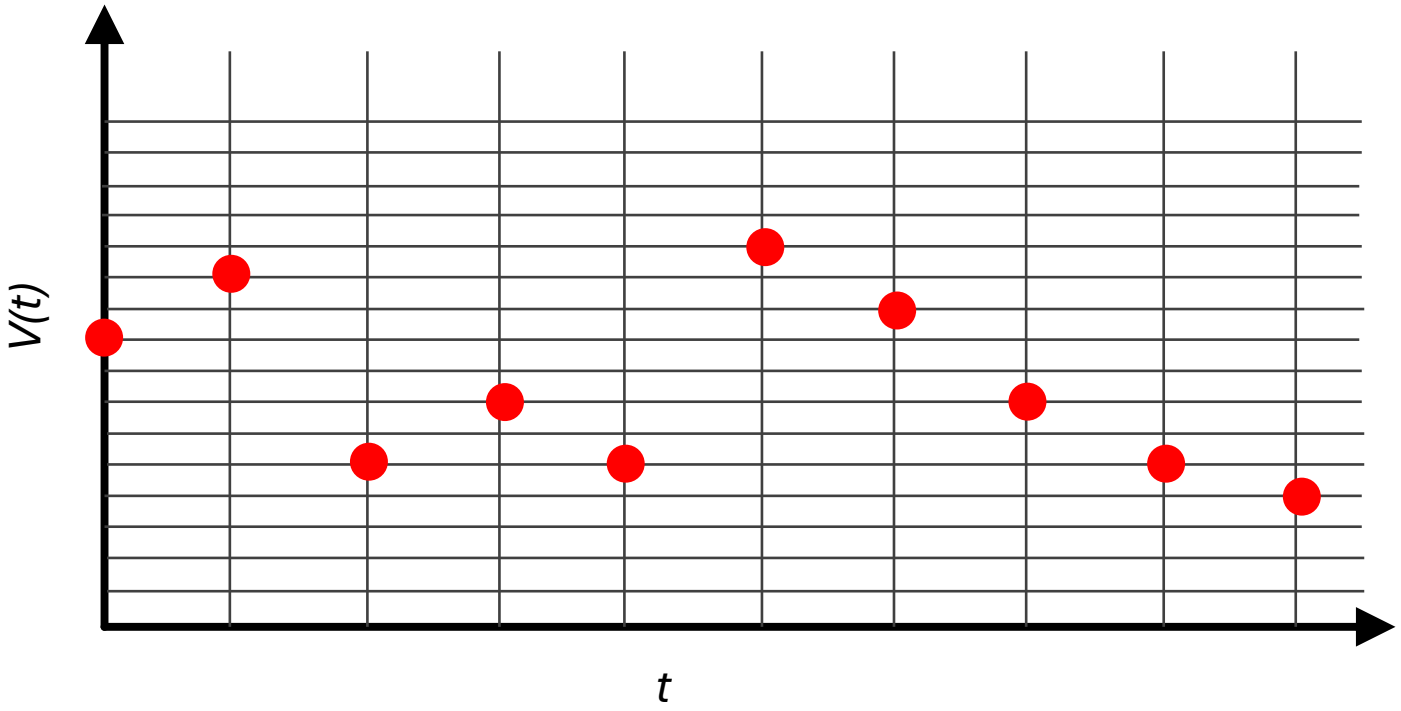
Discretization in Time and **Quantization** in Value



v[n] = [9,11,5,7,5,12,10,7,5,4,]

4 bit value encoding

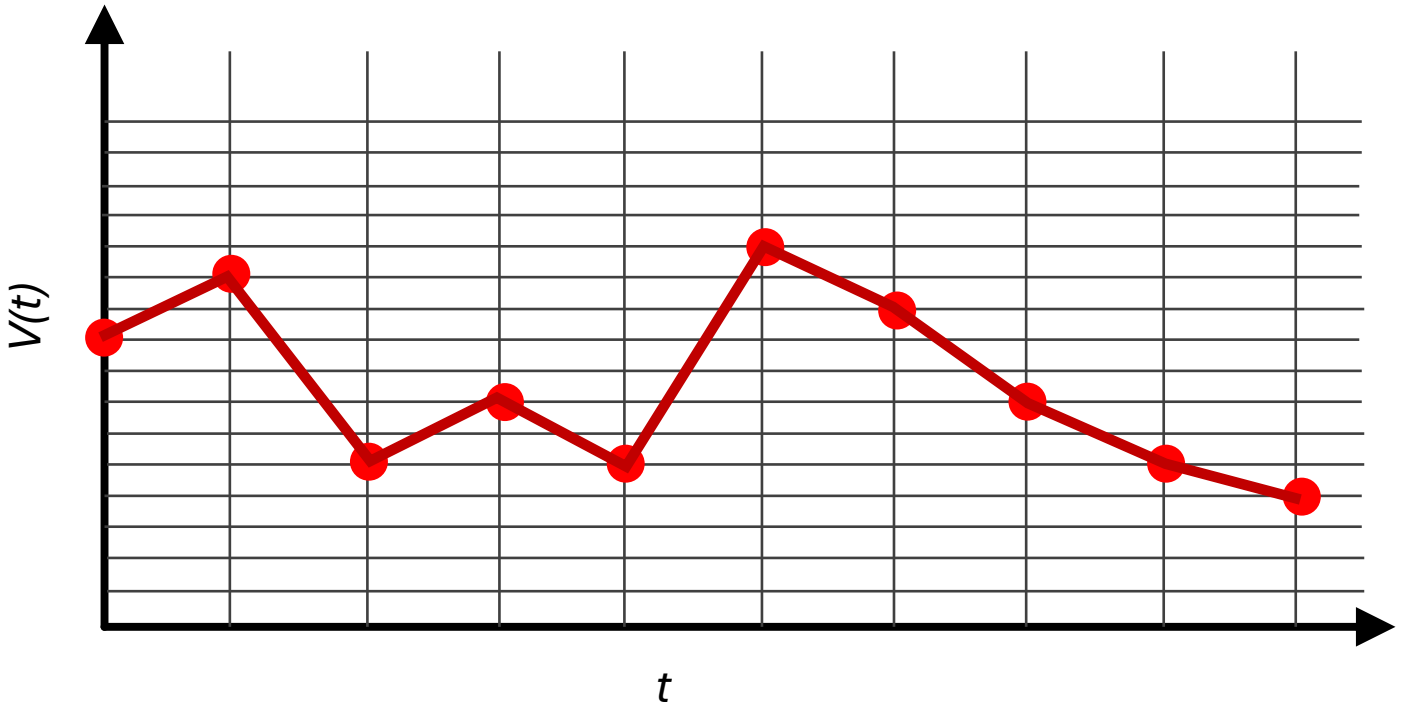
Reproduce



v[n] = [9,11,5,7,5,12,10,7,5,4,]

4 bit value encoding

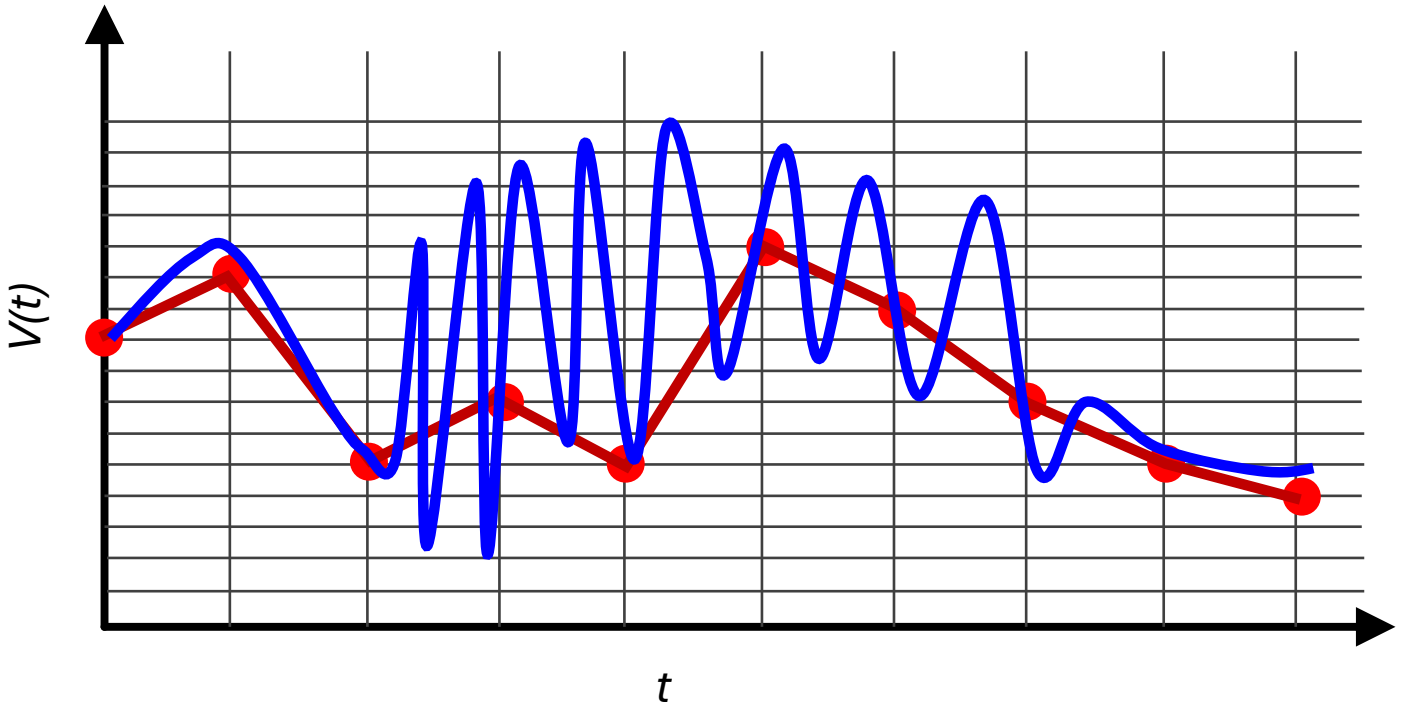
Reproduce



v[n] = [9,11,5,7,5,12,10,7,5,4,]

4 bit value encoding

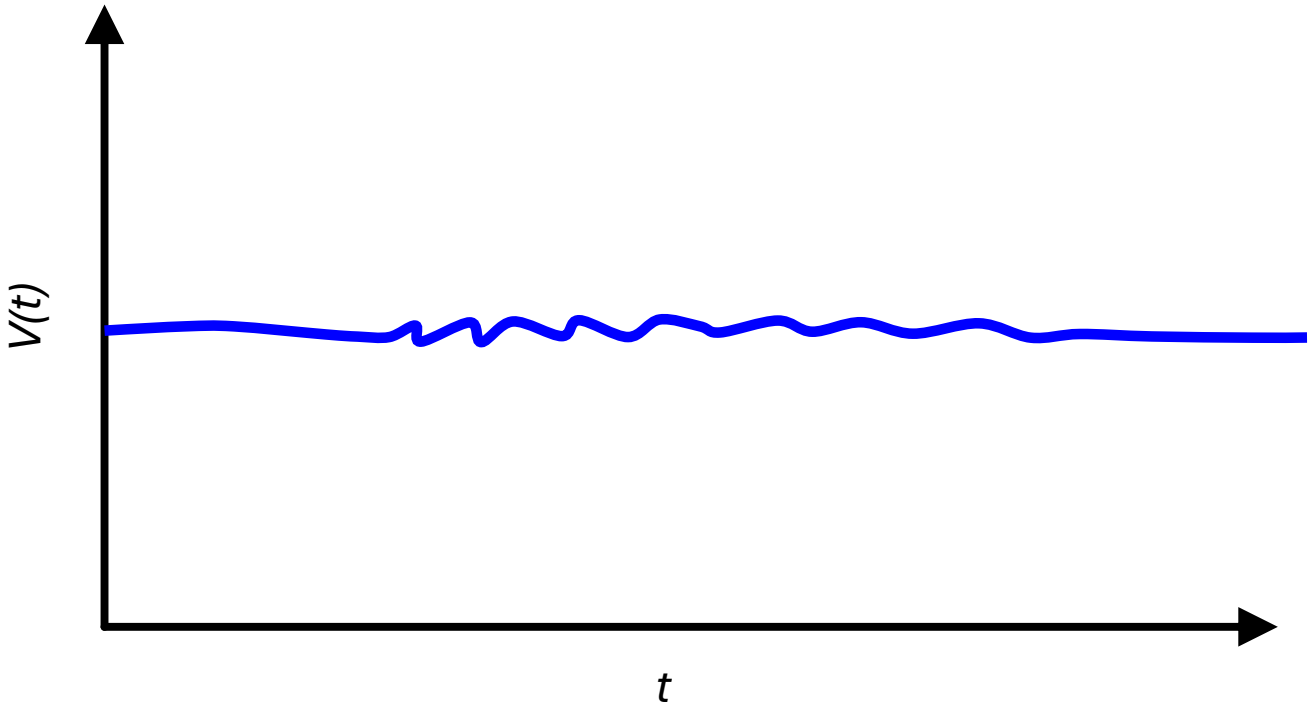
Compare to original… Did not Capture the high-frequency Wiggles!



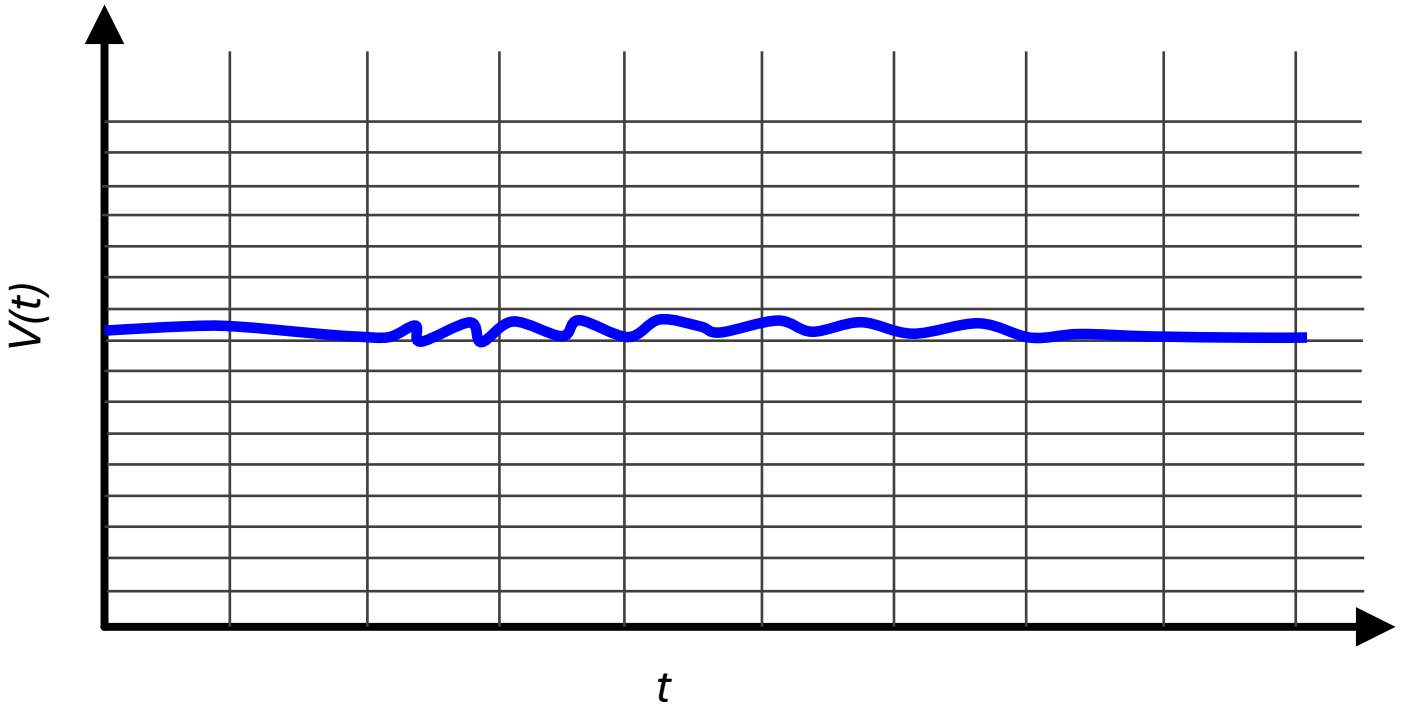
v[n] = [9,11,5,7,5,12,10,7,5,4,]

Potentially Bad Discretization Error

Continuous in Value and in Time

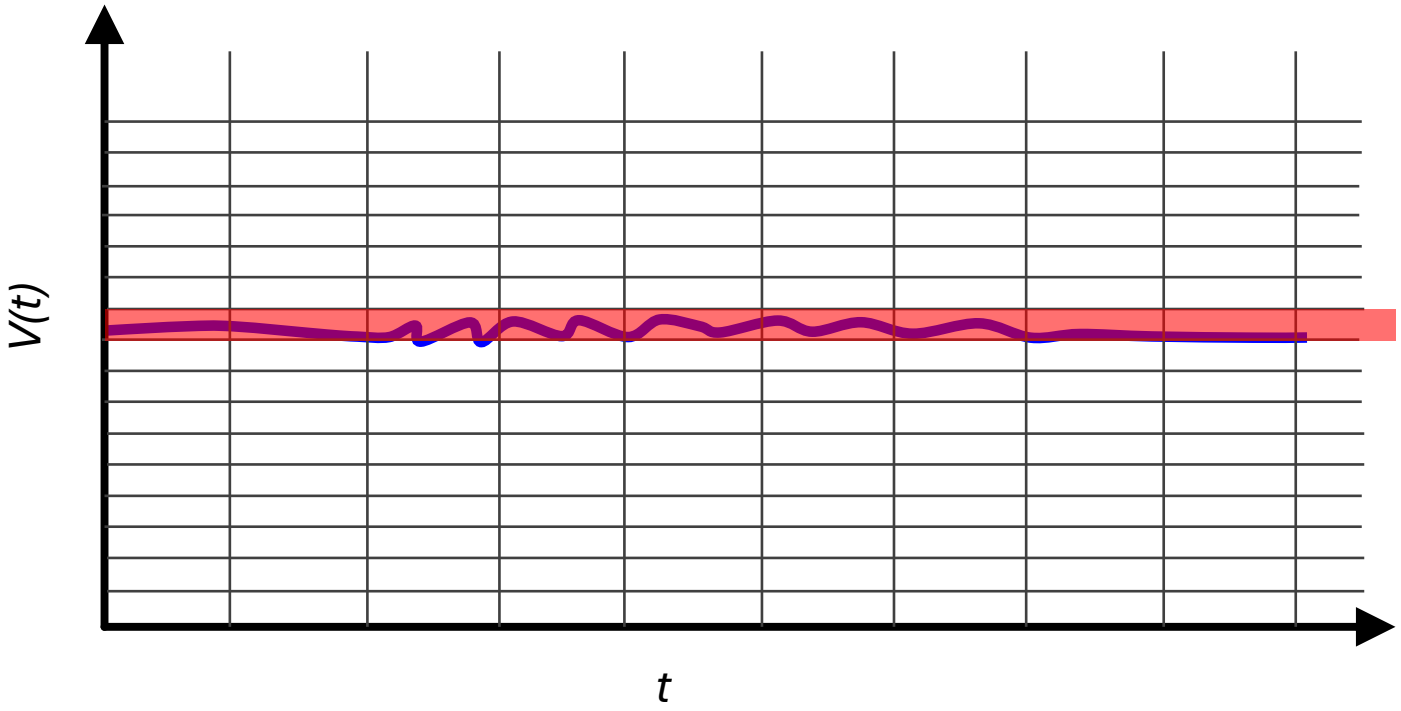


Discretization in Time and **Quantization** in Value



4 bit value encoding

Discretization in Time and **Quantization** in Value



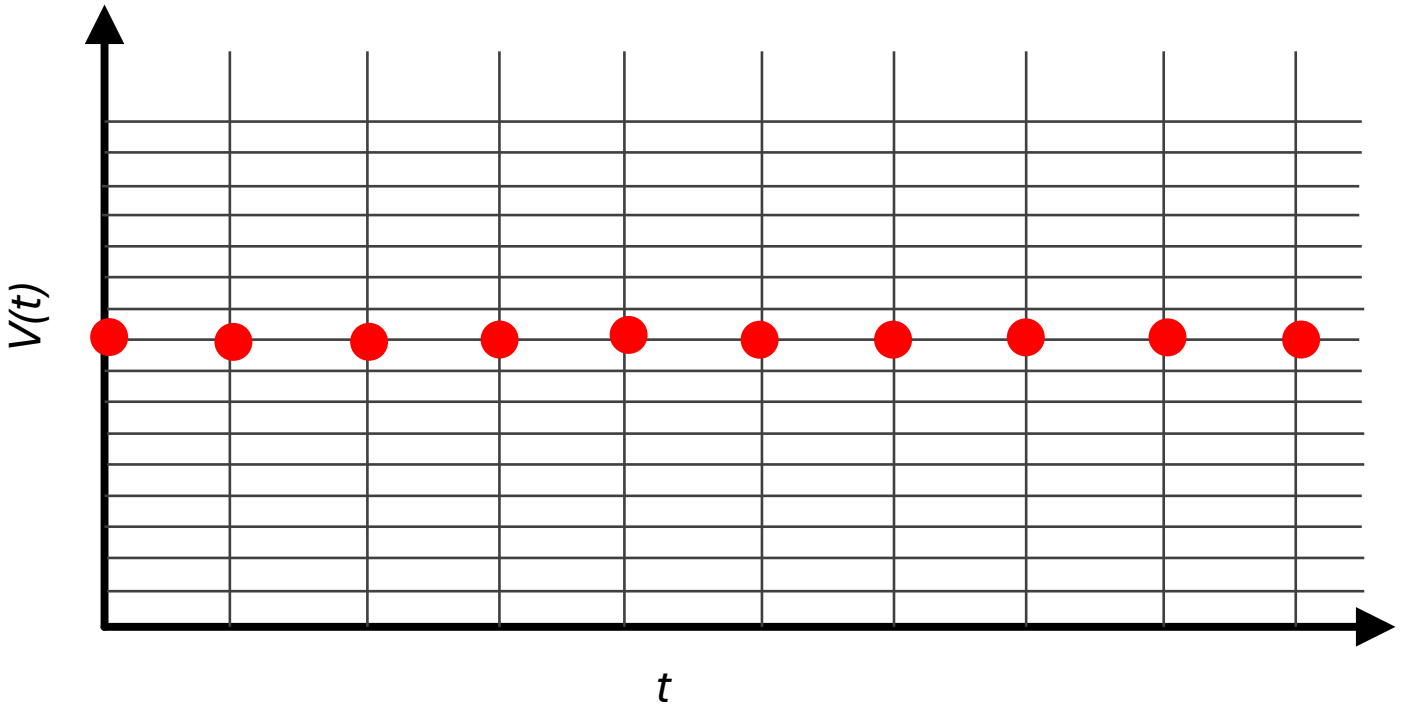
v[n] = [9,9,9,9,9,9,9,9,9,9]

4 bit value encoding

Store in memory

- v[n] = [9,9,9,9,9,9,9,9,9,9]
- 10 4-bit values: need 40 bits in memory!
- Great. All is good.

