Signed Values in Verilog and Analog/Signal Processing Things

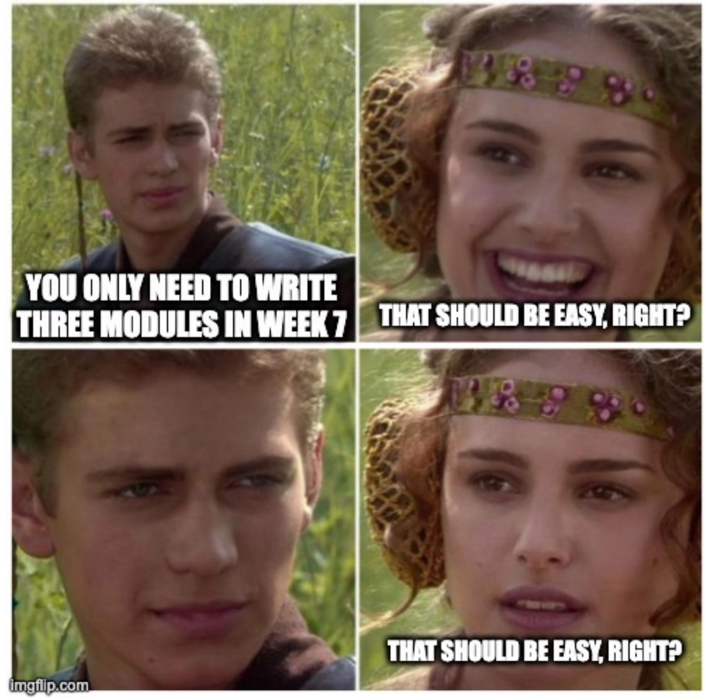
6.111 Fall 2024

Administrative

- Week 06 due yesterday
- Week 07 Out today (due *next Thursday*…give you an extra day*)*.
- Last lab. We're not doing 8.
- After abstracts are due tomorrow @5pm, staff will meet to figure out who works with who and email you
- We're going to push the due date of the block diagram report to the Tuesday 29th

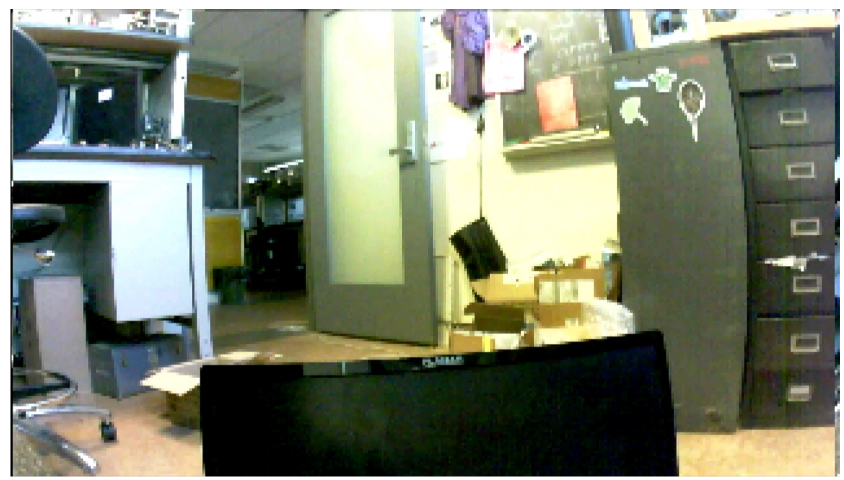
Week 7: Convolution

- Only have to write three modules
- Please start early




```
Phase 25 Post Router Timing
INFO: [Route 35-20] Post Routing Timing Summary | WNS=-0.059 | TNS=-0.319 | WHS=0.050  | THS=0.000  |
```