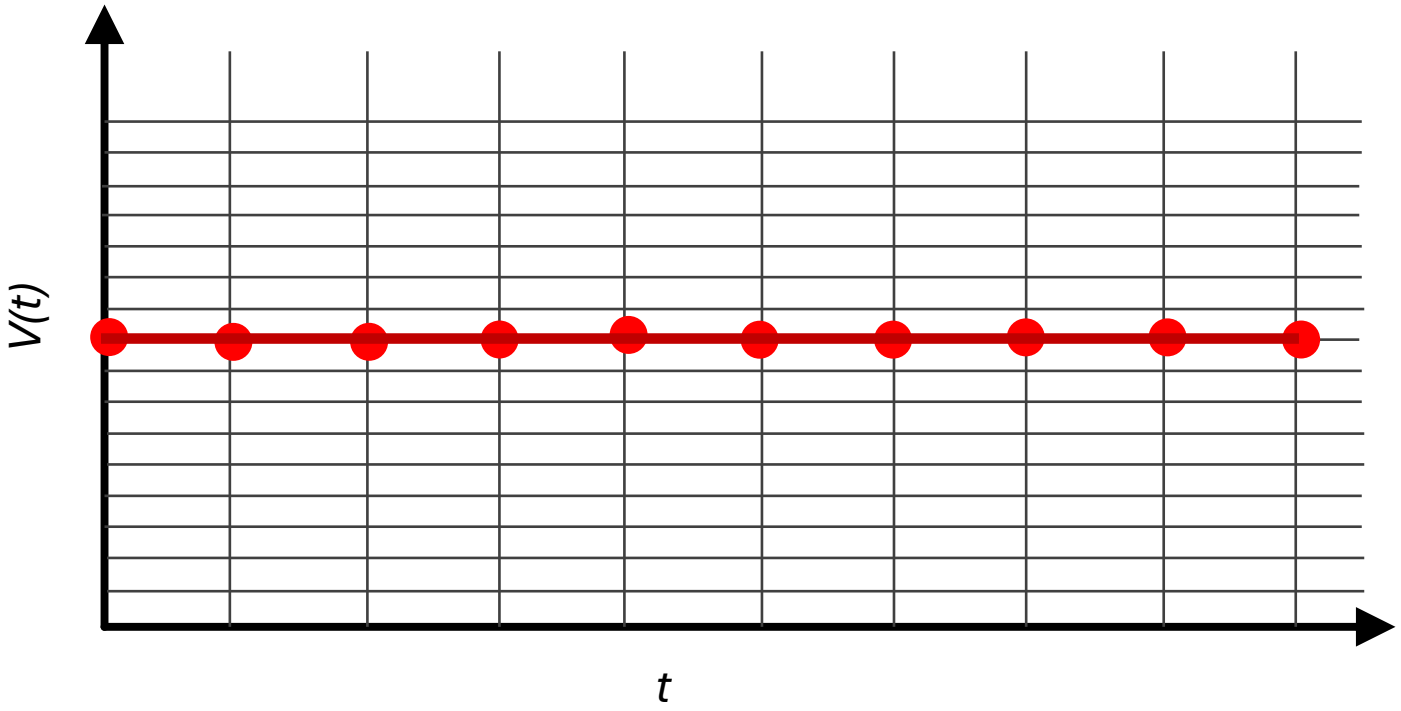
Reproduce



v[n] = [9,9,9,9,9,9,9,9,9,9]

4 bit value encoding

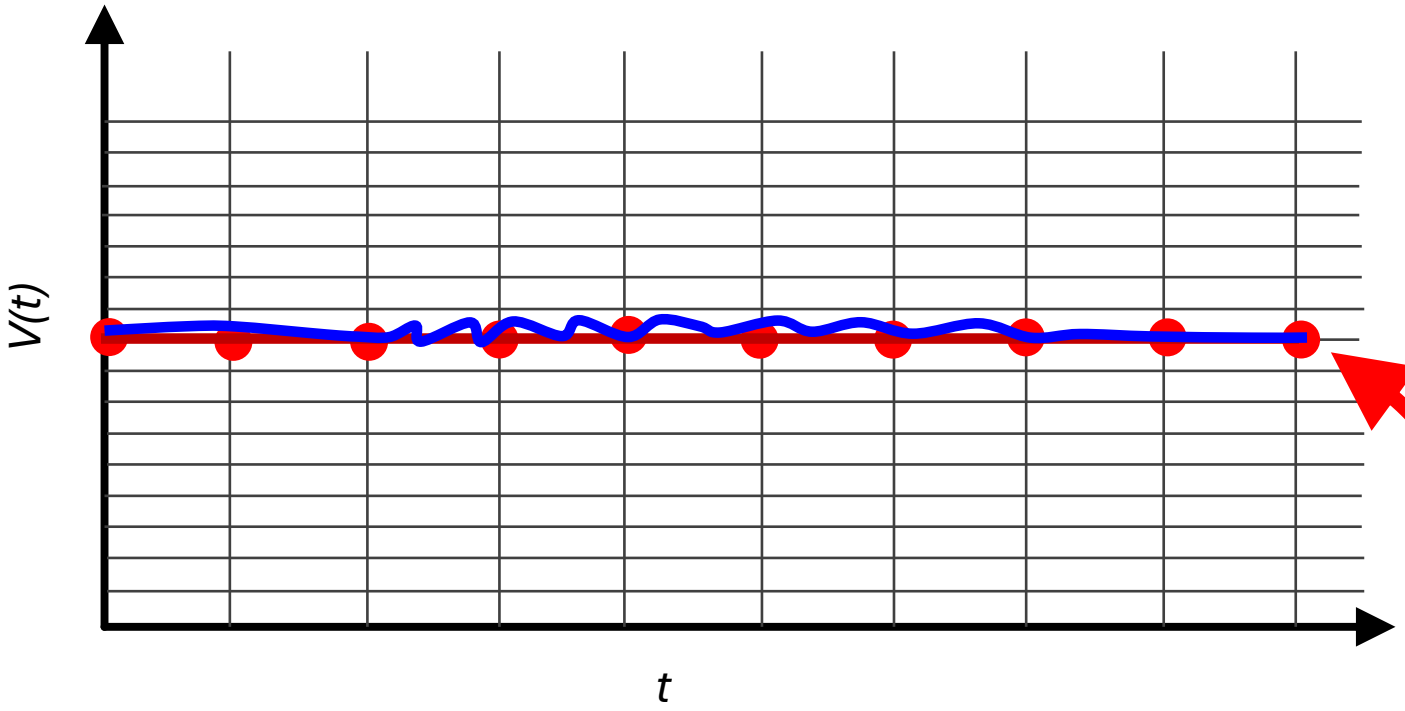
Reproduce



v[n] = [9,9,9,9,9,9,9,9,9,9]

4 bit value encoding

Compare… to original also meh



v[n] = [9,9,9,9,9,9,9,9,9,9]

Potentially Really Problematic

Quantization Error!

Those tiny wiggles might be really important in certain contexts! Tiny heartbeats!

Conclusions

- Care must be taken when choosing what rate you sample (**discretize**) your signal and at what bit-depth you **quantize** your sample
- There's no right answer, since it depends on context/use cases.
- Ideally want to sample at high rate and quantize with many bits…
- But taken to the extreme this uses a lot of resources (lots of memory and resources/lots of bits) so downward pressure on choices

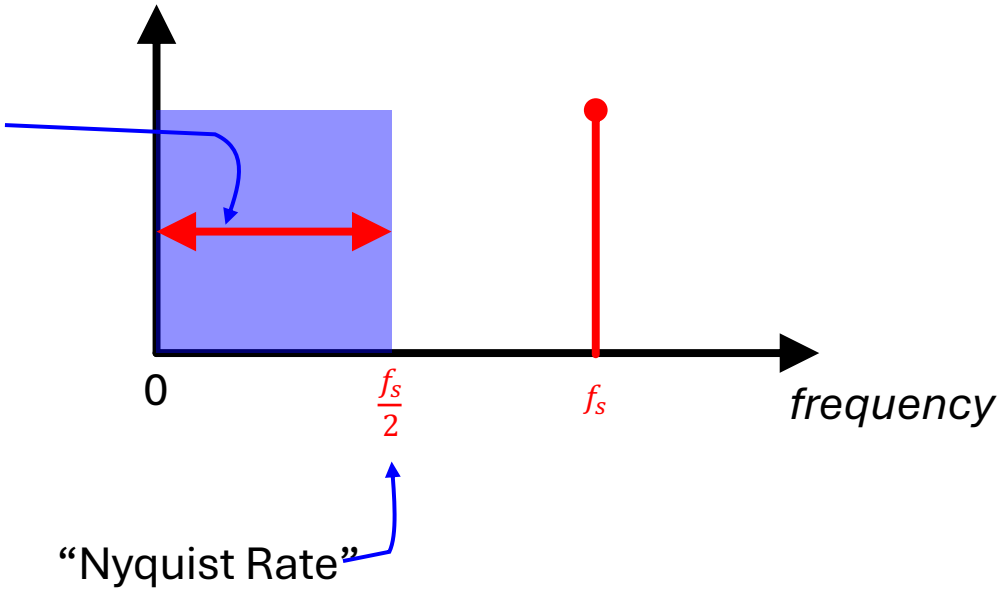
Is that all there is to it?

- No, it is wayyy more complicated
- Let’s just consider sample rate for right now

Sample Rate

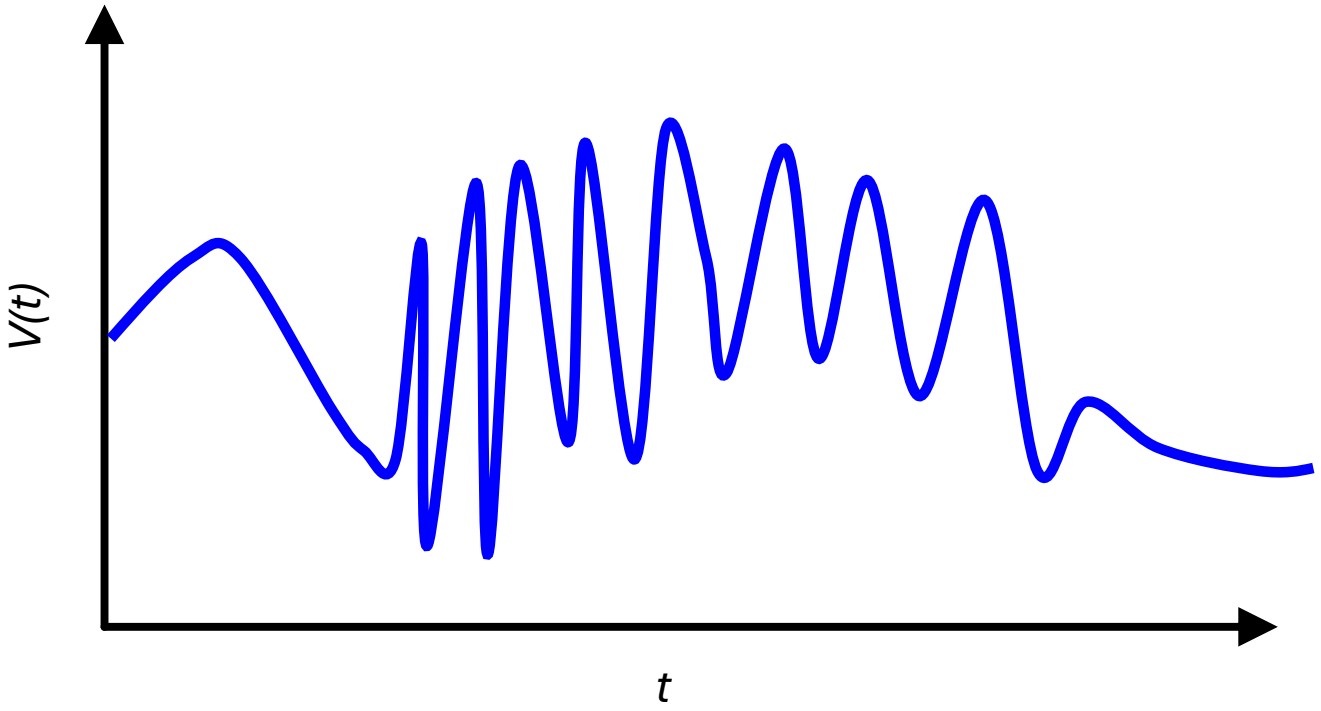
- How frequently we sample our signal directly influences what we can effectively capture.
- A sample rate of f_s is only capable of expressing signals with frequencies less than $\frac{f_s}{2}$

*Signals with frequencies in this region of the spectrum can be **fully captured***

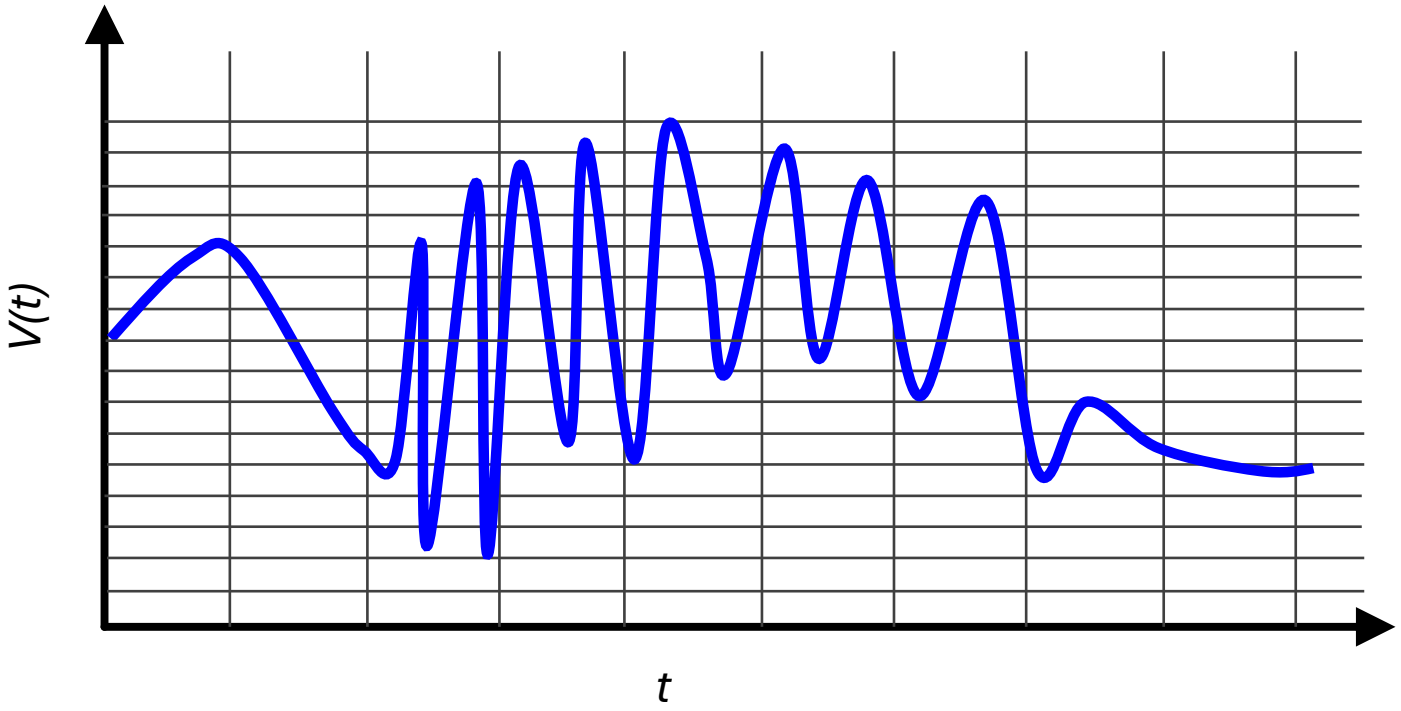


Nyquist, Shannon, few others showed this in the 1930s

Let’s consider this situation though….

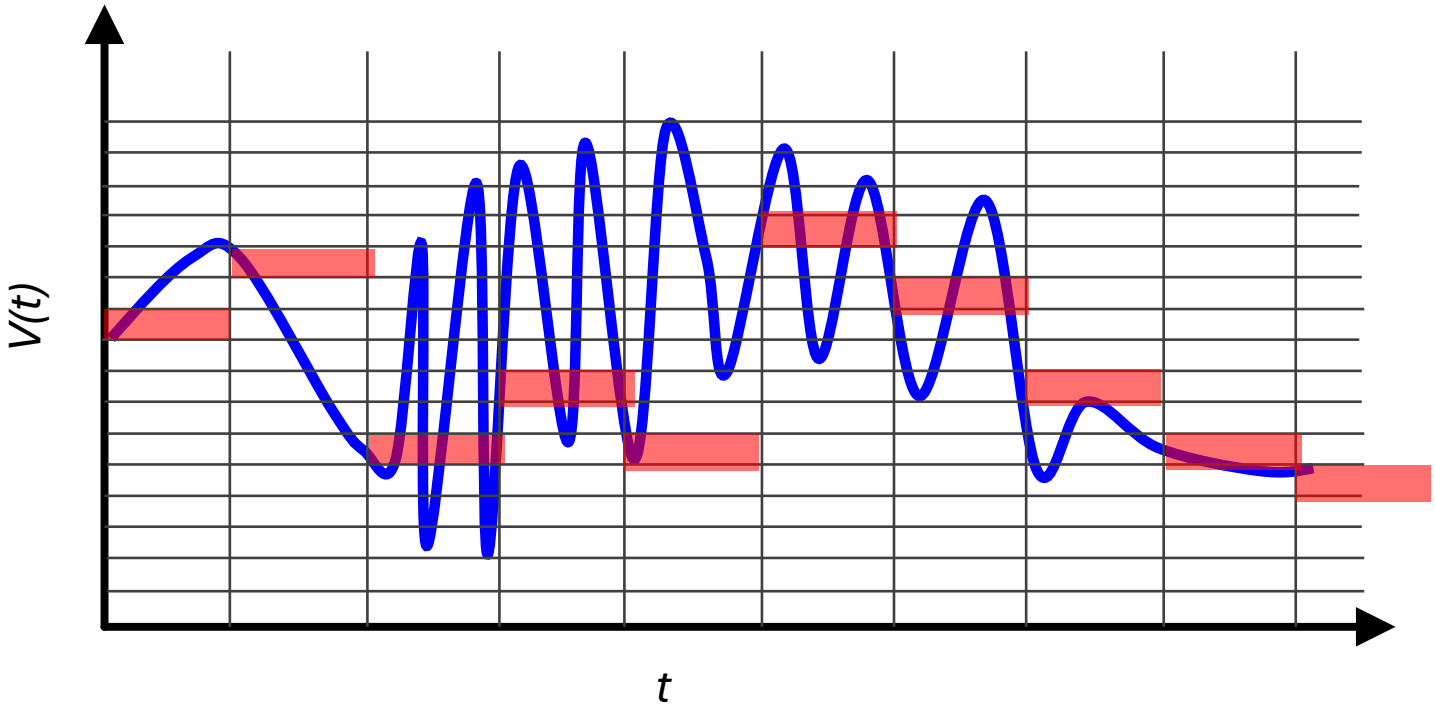


Let’s digitize it…at this sample rate we shouldn’t be able to capture it



4 bit value encoding

Discretization in Time and **Quantization** in Value



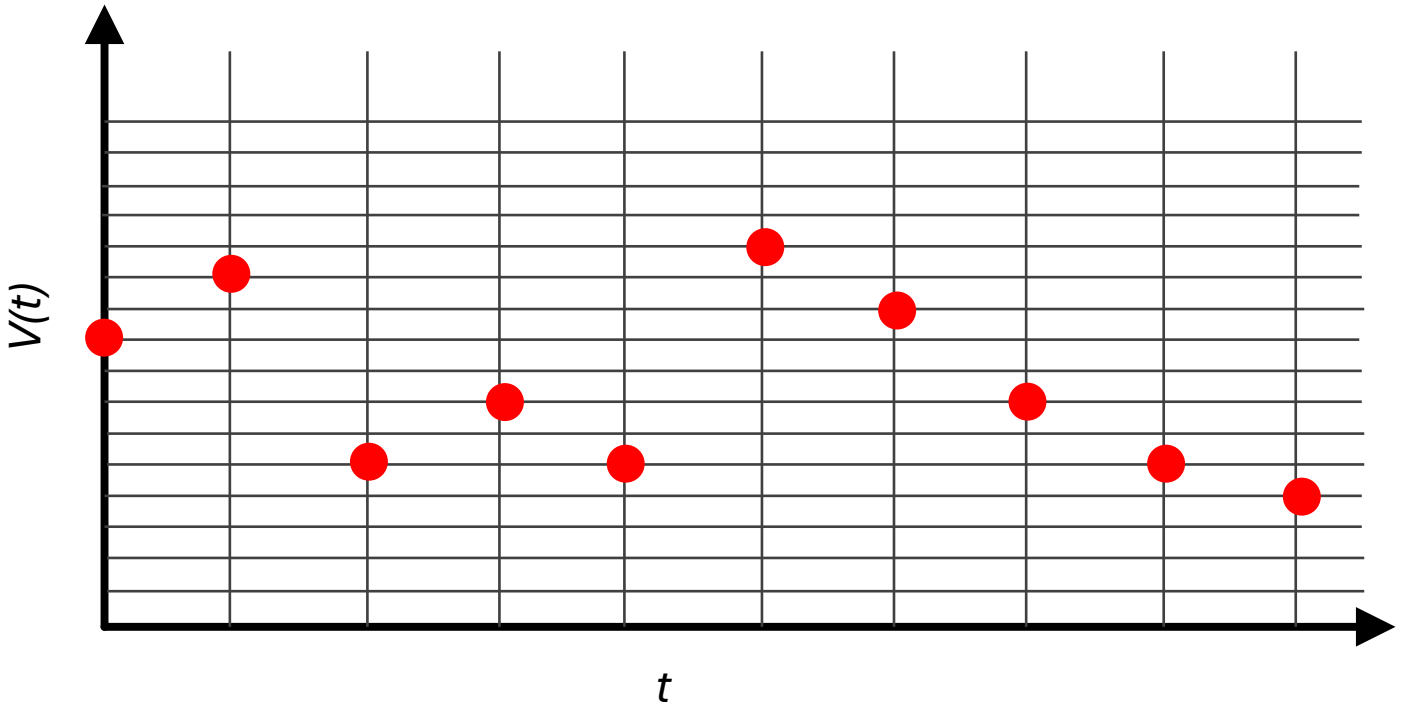
v[n] = [9,11,5,7,5,12,10,7,5,4,]

4 bit value encoding

Store in memory

- v[n] = [9,11,5,7,5,12,10,7,5,4,]
- 10 4-bit values: need 40 bits in memory!
- Easy-peasy one-two-threesy

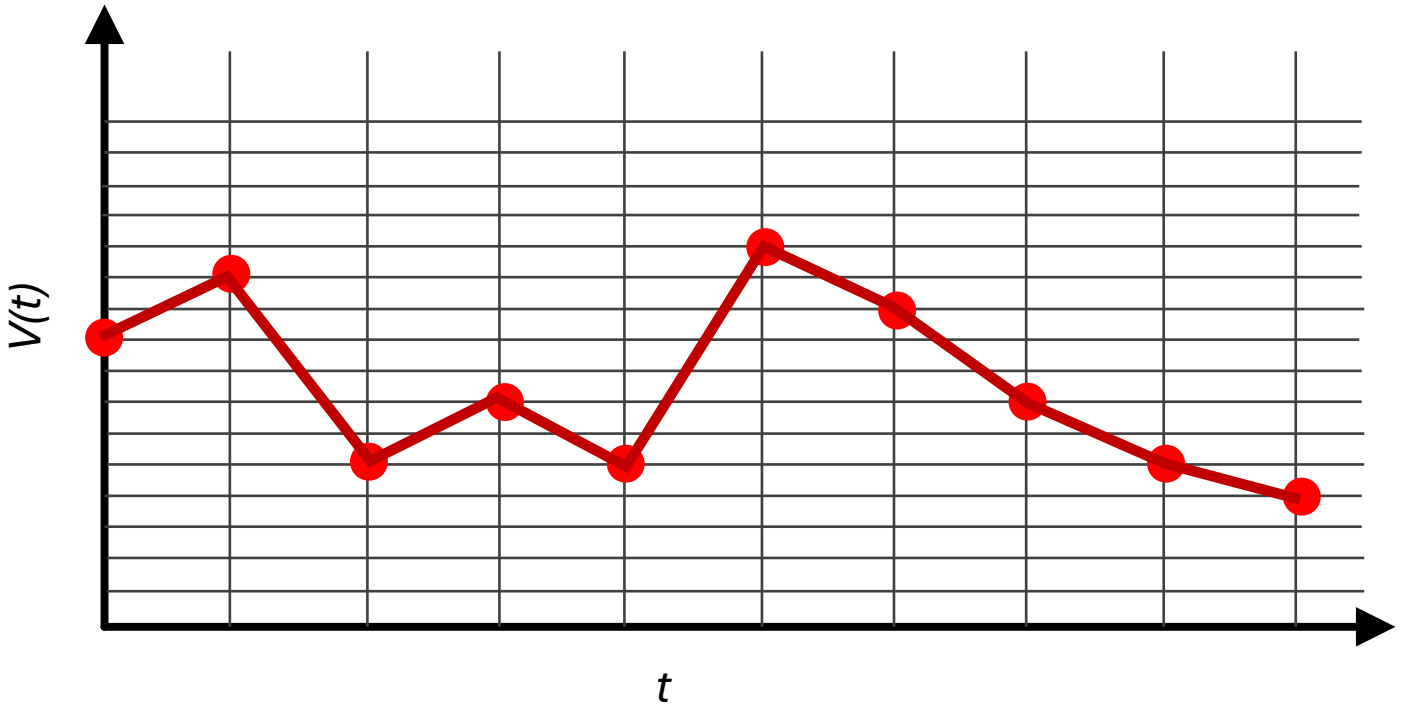
Reconstruct



v[n] = [9,11,5,7,5,12,10,7,5,4,]

4 bit value encoding

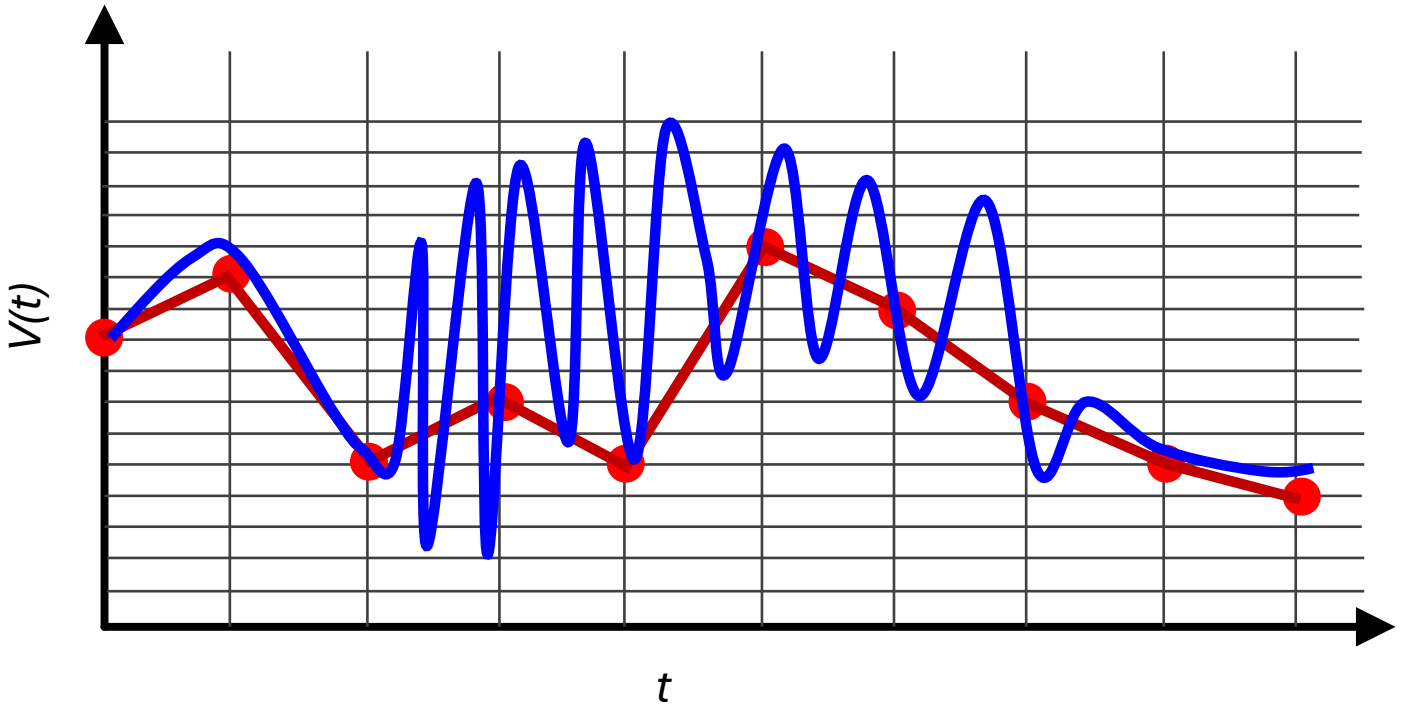
Reproduce



v[n] = [9,11,5,7,5,12,10,7,5,4,]

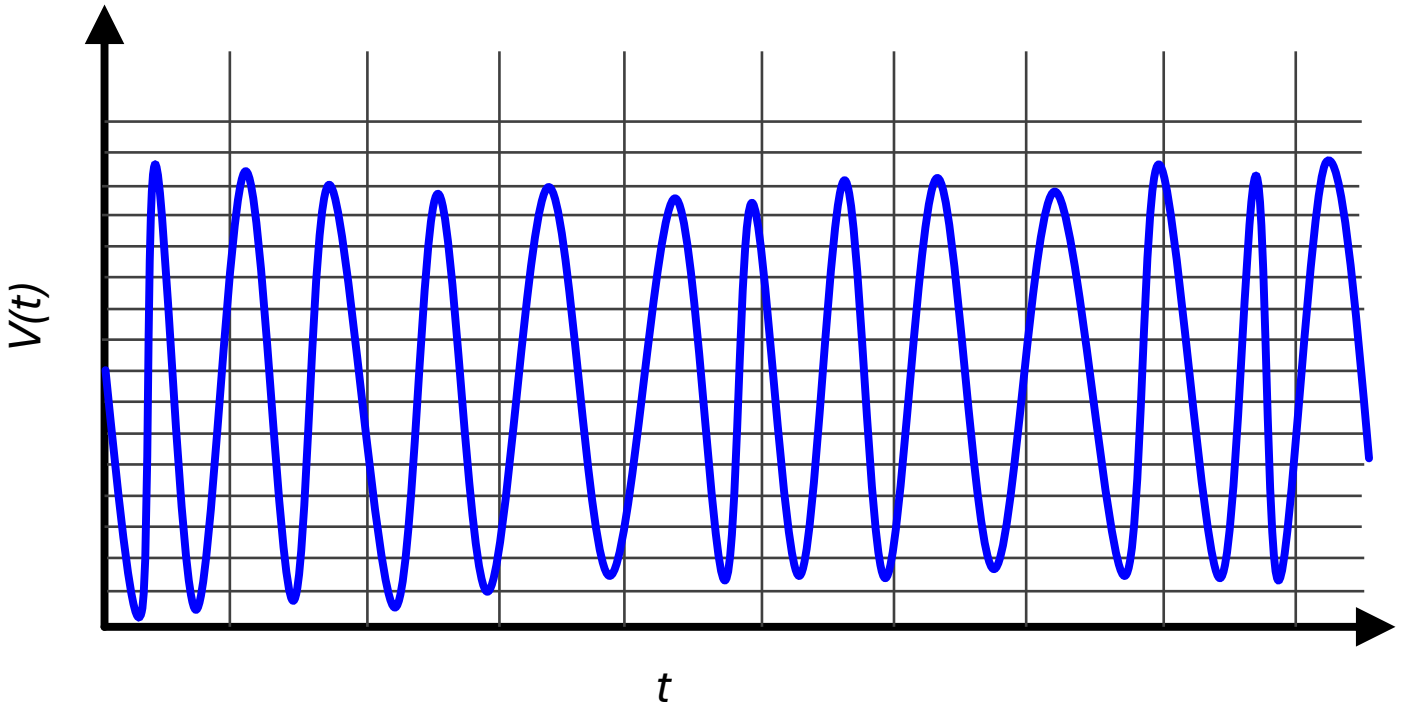
4 bit value encoding

Compare to original… Did not Capture the high-frequency Wiggles!

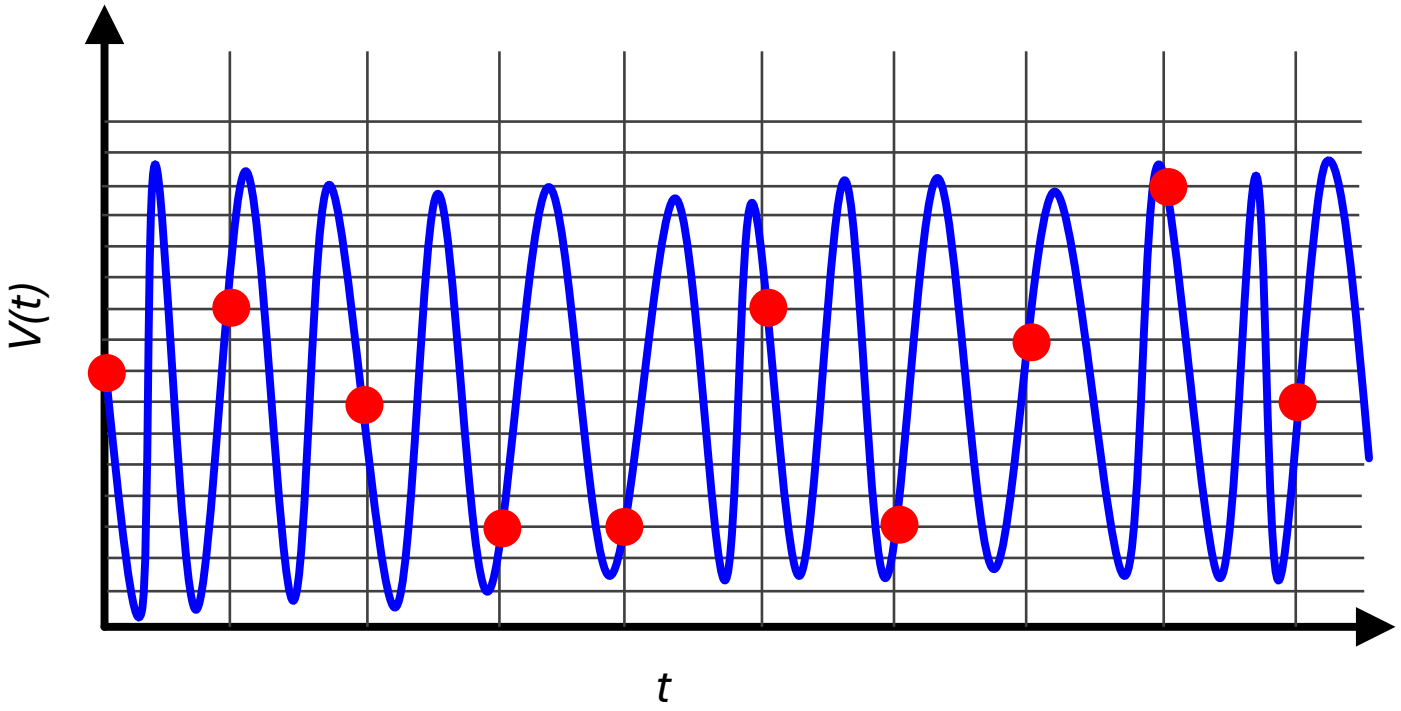


*Great….but we still captured something! What **<u>is</u>** that signal expressed by the red interpolation?*

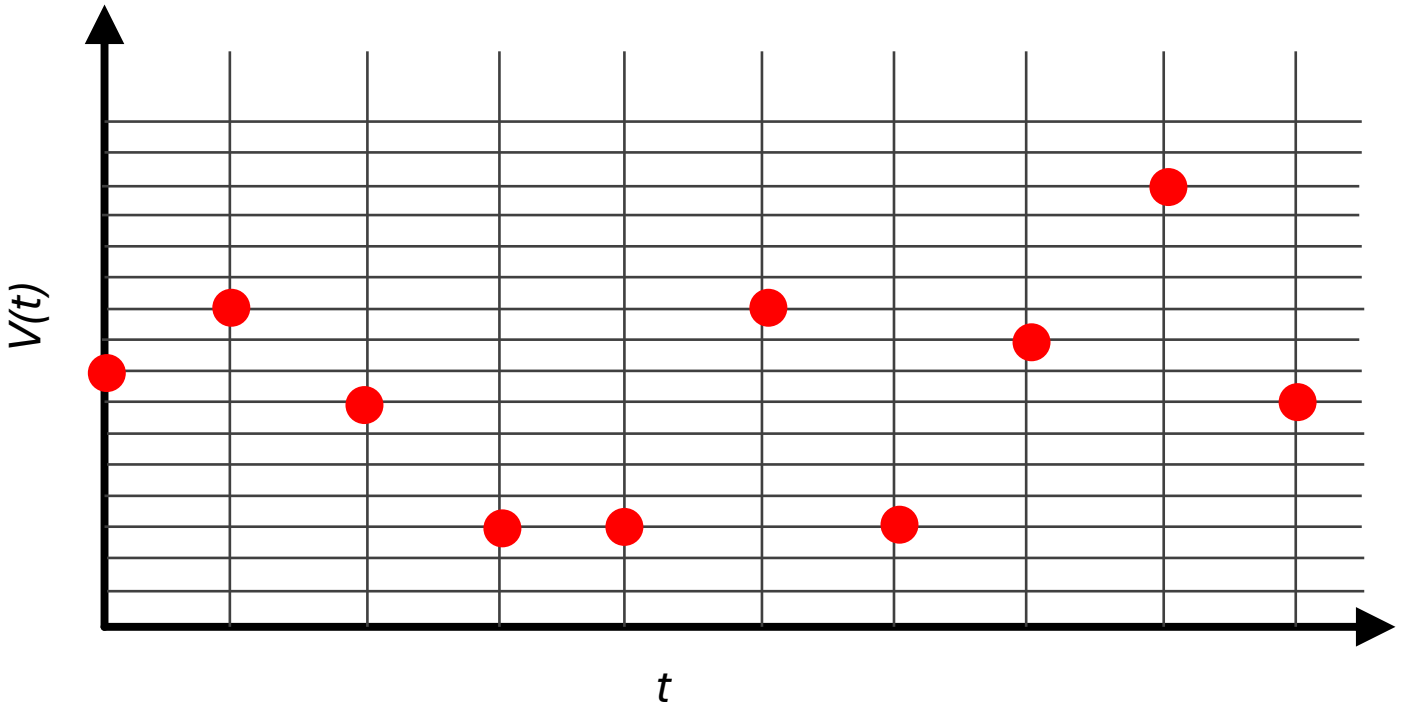
Or....consider this...