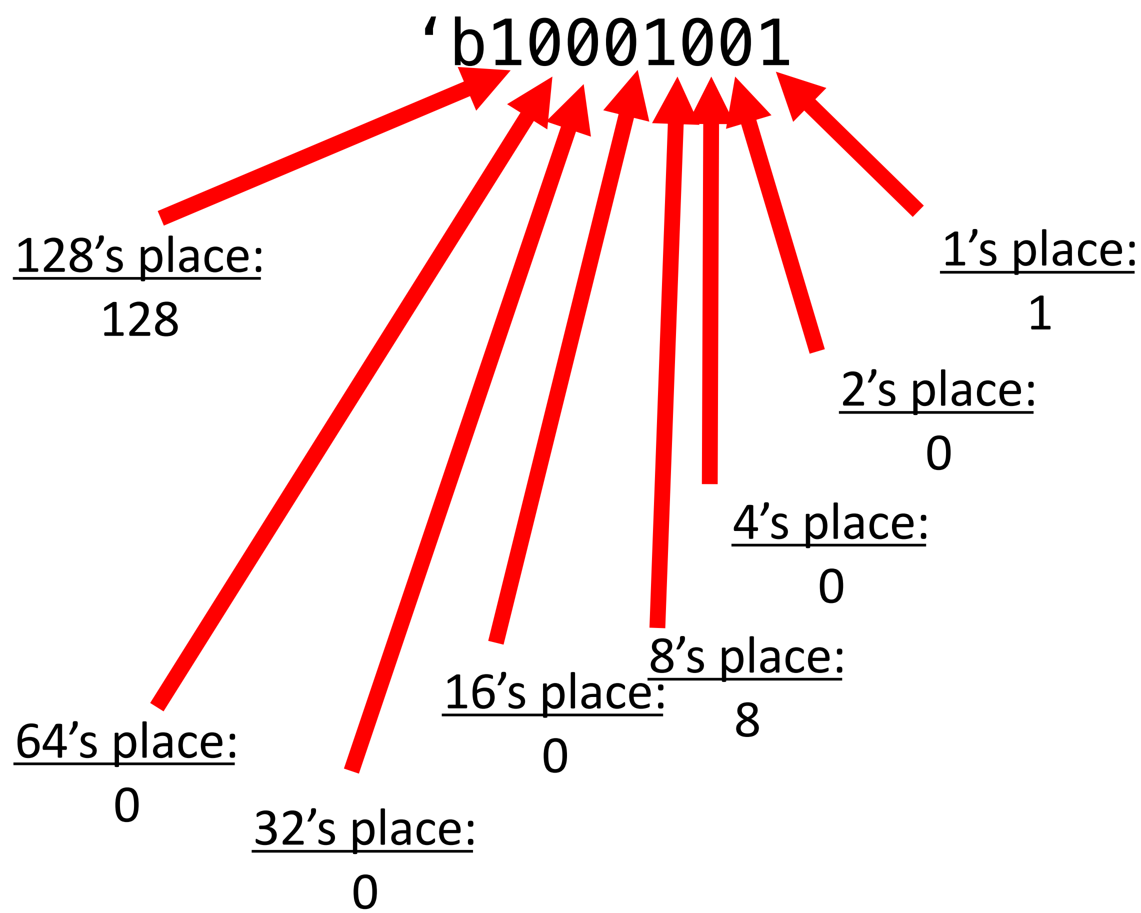
Light travels 1.7 mm in the time that my design initially failed by



Signed Numbers

How to Represent Numbers

- Simplest approach is to just read the binary number in regular base 2 (just like in our friend base 10!)



128+8+1 = **137**

Most arithmetic works out well too!

- Add/Subtract:

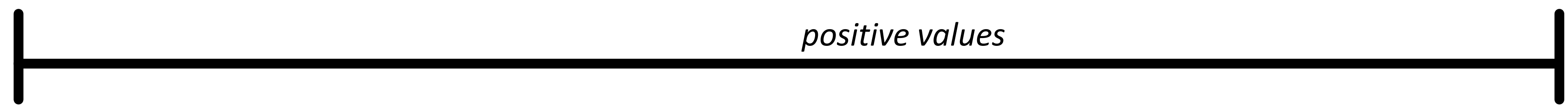
```
    b10001001      (137)
  + b00000101        (5)
  --------------------------
    b10001110      (142)
```

- Multiply/Divide:

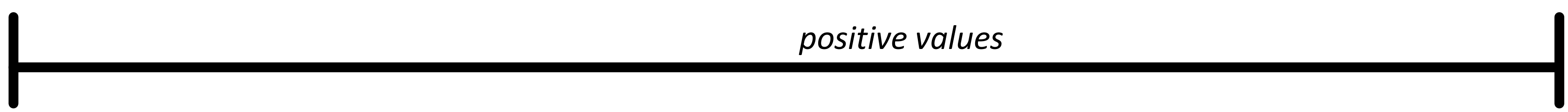
```
     ‘b00000101        (5)
   * ‘b00000110        (6)
  --------------------------
    ‘b00000000        (0)
   ‘b000001010       (10)
  +’b000010100       (20)
  --------------------------
   ‘b000011110       (30)
```

Unsigned Values:

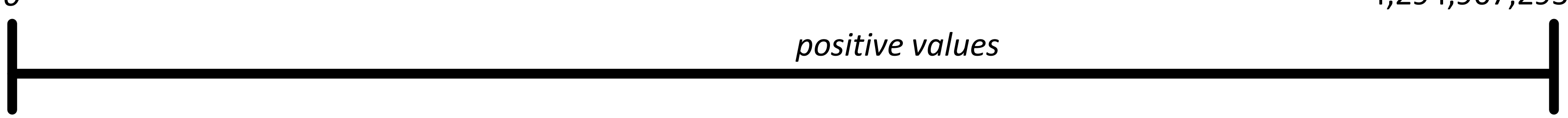
- 1 byte (8 bits): 2^8 values: 256 numbers to rep
  - Express from 0 to 255



- 2 bytes (16 bits): 2^{16} values: 65,536 numbers
  - Express from 0 to 65,535

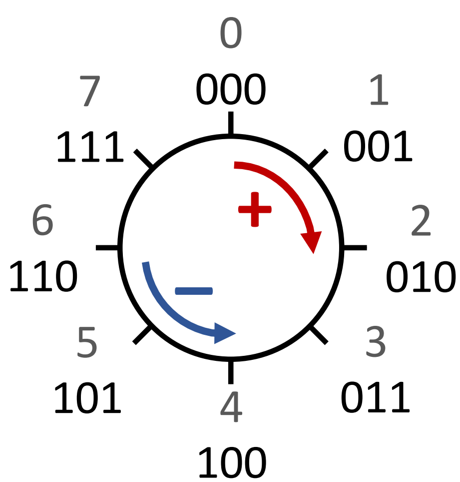


- 4 bytes (32 bits): 2^{32} values: 4,294,967,296 nums
  - Express from 0 to 4,294,967,295

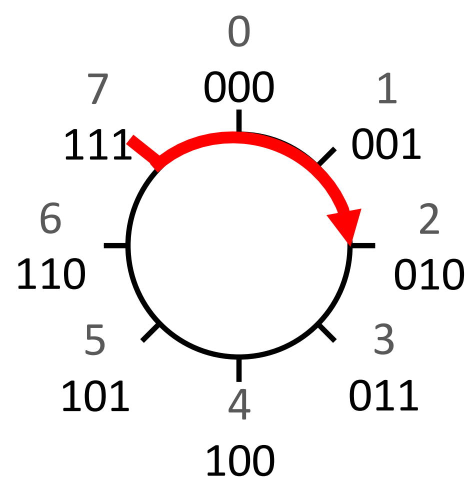


Inherent Modularity

- If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition)
  - This is known as an overflow
- Common approach: Ignore the extra bit
  - Gives rise to modular arithmetic: With N-bit numbers, equivalent to following all operations with mod 2^N
  - Visually, numbers “wrap around”:



Example: (7 + 3) mod 2^3 ?