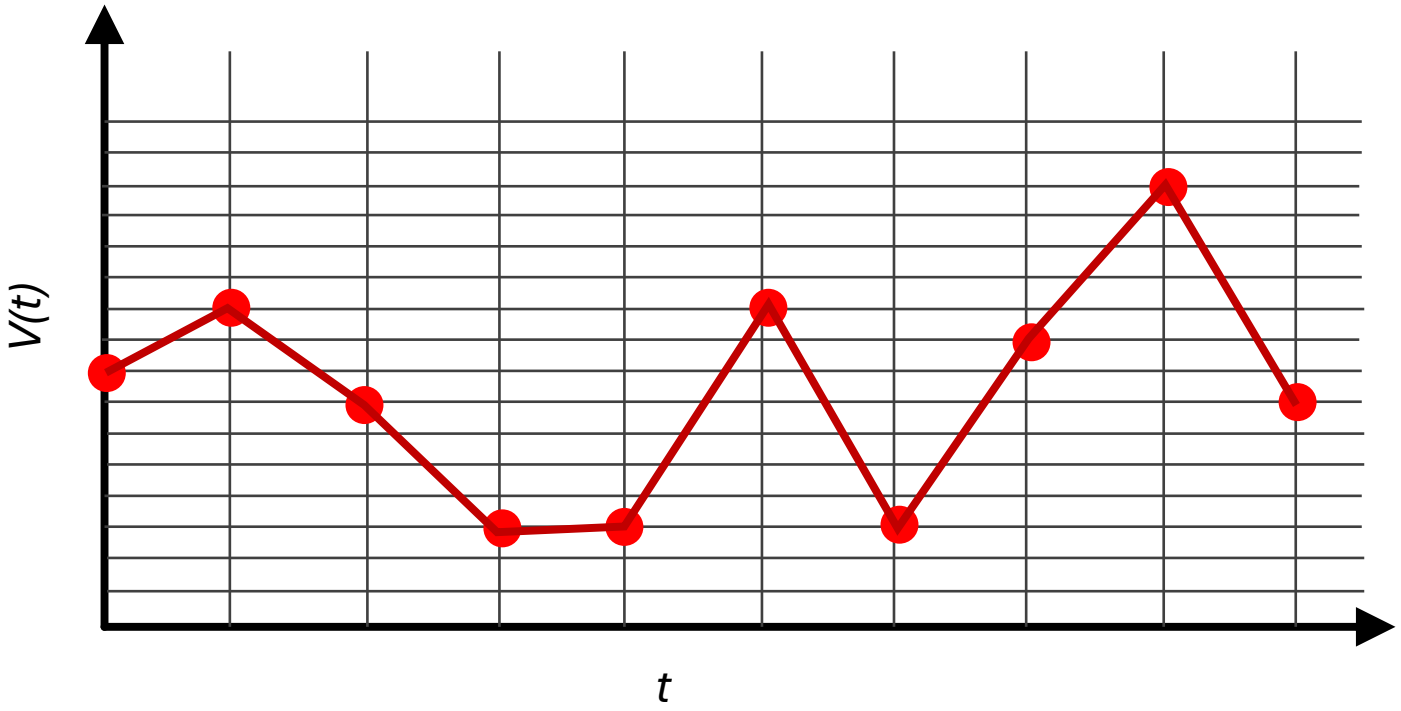
Sample it...



Store it…



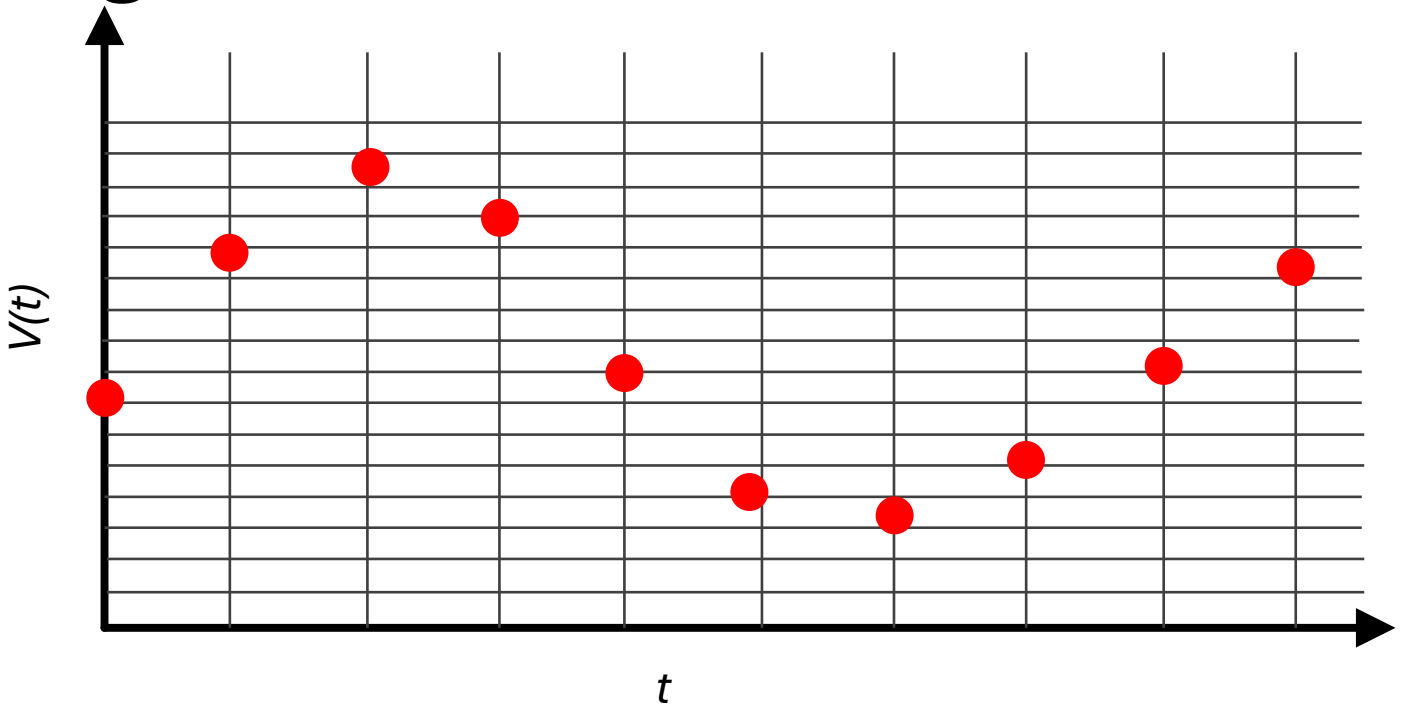
Reconstruct it…



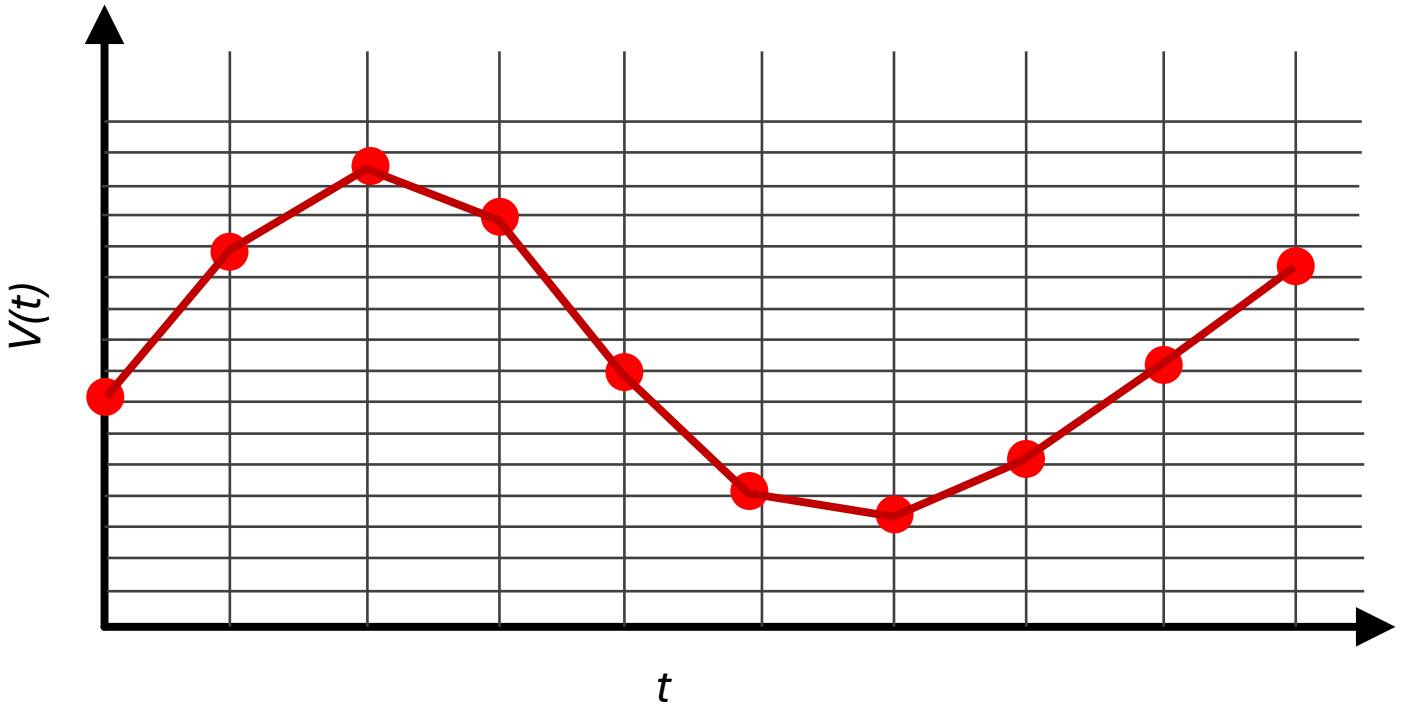
We’ve created a a different signal from what was before! WTH?

Or Consider this…

if we start with this data, knowing nothing else……

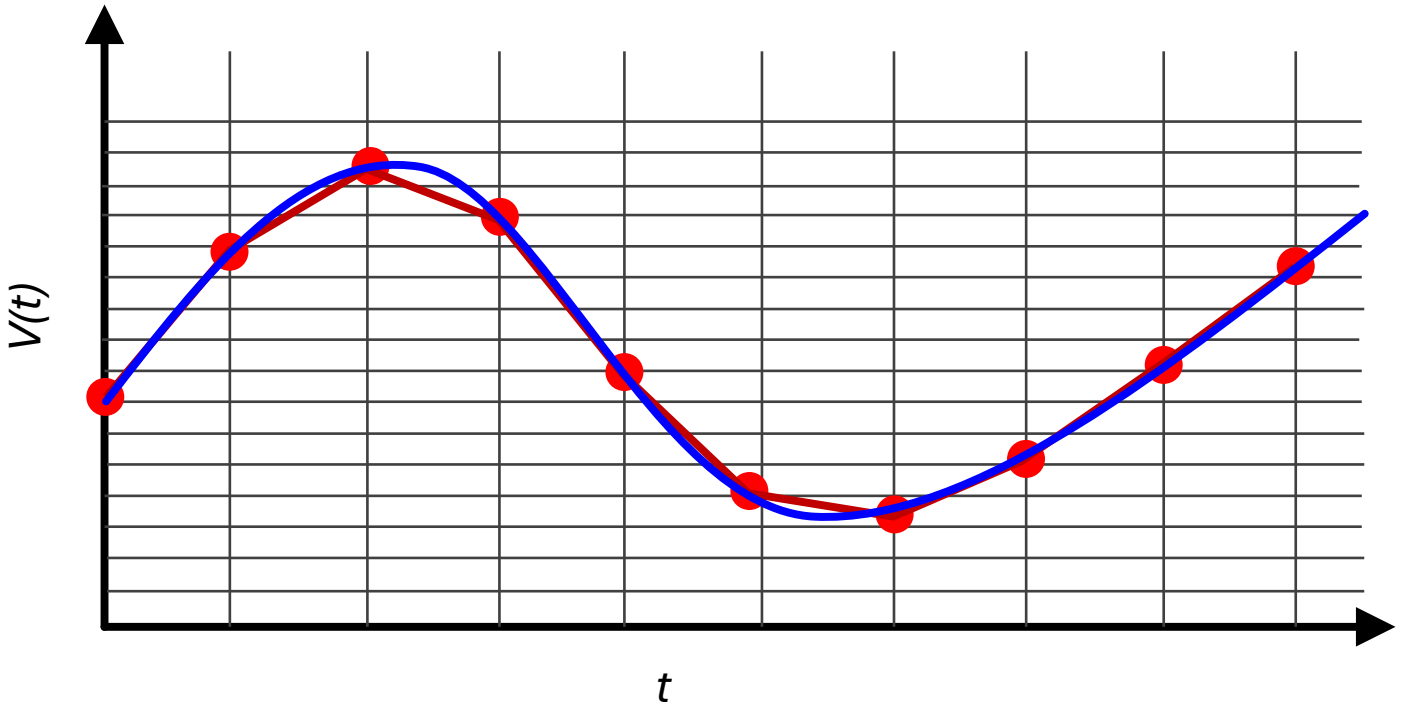


And we Reconstruct the signal…is this ok?

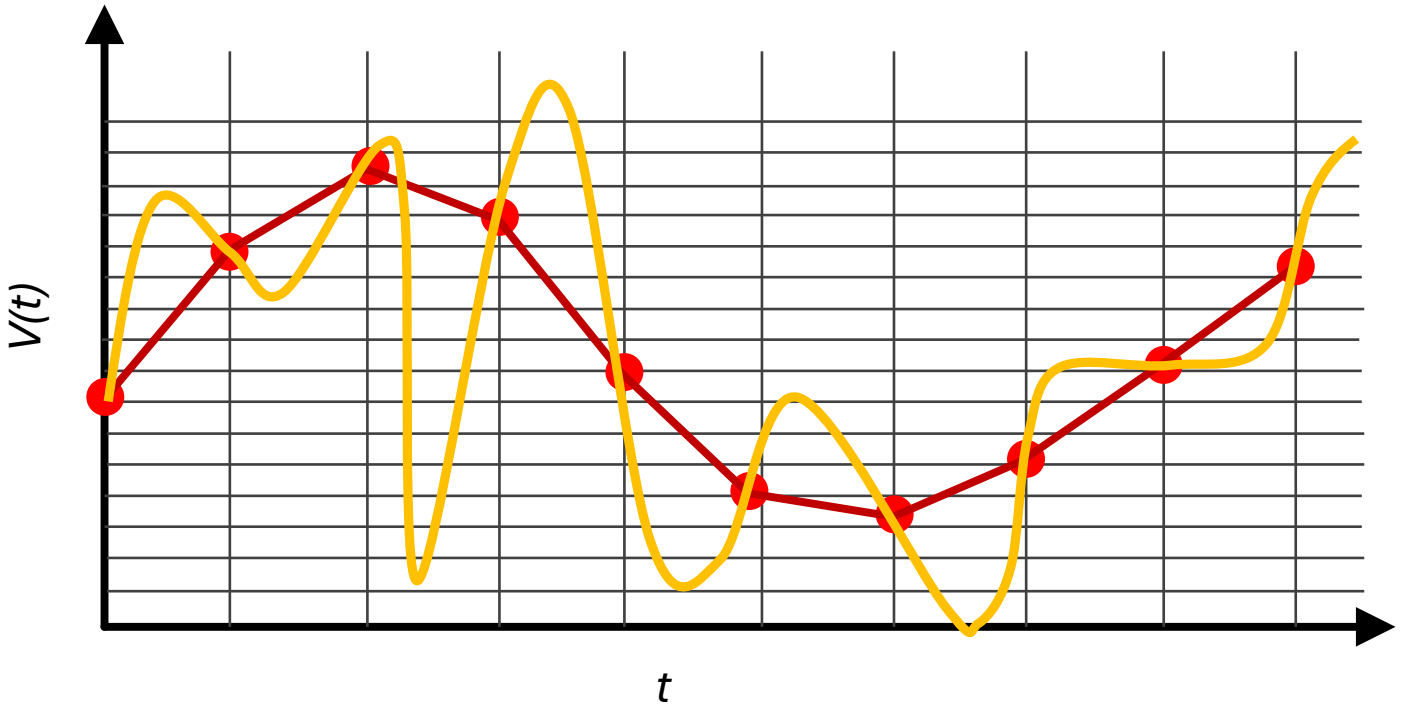


First-order hold (connect-the dots)

If it came from this, ok… but…

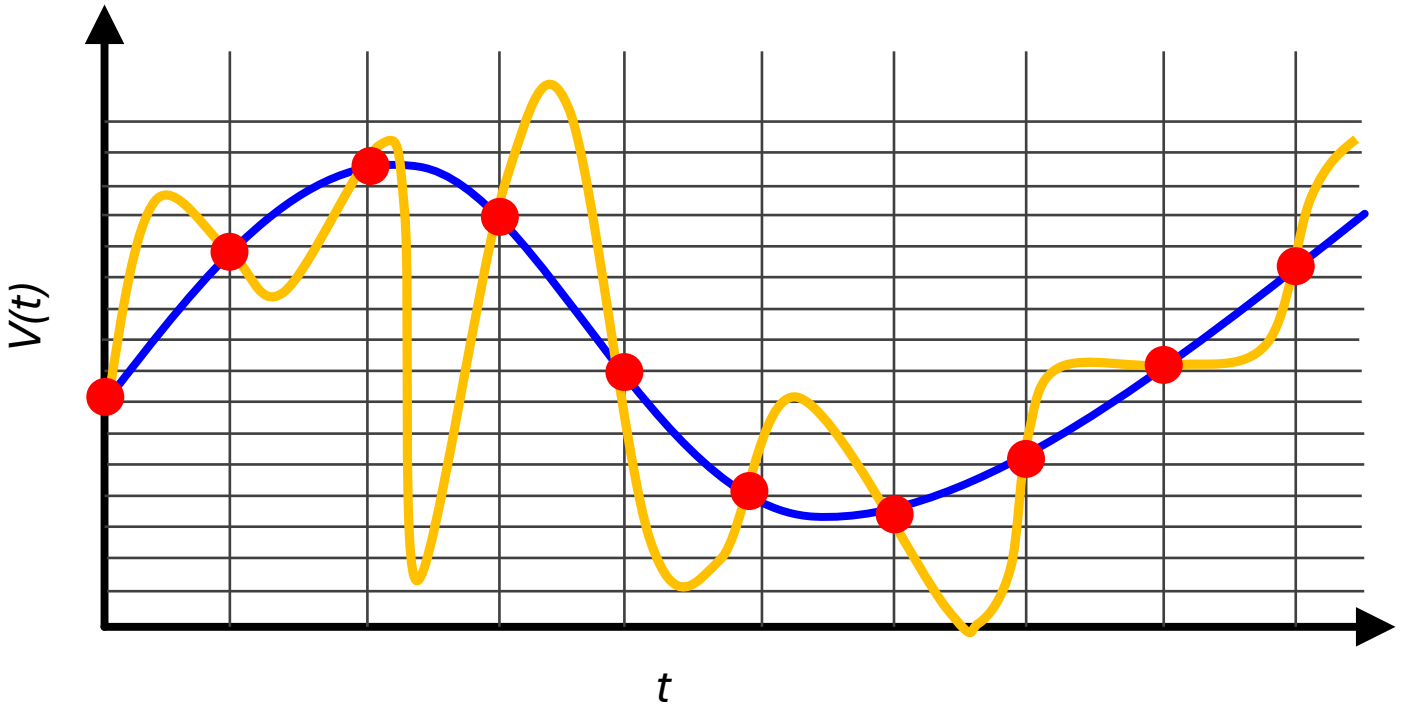


It could have also come from this…Uh oh



First-order hold (connect-the dots)

Which one Made the Signal



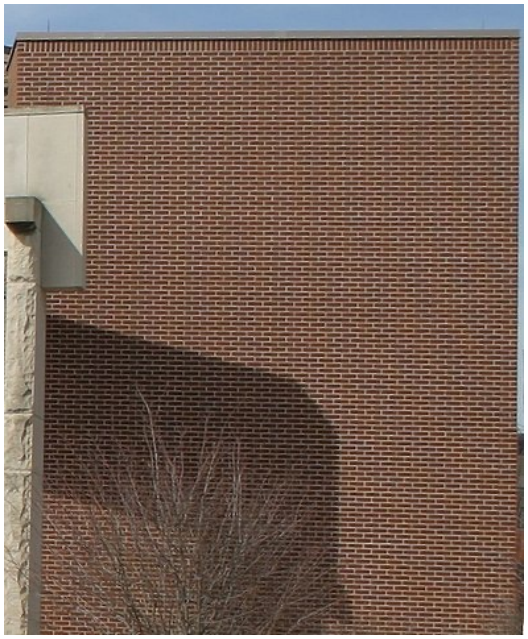
There's ambiguity in what those samples could represent…that means it really doesn't convey much, if any, information

Aliasing

- While we can't fully capture and reproduce signals with a frequency higher than the Nyquist sampling rate, it doesn't mean they **won't** have an impact!
- Energy from that high frequency will leak into the frame…a form of "spectral leakage"
- A sample rate of f_s can fully capture all information in a signal if and only if, the highest frequency in that signal is at or below $\frac{f_s}{2}$!
- **If you don't do this**, aliasing will appear (higher frequencies appear as a different signal (an "alias")) that can be expressed with the sample rate

Aliasing Can Happen in Space too

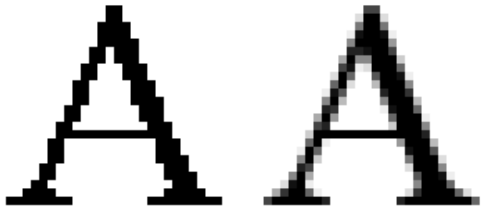
- Just like there are temporal frequencies (in time), images have spatial frequencies.
- Same issues arise!