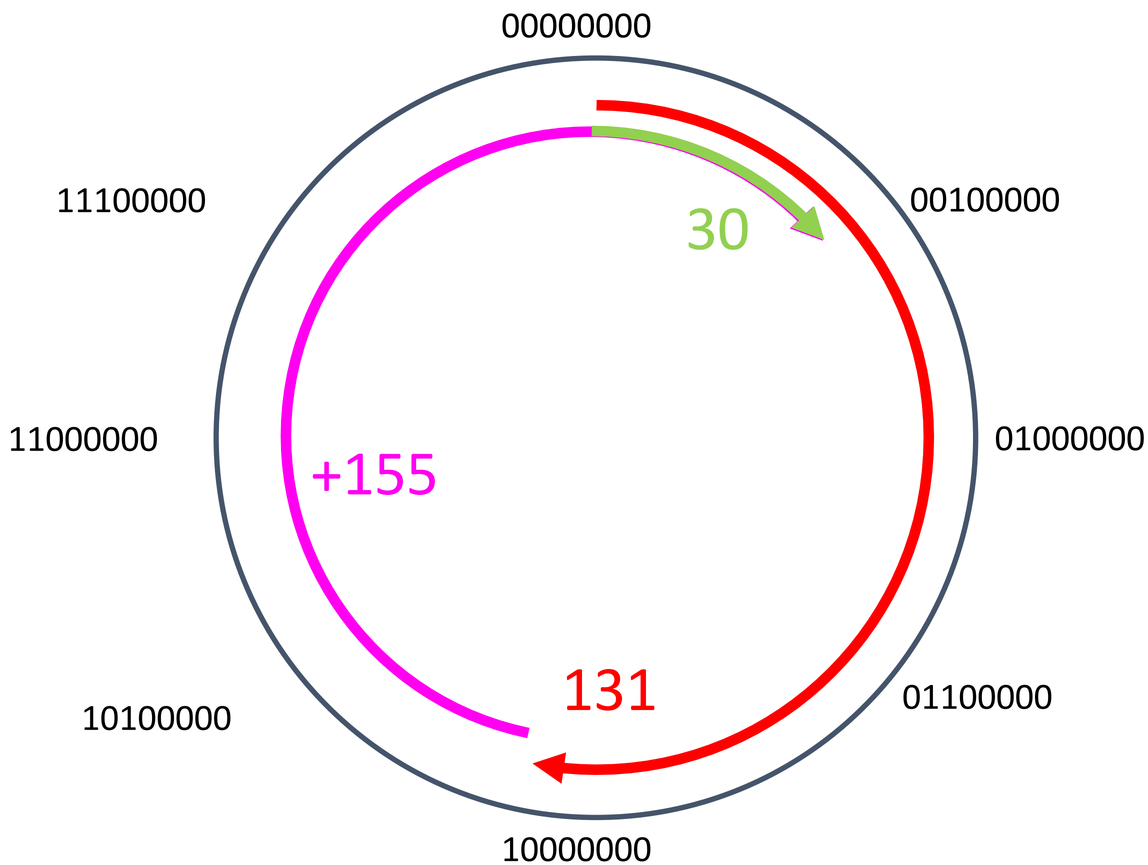


Happens with more bits too (8 bits)

- What happens if you add 131 to 155 with 8 bit?

```
   10000011    (131)
+  10011011    (155)
  ___________
  100011110    (286)
   00011110    (30)
```

overflow



What About Negatives?

- Our Number Schemes so far only allow representation of positive numbers (and zero).
- What about negatives? How can we do this in an efficient manner?

One Solution: "Sign Bit" (did this with Pong)

- If most-significant-bit (msb) is 0, interpret like a negative sign:
  - If 0, lower bits are from a positive number
  - If 1, lower bits are from a negative number
- To get the negative of the number, flip the msb:

```
'b00010001 == +(16+1) == 17

'b10010001 == −(16+1) == −17

'b00000000 == 0
'b10000000 == −0
```

- Major problem(s)?

Another Solution: “One’s Complement”

- If most-significant-bit (msb) is 0, interpret like an unsigned value.
- If msb is 1, then number is negative, else positive.
- To get the negative of the number flip all the bits:

$$-A = \sim A$$

bitflip

```
‘b00010001 == +(16+1) == 17

‘b11101110 == bitflip of 17== -17

‘b00000000 == 0
‘b11111111 == -0
```

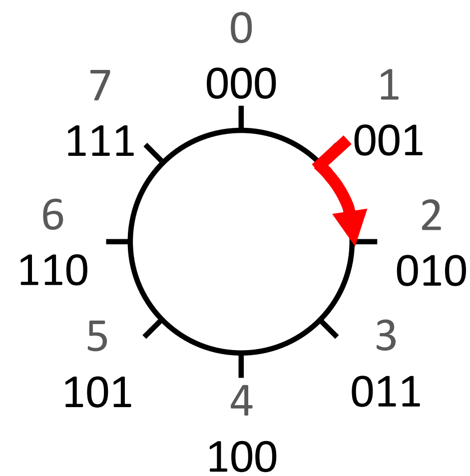
- Major problem(s)?

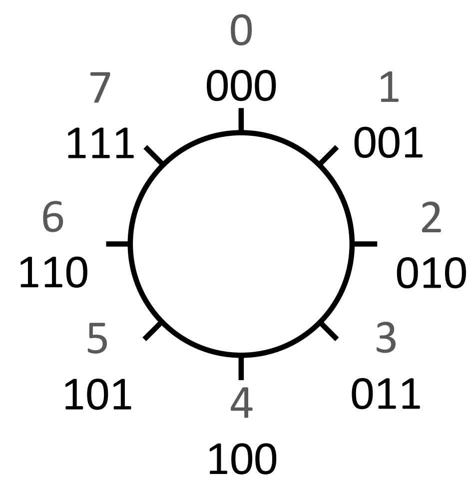
Inherent Modularity to the Rescue

- Return to our 3-bit* number system:

- If I want to add 1, I just add 1 and move clockwise by 1 unit

- If I want to subtract 1, is there a number I could add using our same regular adding rules to get the same result? If so, that number could be called "-1", right?

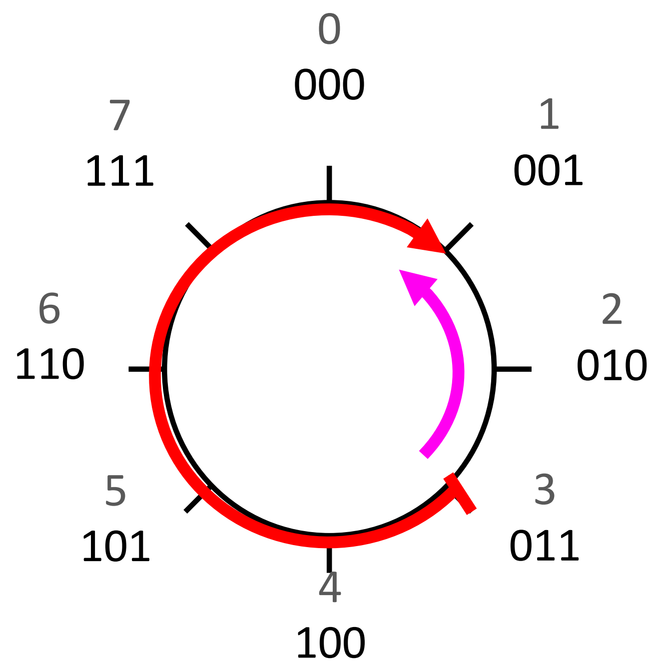




*3 bits here since easy to think about and draw, but could do with any number of bits

A Negative Number

- If I start at ”3” aka ‘b011, what could I add to get to 1?
- To go back 2, I can add:
  - $2^3 - 2$ = 6
- (3+6)%8 = 1.
- Or: “-010” = 110



Negating a Number

- In a 3bit space, The negative of a number can be expressed as:

$$\text{“}-A\text{”} = 8 - A$$

- Or written a different way:

$$\text{“}-A\text{”} = 1 + \text{`b111} - A$$

- ‘b111 *minus* any 3 bit value will be the same as the bitflip of that value (~A)

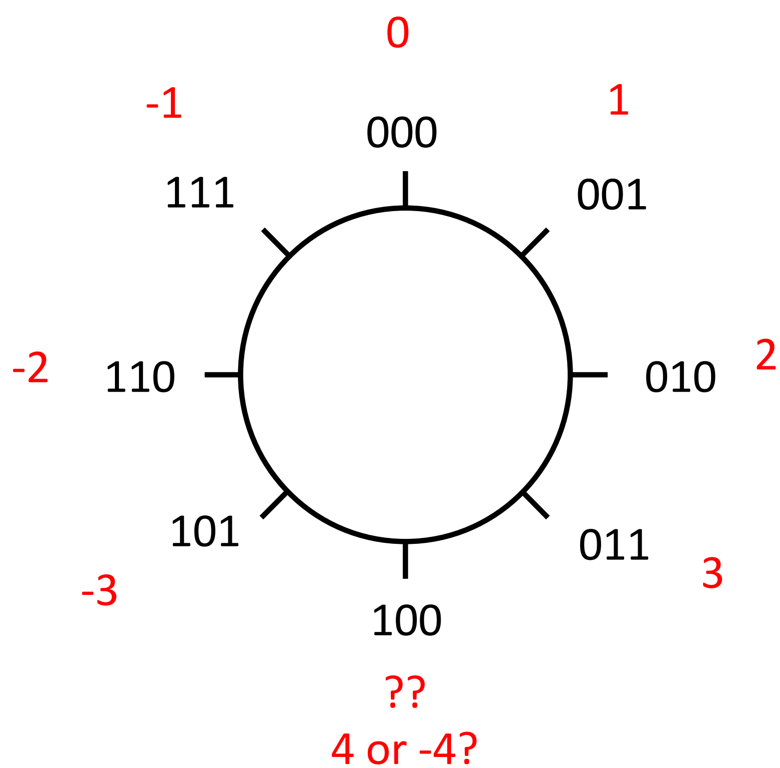
$$\text{“}-A\text{”} = 1 + (\text{`b111} - A)$$