Anti-alias Filtered

Not Anti-alias Filtered

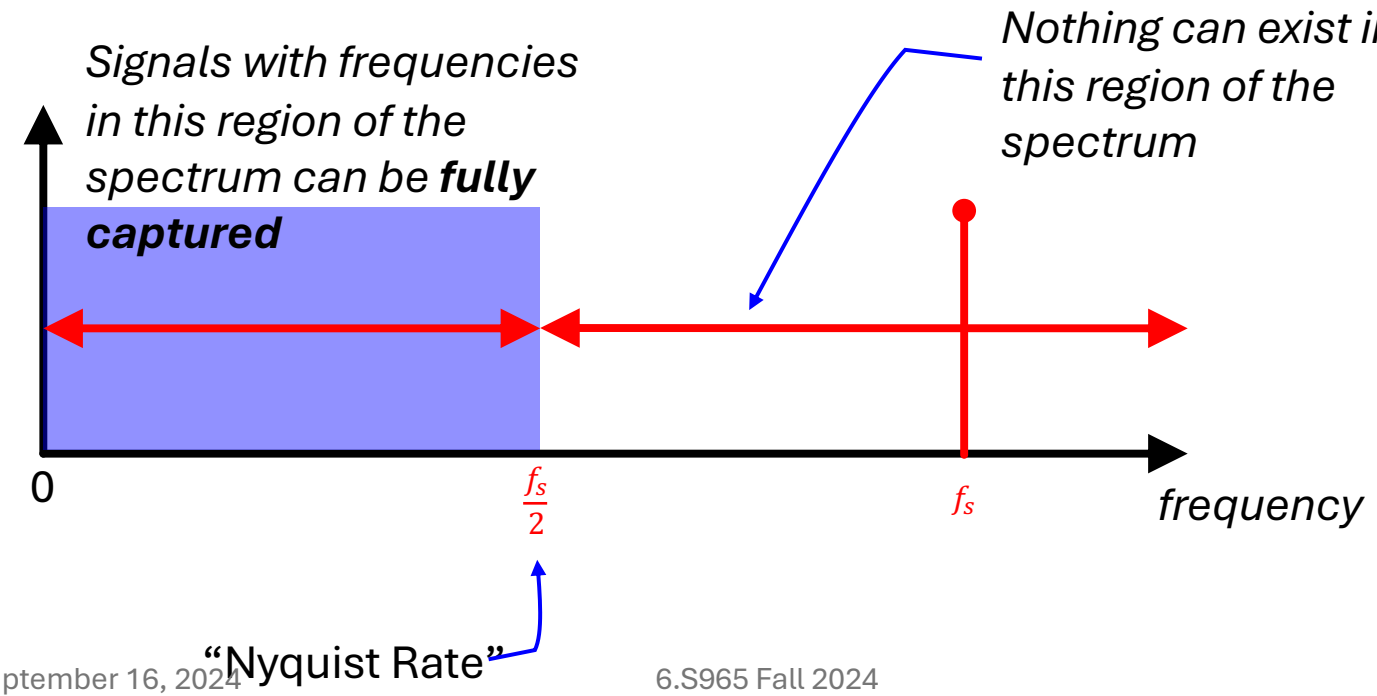
https://en.wikipedia.org/wiki/Aliasing



This font has been processed with an anti-alias filter to prevent artifacts when displayed

Solution

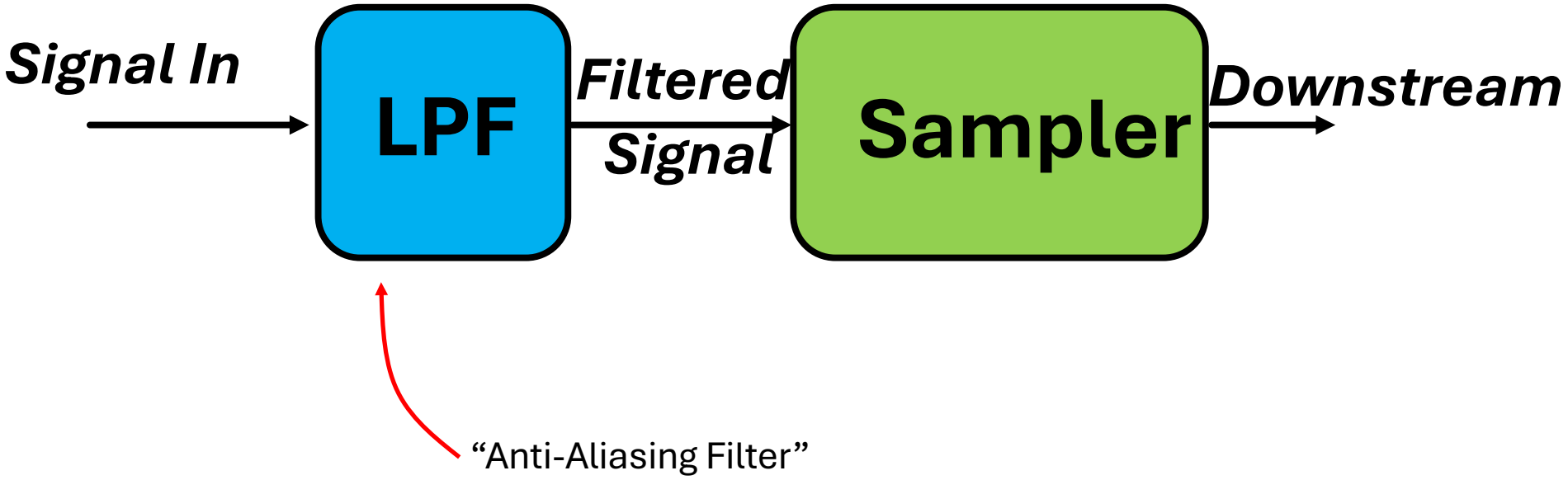
- The **ONLY** way to guarantee that a set of discrete points can unambiguously represent a signal is to guarantee that prior to sampling, we remove **all energy** that it exists in frequencies higher than the Nyquist Sampling Rate
- To do this we need a Low-Pass Filter!



There are exceptions

Low Pass Filter

- Prior to Sampling, we must be sure that our signal has no significant energy above our Nyquist Rate



How Do You Actually Make a Filter?

- Several types of filters. Two big ones:
  - **IIR**: Infinite Impulse Response:
    - Uses past output history for filtering
  - **FIR**: Finite Impulse Response:
    - Uses input history for filtering
  - CIC: Cascaded Integrator Comb:
    - Special case of FIR mixed with down-samplers/decimators

Filters

- ***Stateful*** systems that analyze history signals to select for particular signal attributes:
  - **Low-pass Filter:** Lets through low-frequency signals
  - **High-pass Filter:** Lets through high-frequency signals
  - **Band-pass Filter:** Lets through selective group of frequencies
  - **Band-stop Filter:** Blocks selective group of frequencies
  - **Matched-Filter:** Values come from time-series of feature of interest being convolved with signal

Infinite Impulse Response Filter (IIR)

$$y[n] = \alpha \cdot y[n-1] + \beta \cdot x[n]$$

- The current output ($y[n]$) of the filter is based on the weighted sum of the previous output ($y[n-1]$) of the filter + the value of the input ($x[n]$)*
- Sometimes called a recursive filter: “y is based off of y is based off of y…”
- Information enters the system through x but its influence on the output is dependent on the values of α and β

*can also be based on multiple past values of y and x

Infinite Impulse Response (Modified)

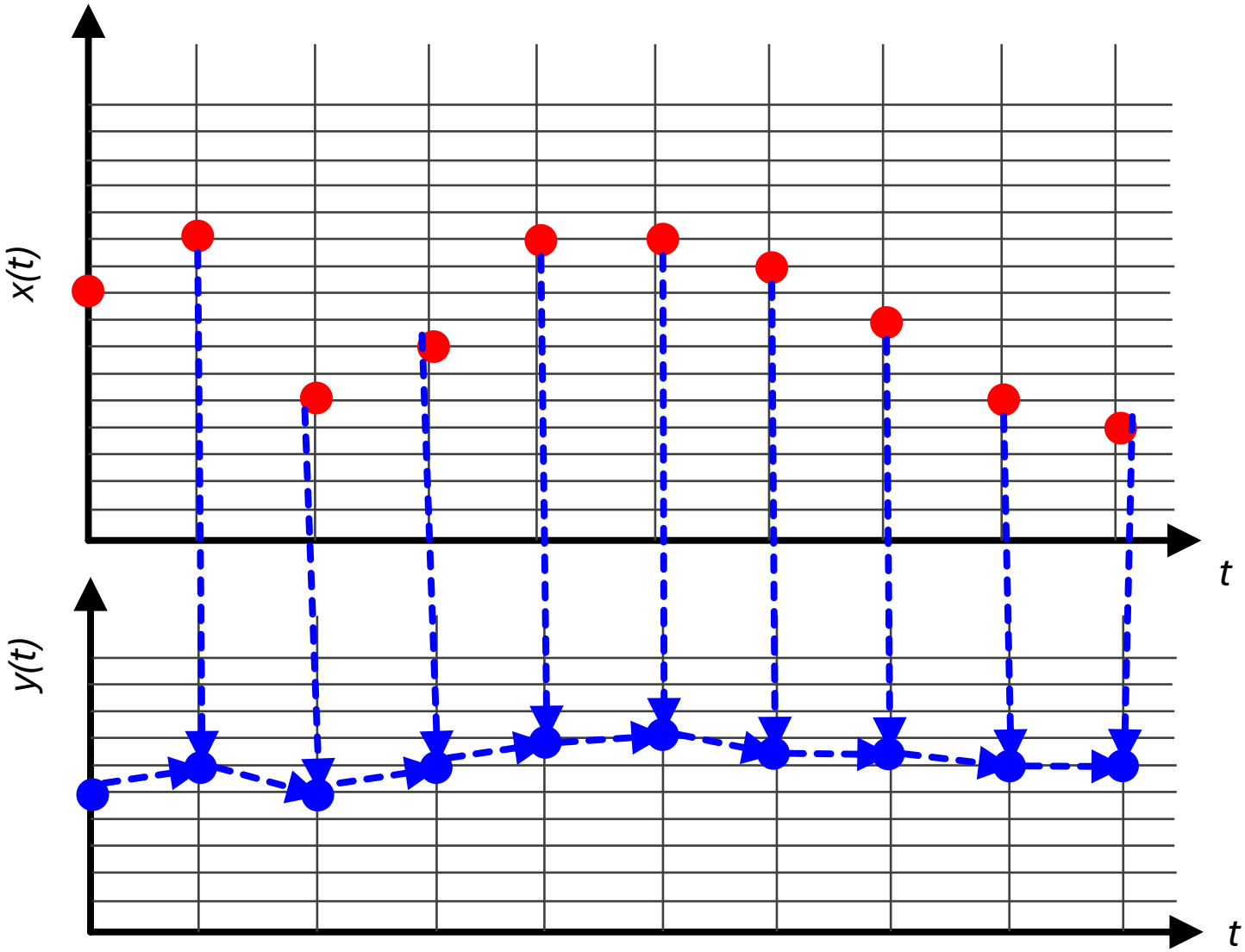
$$y[n] = \alpha \cdot y[n-1] + (1-\alpha) \cdot x[n]$$

$$0 \leq \alpha \leq 1$$

- Fix the relationship of the new input and old output to one variable α :
  - As $\alpha \to 1$ input has less weight (takes time for it to affect output…blocks more high frequency events)
  - As $\alpha \to 0$ input has more weight (output quickly follows input…allows through more high frequency events (and everything actually)

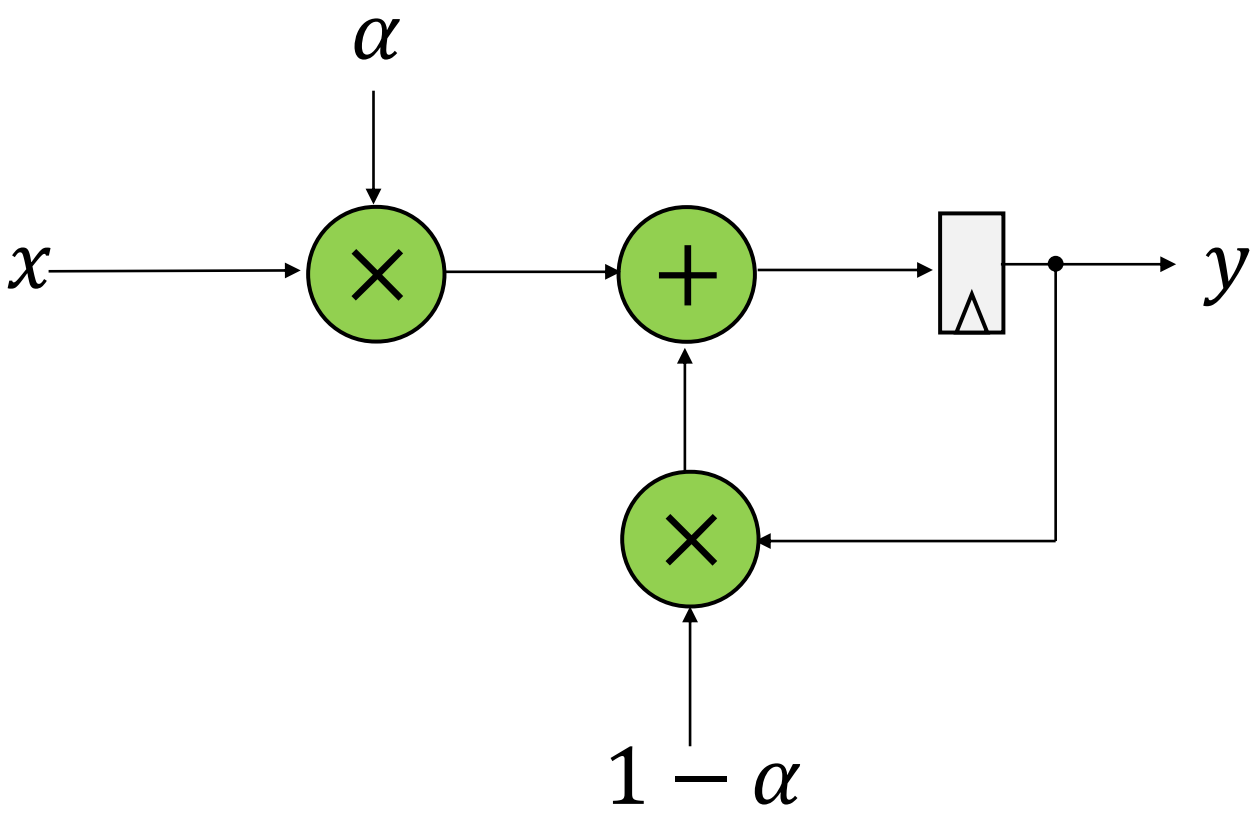
IIR Filter

$$y[n] = \alpha \cdot y[n-1] + (1-\alpha) \cdot x[n]$$



Infinite Impulse Response (Modified)

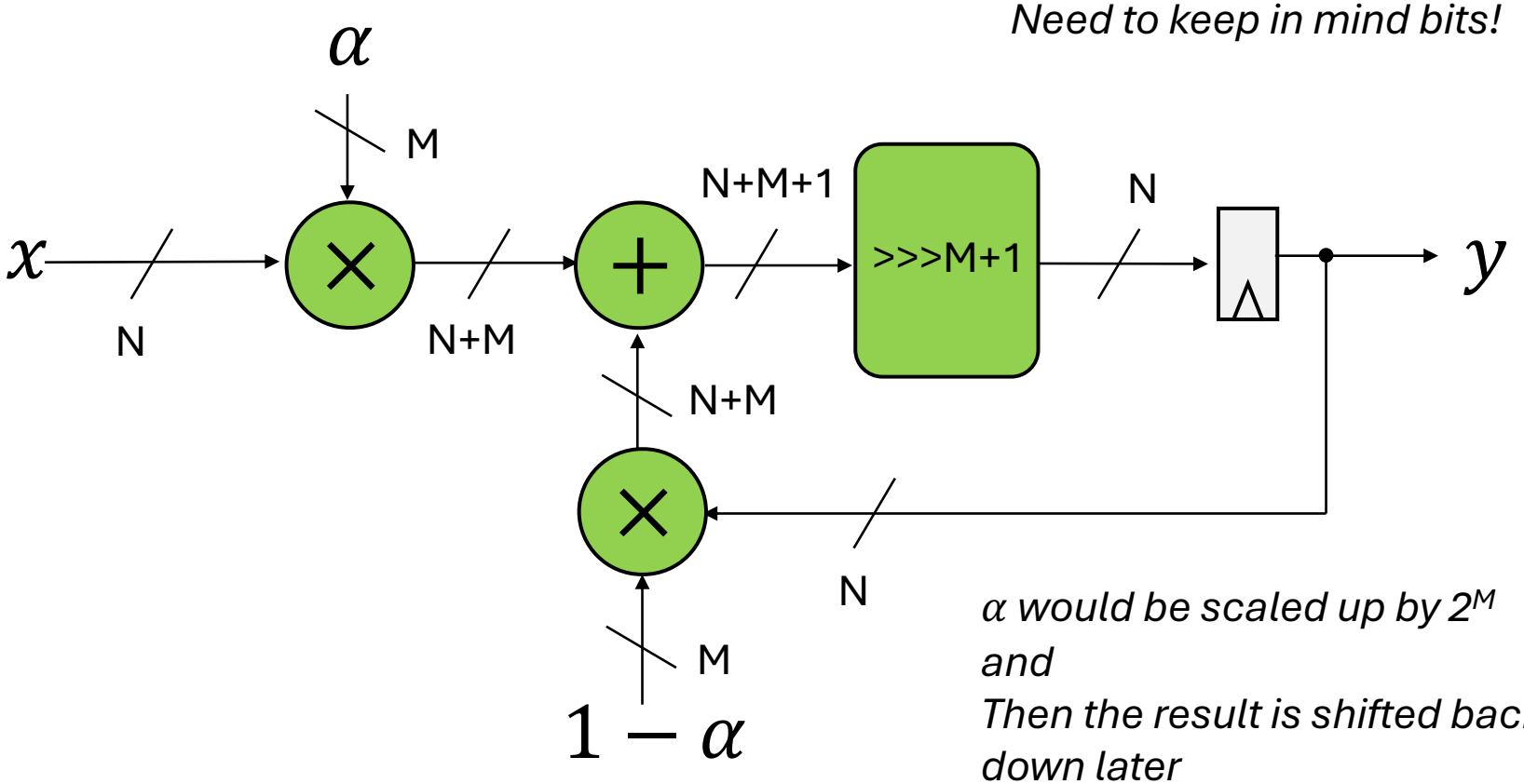
$$y[n] = \alpha \cdot y[n-1] + (1-\alpha) \cdot x[n] \qquad 0 \leq \alpha \leq 1$$



Infinite Impulse Response (Modified)

$$y[n] = \alpha \cdot y[n-1] + (1-\alpha) \cdot x[n] \quad 0 \leq \alpha \leq 1$$

Need to keep in mind bits!



α would be scaled up by 2^M and
Then the result is shifted back down later

IIR

- Computationally lightweight
- No very flexible, often poor performance since not a lot of parameters to adjust.

Finite Impulse Response

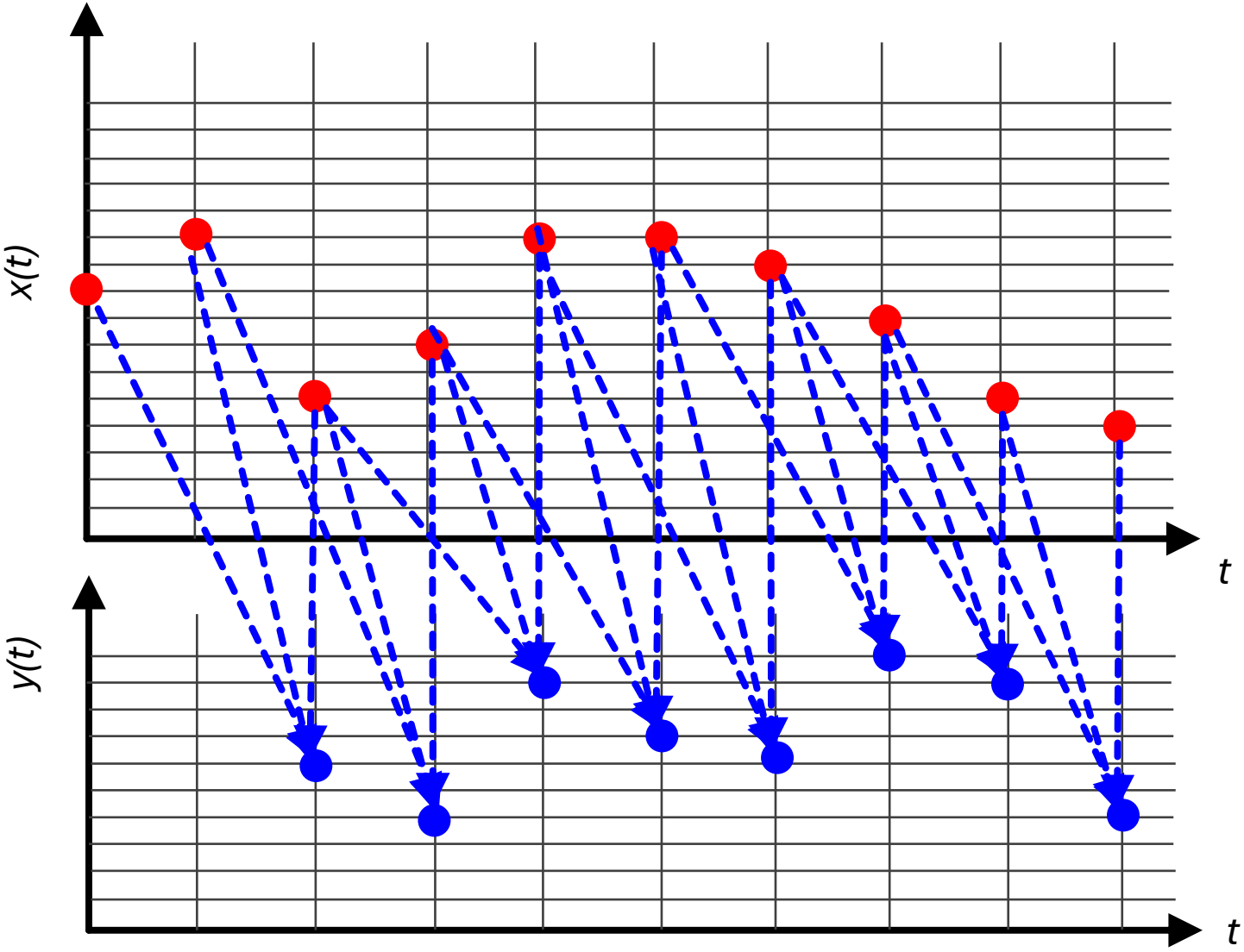
- Have the output be based off of a sliding window of the past history of the input.
- Literally just convolution basically

$$y[n] = b_0 \cdot x[n] + b_1 \cdot x[n-1] + b_2 \cdot x[n-2]$$

- Very powerful!! Huge flexibility in choosing those coefficients and can get a ton of behaviors!

FIR Filter

$$y[n] = b_0 \cdot x[n] + b_1 \cdot x[n-1] + b_2 \cdot x[n-2]$$



FIR Filters

- Extremely flexible
- Often times **many, many** "taps" long (N in 100s is not uncommon)

$$y[n] = \sum_{k=0}^{N-1} b_k \cdot x[n-k]$$

- The values you pick for these taps are arrived at using a number of DSP-oriented algorithms (beyond scope of course…but in 6.341, etc)

FIR Filters

$$y[n] = \sum_{k=0}^{N-1} b_k \cdot x[n-k]$$

- Some online tools, Matlab, Python, Vivado all have tools that allow you to:
  - specify how you want your filter to look
  - Provide you the coefficients needed to generate that filter
- The b coefficients are generally provided as real numbers between 0 and 1. But since we don't want to do floating point arithmetic, we usually scale them by some power of two and then round to integers.
  - Since coefficients are scaled by 2^M, we'll have to re-scale the answer by dividing by 2^M. But this is easy – just get rid of the bottom M bits!
- More taps generally means you can get better response:
  - Closer to ideal filter!

FIR Filters

- They implement convolution, so can be much more than just “filters”
- You can use them to:
  - Remove complicated features to signals
  - Add complicated features to signals
  - Making an FIR filter “dynamic” can lead to systems that dynamically tune themselves.
  - Make a ”matched filter” to look for features.
- Very much a work-horse type module.

FIR Filters Use A Lot of Math

- Each sample of a FIR involves the same amount of multiply-accumulates as there are taps
- This means you can end up needing to do 100's of heavy math operations per sample
- And you also need access to all those old samples to make it work.

FIR Filter (Iterative Implementation)

$$y[n] = \sum_{k=0}^{N-1} b_k \cdot x[n-k]$$

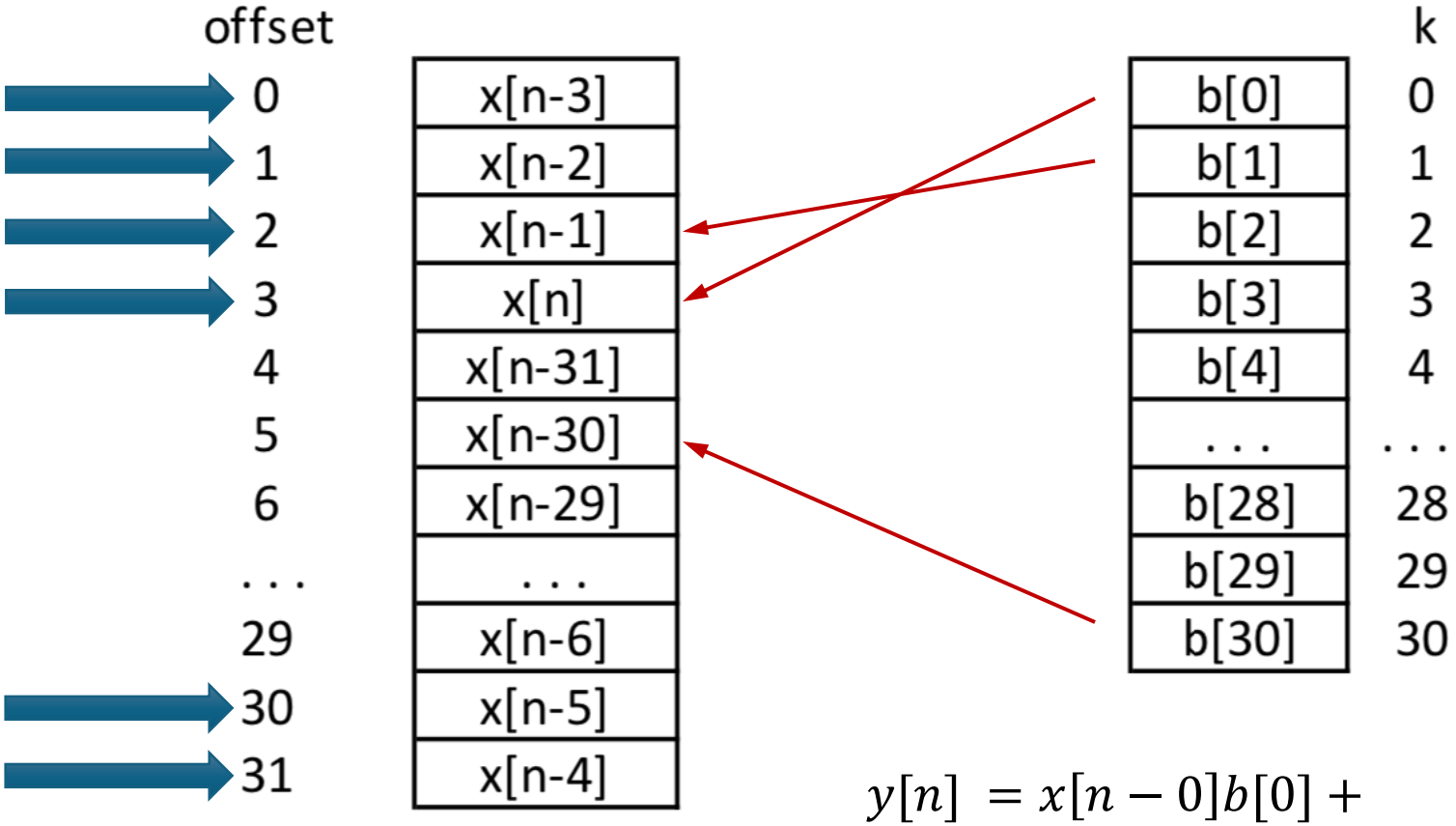
- For audio and mid-frequency phenomena, usually plenty of clock cycles exist between each signal sample (you have 2000 clock cycles of 100 MHz between each audio sample of 48 ksps audio for example!)

- Just make a low-resource state-machine-based module.
- After every sample, do each multiply-accumulate for each tap. As long as you have enough cycles, you can do thousands of taps. Can even break up into more

Memory Requirements

- FIR filters may require large histories of signals (thousands of samples back)
- Ideally you'd hold in a dense format (like BRAM) but that only allows 1 or 2 reads per cycle.
- Might be fine for low data rates.

Circular Buffer/Pointer



$$y[n] = \sum_{k=0}^{30} x[n-k]b[k]$$

$$y[n] = x[n-0]b[0] + x[n-1]b[1] + \ldots x[n-30]b[30] +$$

Higher Speed FIRs get nasty though

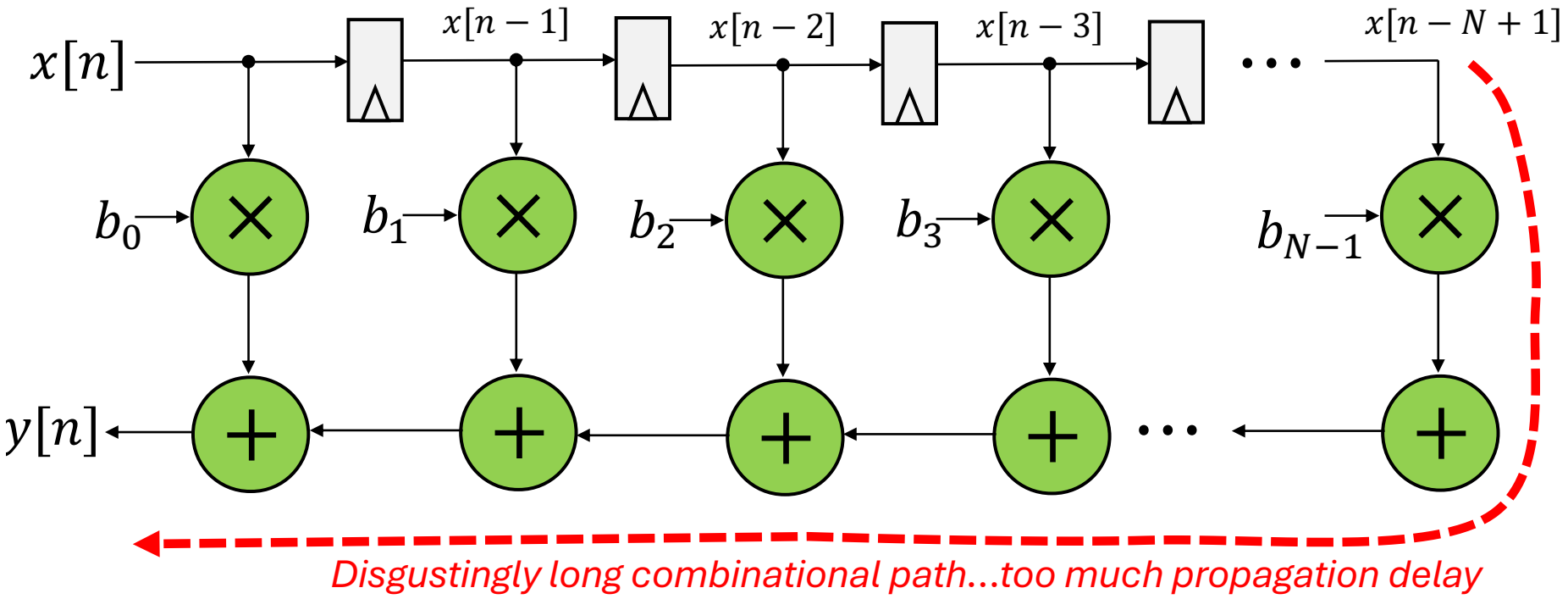
- If your data stream is now operating closer to your clock rate, for FIRs of any reasonable size, you won't have enough clock cycles to get and do everything serially.
- Pipelining is the solution here.

How Much Data is That?

- If you're handling a 200 MHz data stream, and running it through a 30 tap FIR filter….
- That means you need to be doing 6 billion multiply-accumulates per second
- This is where FPGAs, hardware systems really start to shine

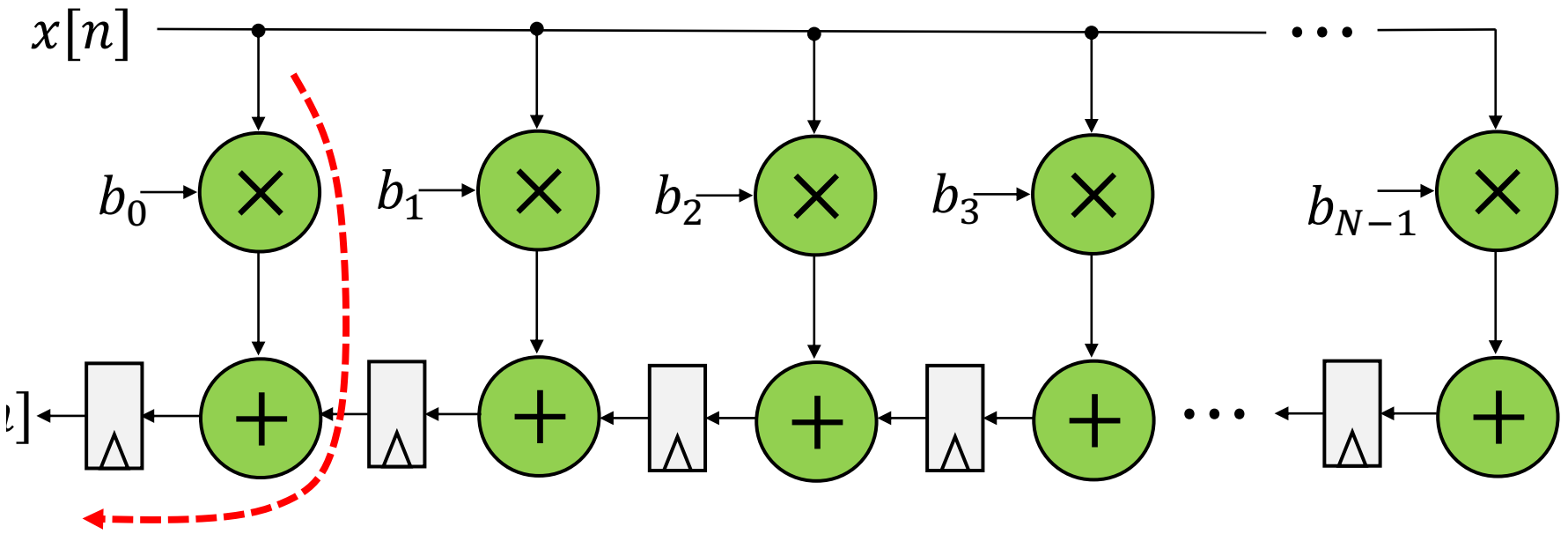
Finite Impulse Response Implementation

$$y[n] = \sum_{k=0}^{N-1} b_k \cdot x[n-k]$$



Finite Impulse Response (Modified)

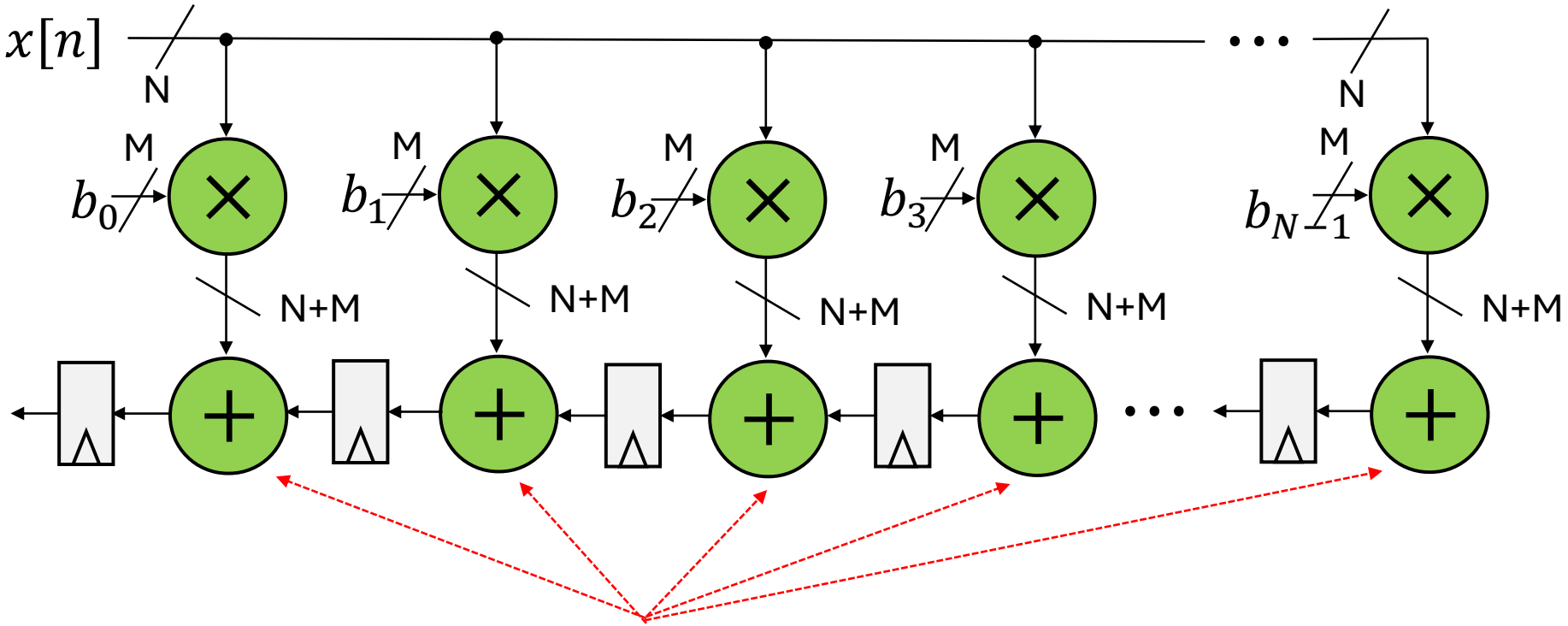
$$y[n] = \sum_{k=0}^{N-1} b_k \cdot x[n-k]$$



Much nicer critical path (worst propagation delay)

Bit Growth

$$y[n] = \sum_{k=0}^{N-1} b_k \cdot x[n-k]$$



Adding values that are N+M bits repeatedly grows the number of bits needed to not lose precision…will grow at between 1 bit per N and 1 bit per $\log_2(N)$! But this can grow large so there's ways to handle it

https://zipcpu.com/dsp/2017/07/21/bit-growth.html

Most FIR Filters (not all) are symmetric too.

- Depending on situation can double-up and feed back delayed signal

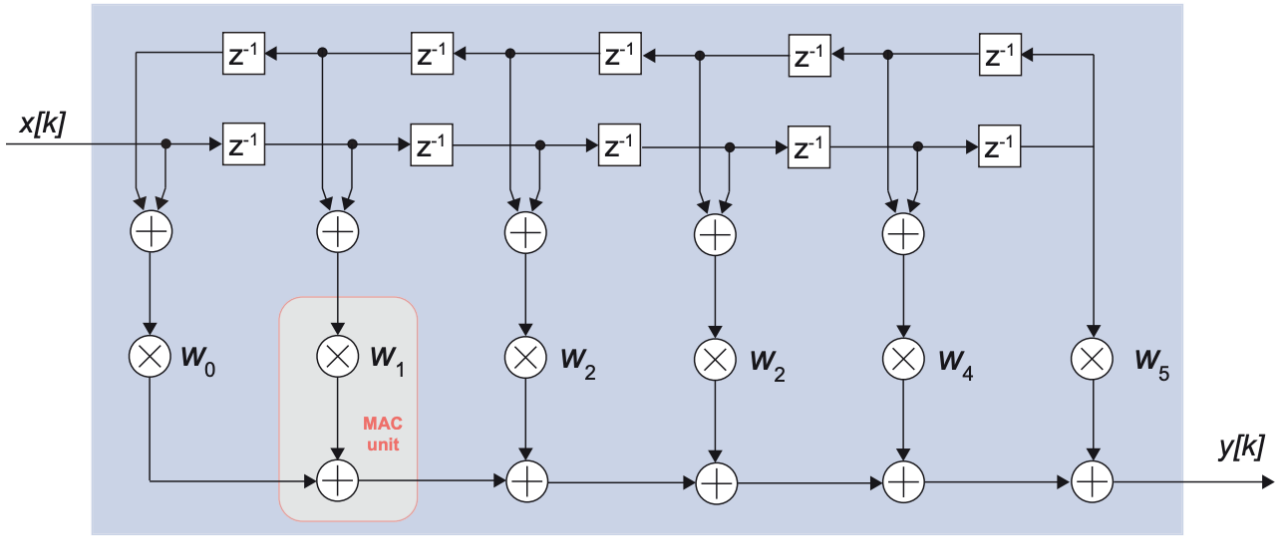


Figure 4.23: Example of a symmetric 11-weight FIR filter.

DSP Blocks

- If we return to the DSP blocks we spoke about earlier…
- It is like it was made for this (it was):

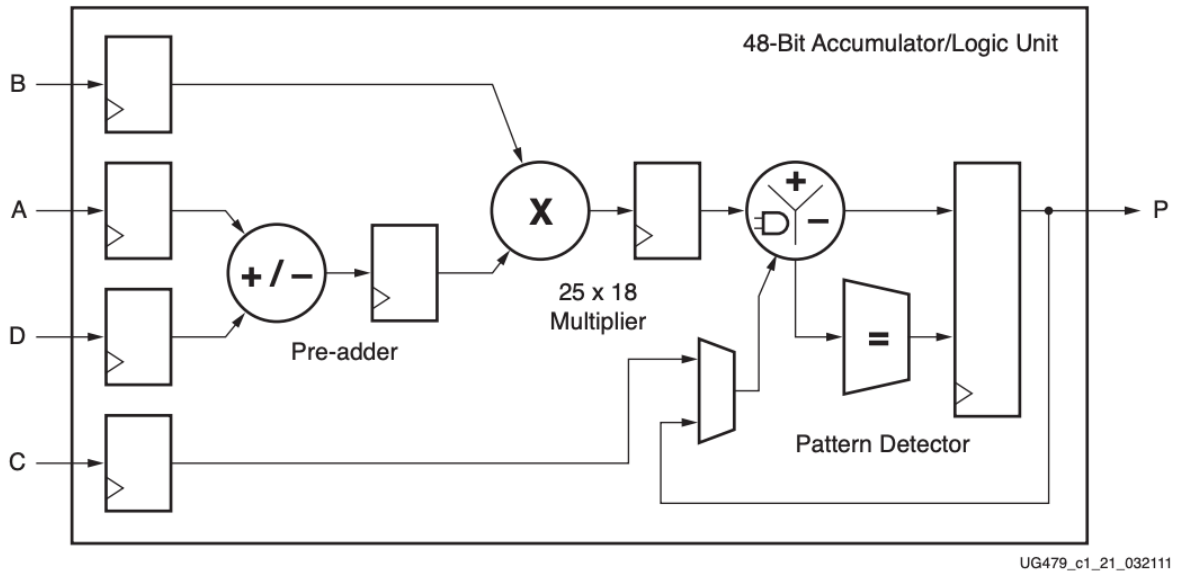
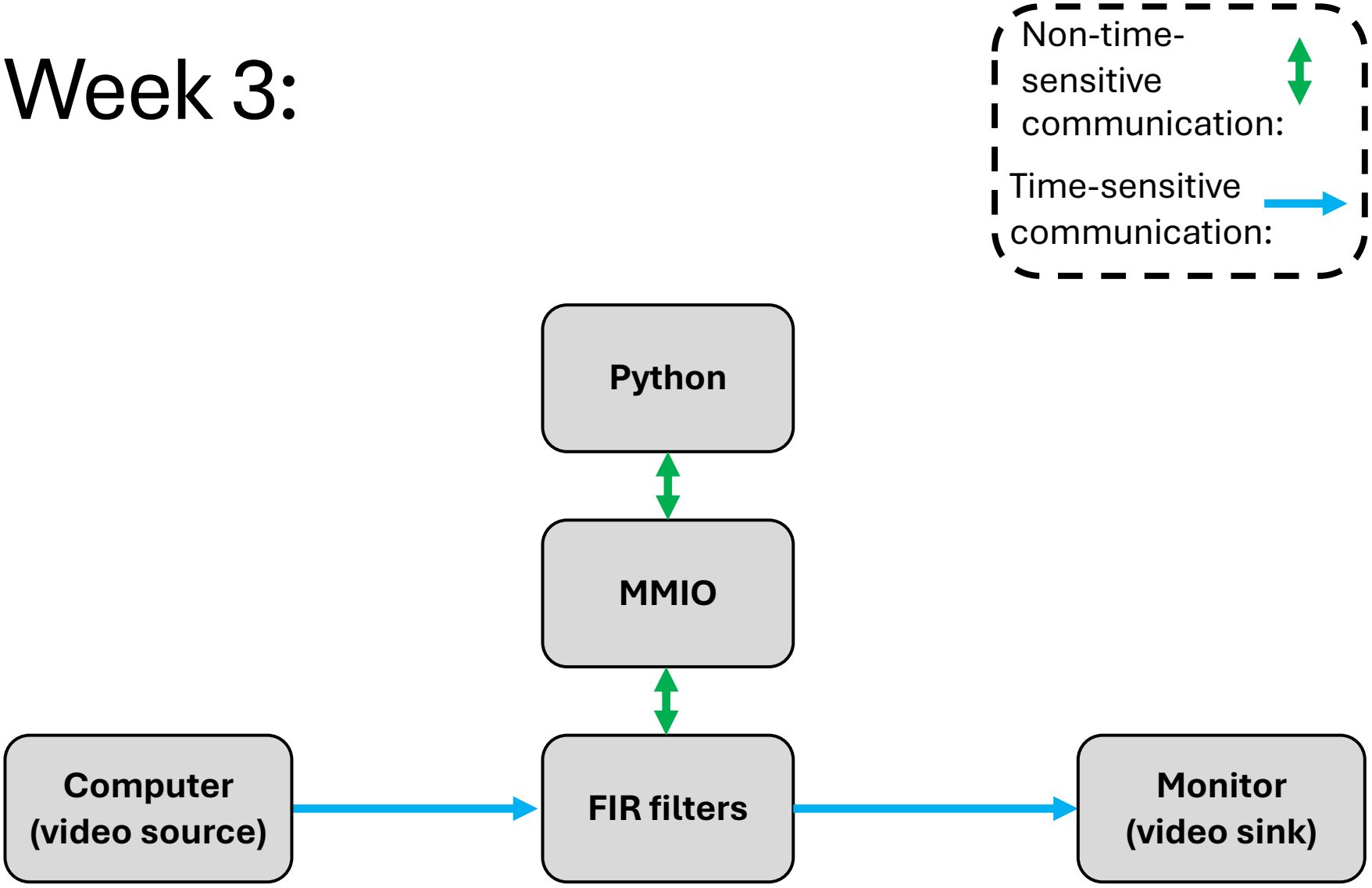


Figure 1-1: **Basic DSP48E1 Slice Functionality**

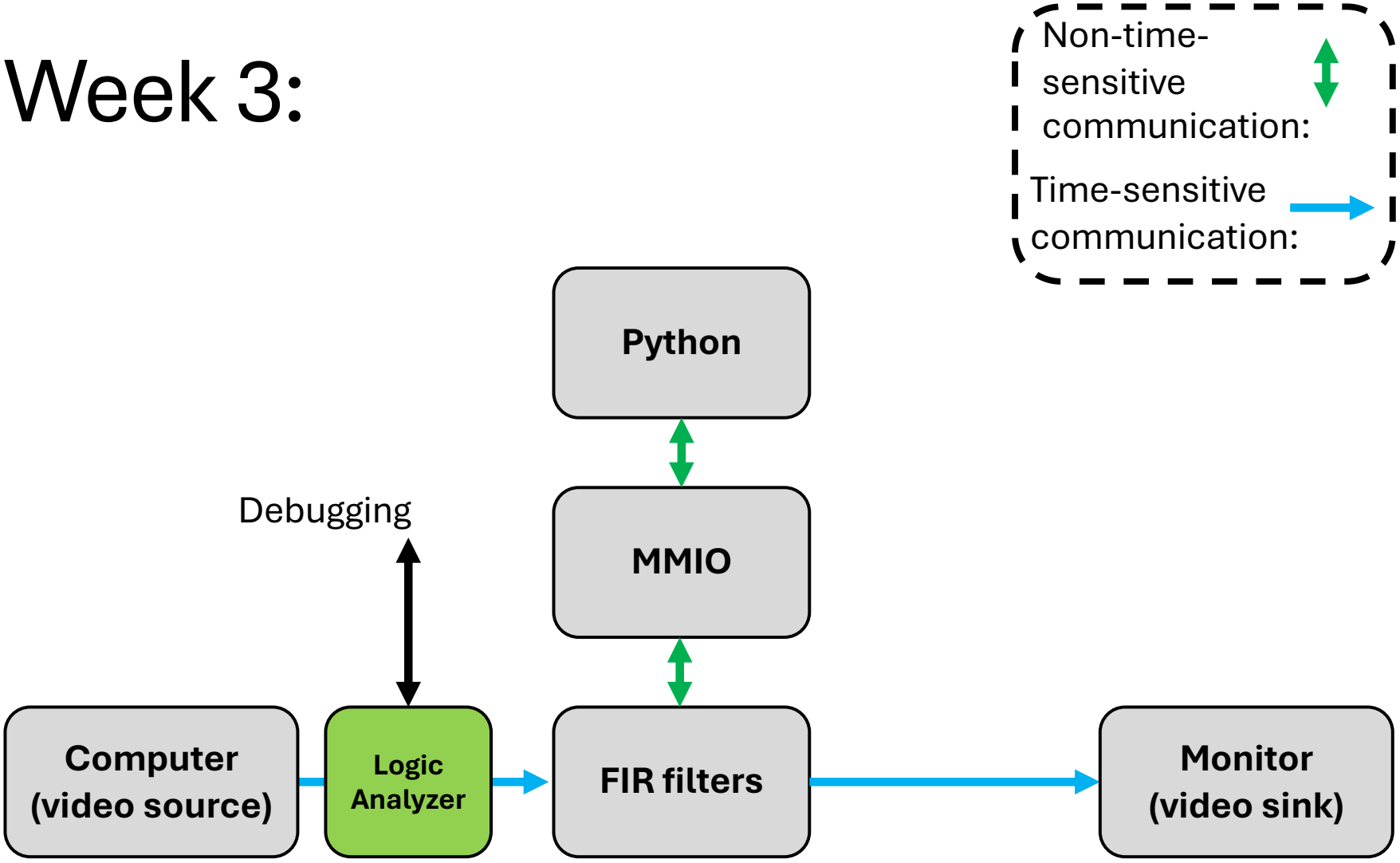
Week 3's Assignment

- Design (SV) and verify (cocotb and numpy) a simple 15-tap FIR filter
- Drop it into a video pipeline on the Pynq board
- Control its tap values using an MMIO interface
- Use ILA to see bits and eyes to see results

Week 3:


Non-time-sensitive communication:
Time-sensitive communication:
Python
MMIO
Computer (video source)
FIR filters
Monitor (video sink)

Week 3:


Non-time-sensitive communication:
Time-sensitive communication:
Python
MMIO
Debugging
Computer (video source)
Logic Analyzer
FIR filters
Monitor (video sink)

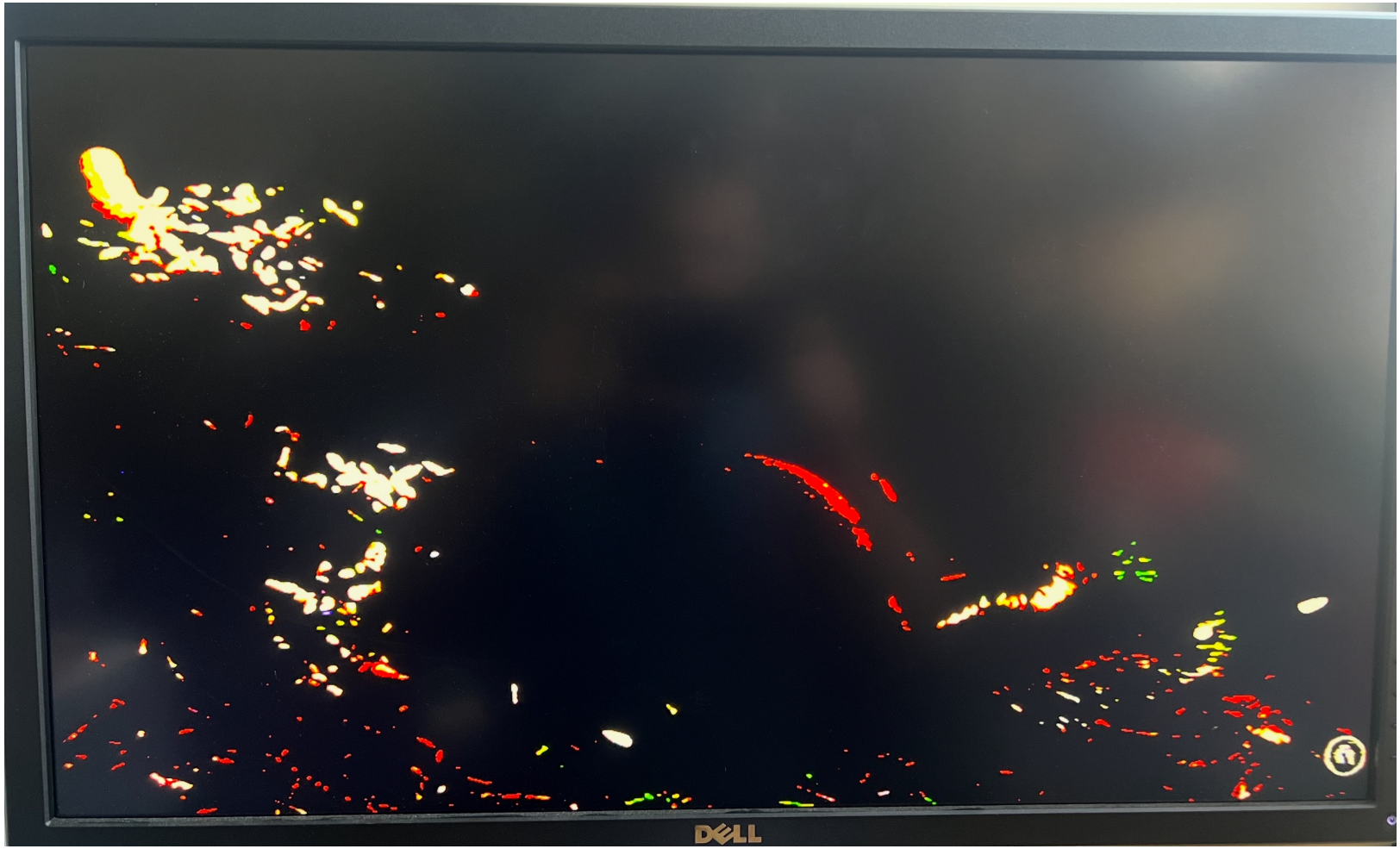
Math is pretty impressive

- We'll be running 15-tap FIR filters on on all three color channels at 720p video rate

- That works out to be 3.3 billion multiply-and-accumulates per second controlled completely from python

Original Video

Low-Pass Filter



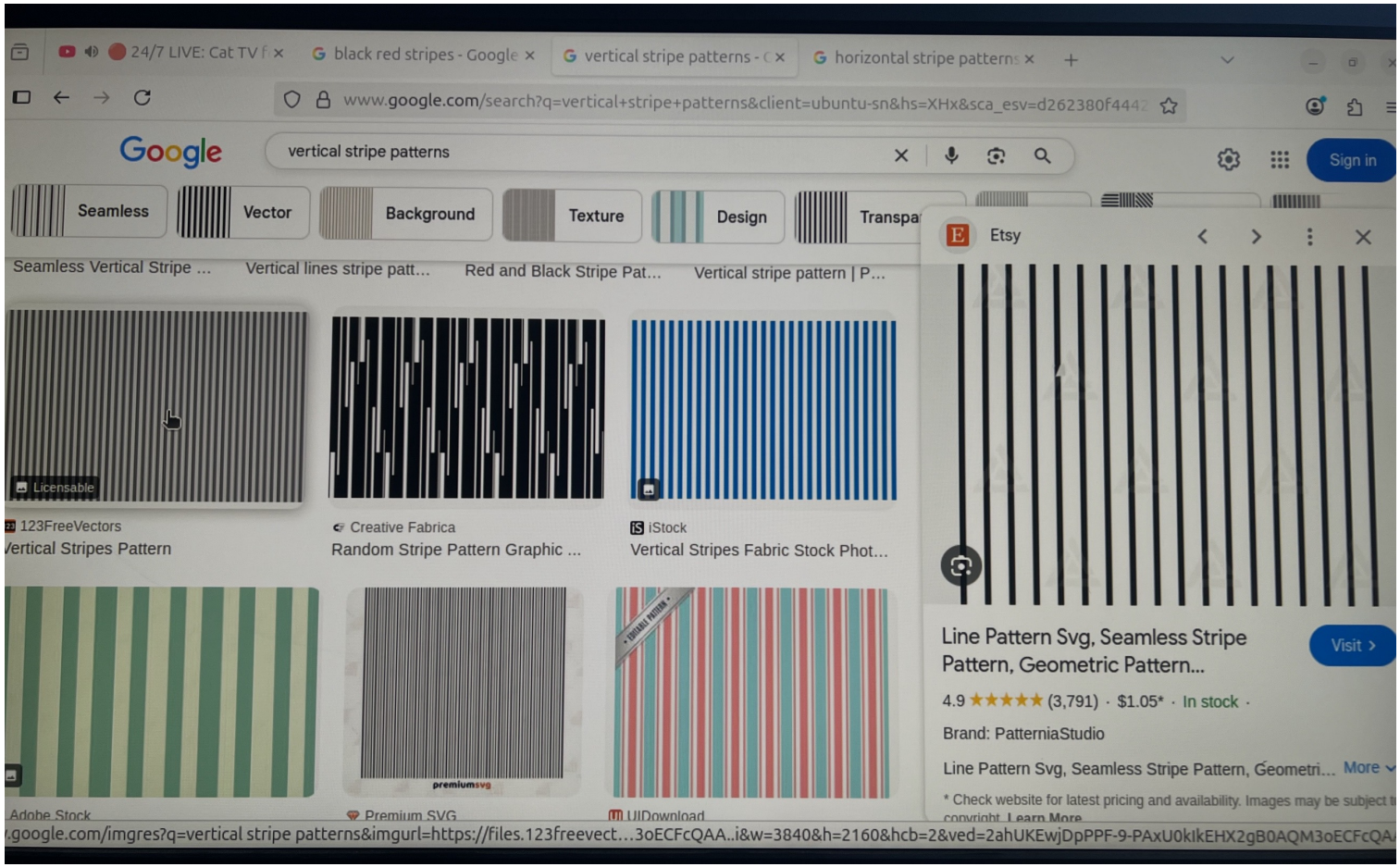
High-Pass Filter

Few Observations

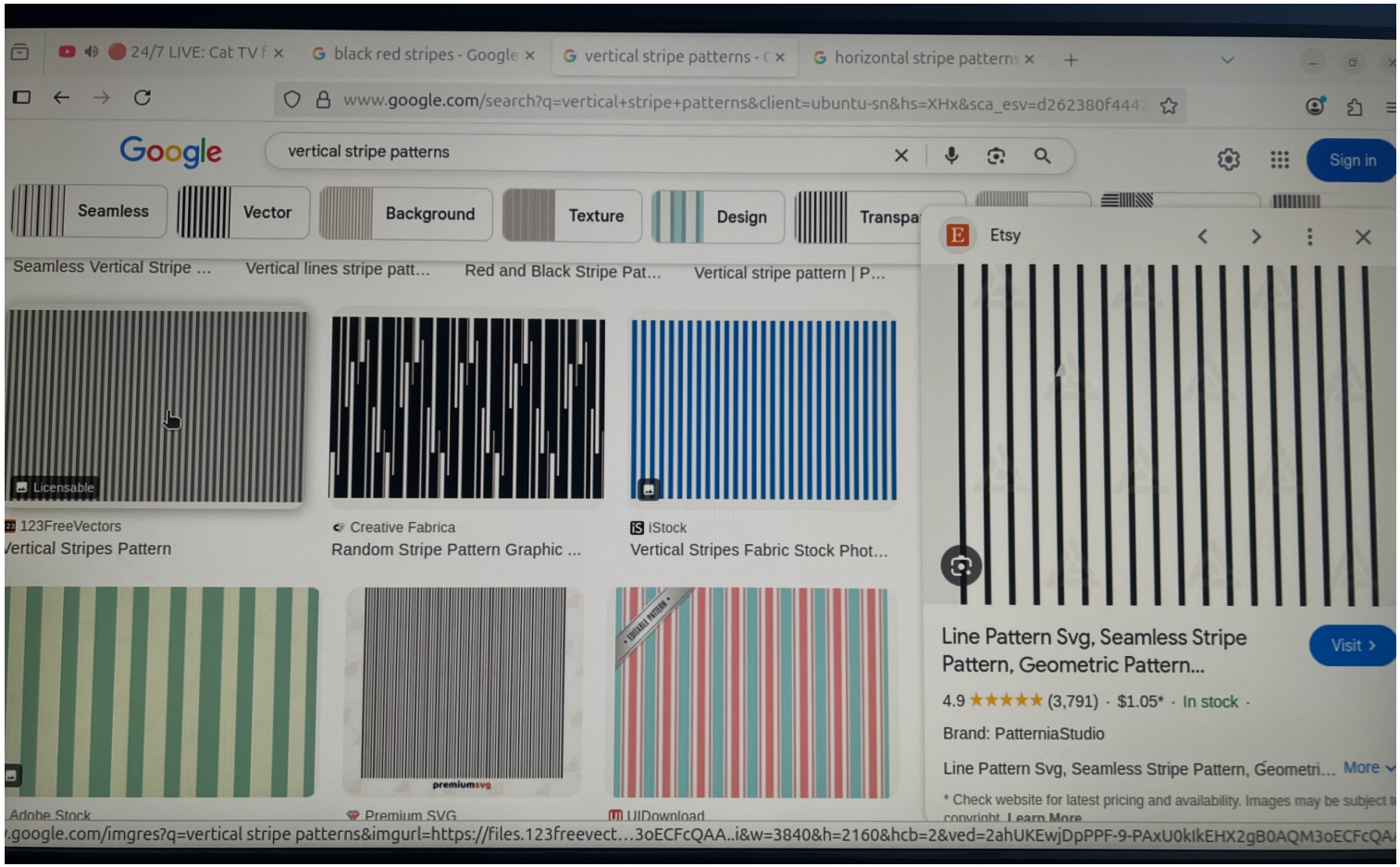
- The filter that we're applying is 1D meaning it is only applied in the horizontal direction as the pixels scan across the page.

- This is very rarely done in image processing…usually do FIR filters in *two dimensions* in which case we call it a kernel

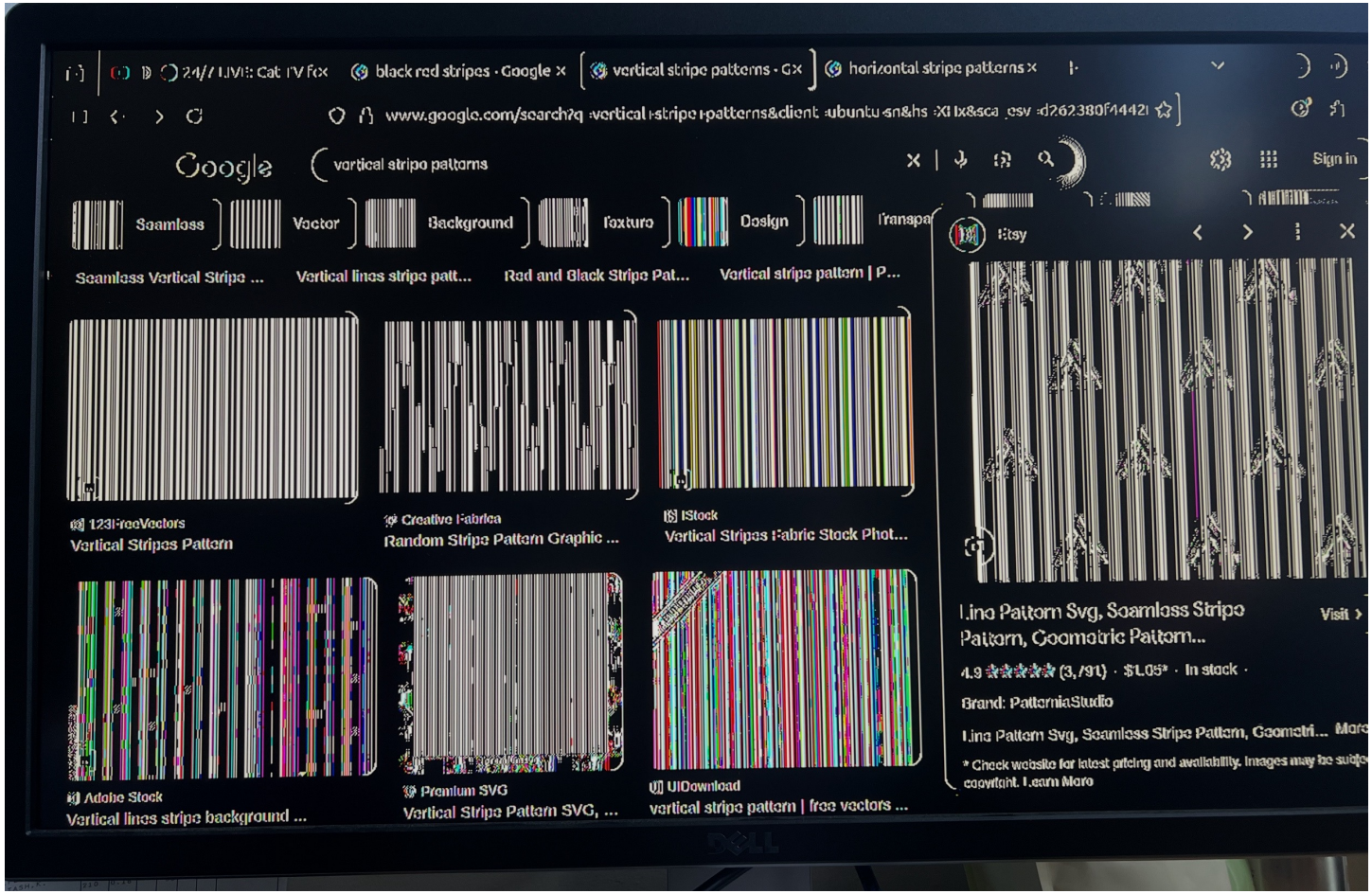
Starting Images



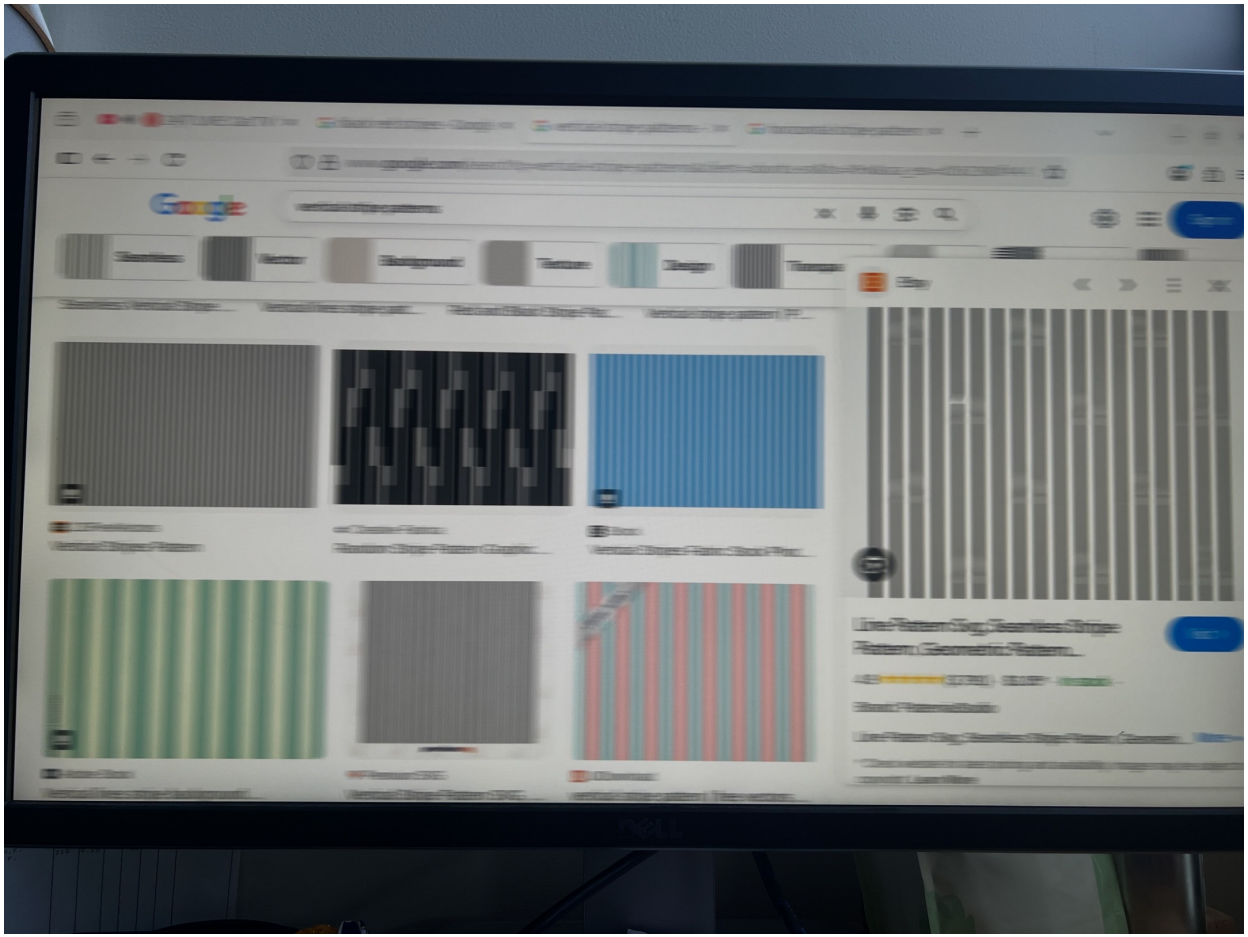
```
coeffs = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1]
```



```
coeffs = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,1]
```

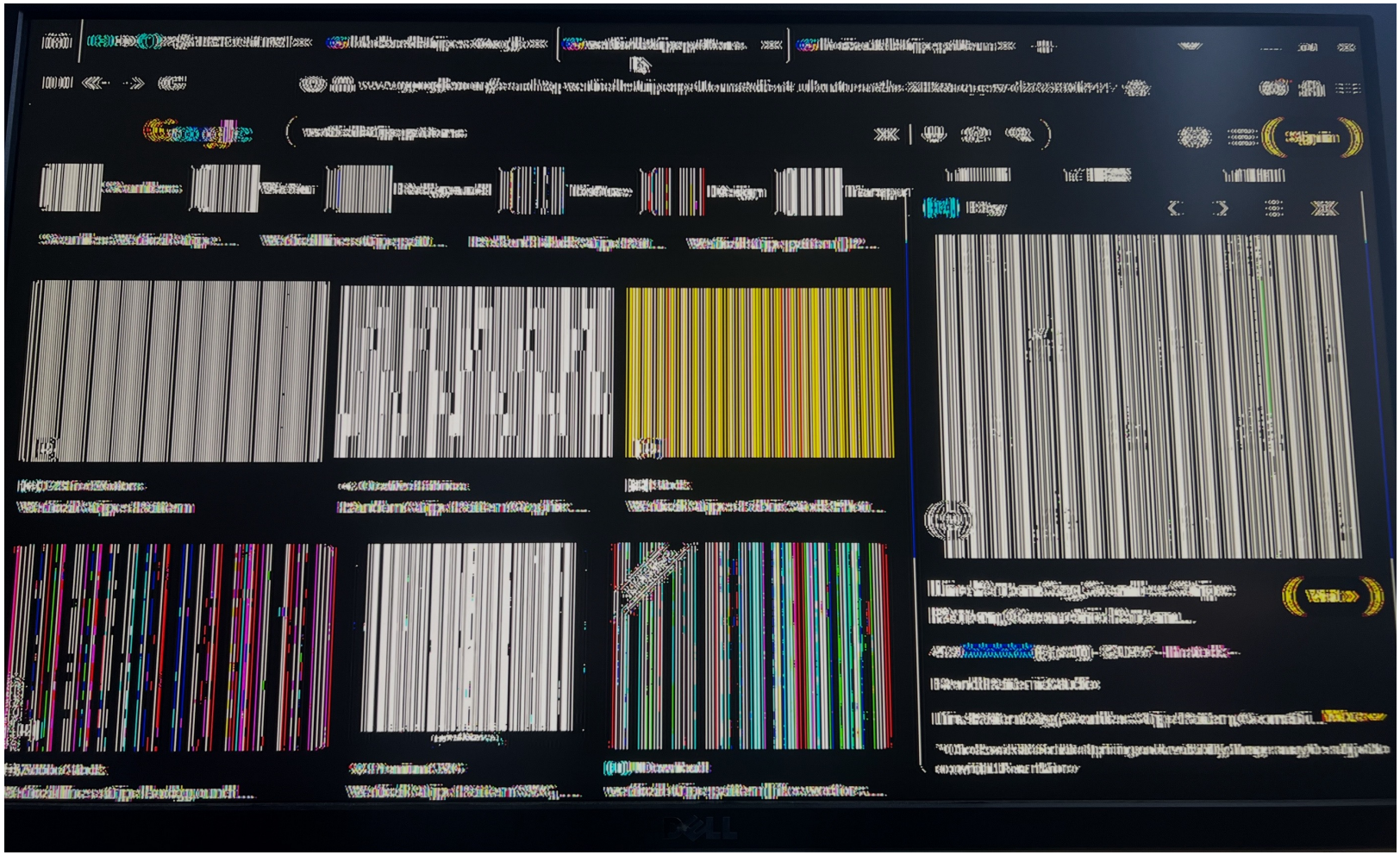


```
coeffs = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

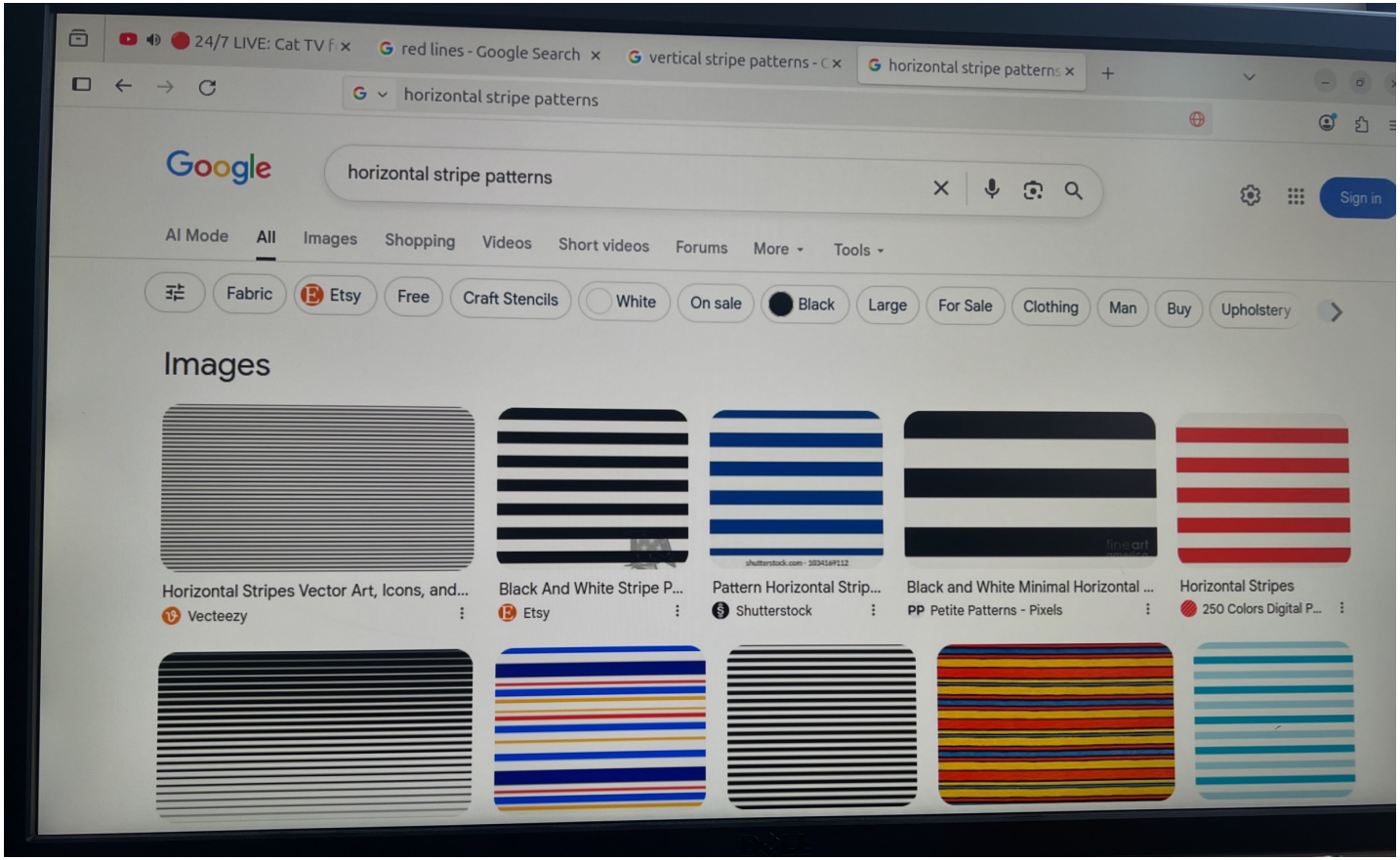


```
coeffs = [3,5,-16,9,12,-5,-41,69,-41,-5,12,9,-16,5,3]
```

(high pass filter)



Vertical patterns aren’t seen!



```
coeffs = [3,5,-16,9,12,-5,-41,69,-41,-5,12,9,-16,5,3]
```

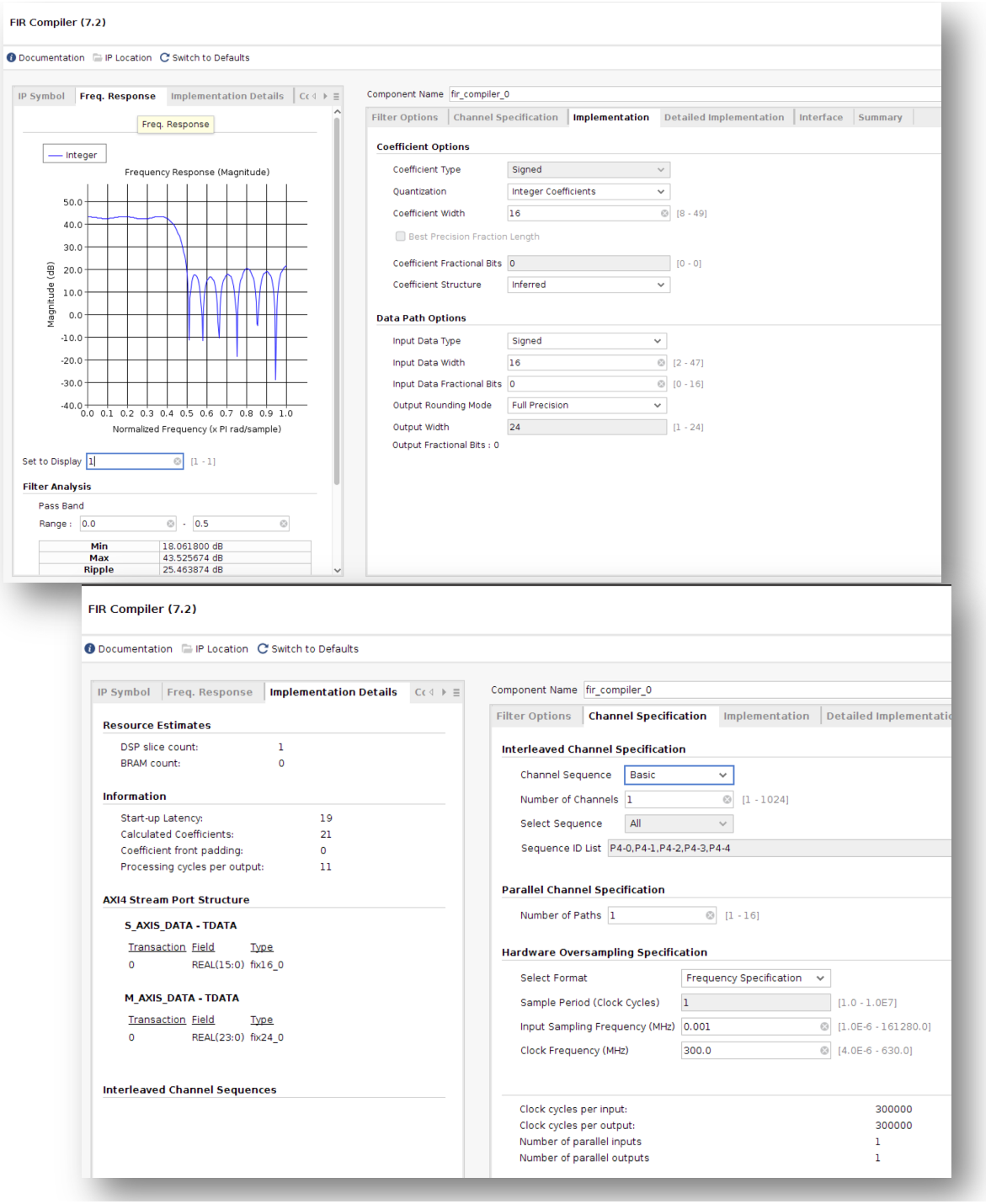
(high pass filter)

Where do the coefficients come from?

- Lots of tools to design for them
- Scipy has some
- Matlab has some

FIR Wizard

- FIRs are so common, Vivado actually has some IP infrastructure to aid in designing them
- Can tune how pipelined vs. Iterative/FSM you want your FIR!
- Or use Python/numpy to determine coefficients



http://t-filter.engineerjs.com