- So the negative of any value must be:

$$\boxed{-A = 1 + {\sim}A}$$

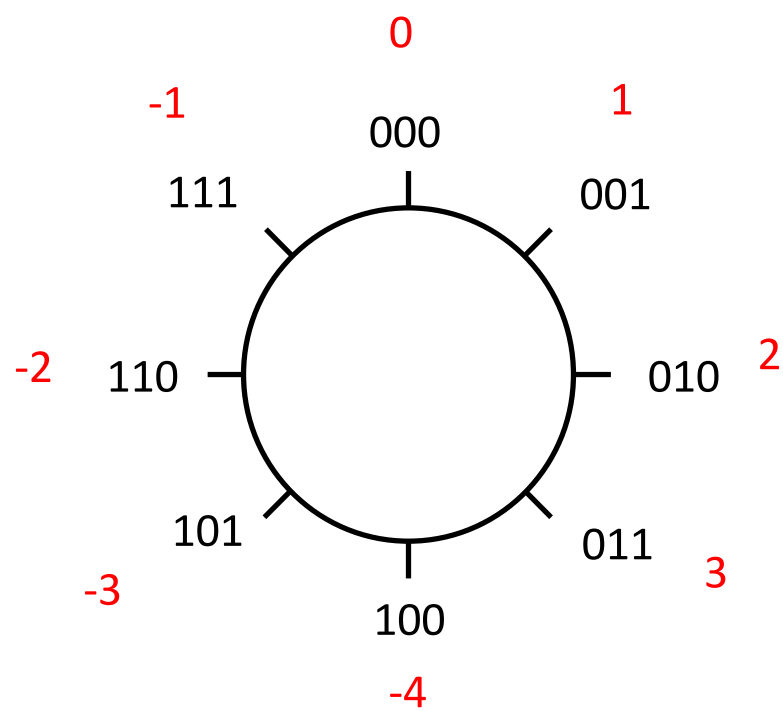
The Solution: 2's Complement

- For 000 to 111 what numbers do we get in this scheme?



Interesting…

- If we make 100 into -4, the system of numbers becomes consistent and easily extensible to more bits.



- With this model we can come up with some rules/observations…

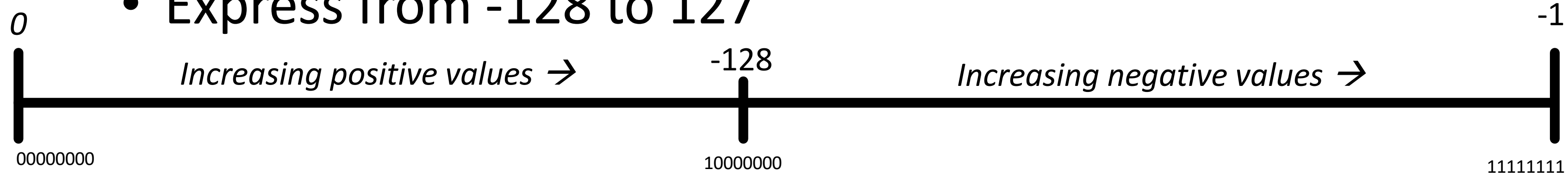
Two's Complement (Signed) Ints

- For an n bit signed int, we represent from:
  - Min: -2^{n-1}
  - Max: $2^{n-1} - 1$
  - Zero is always *all* zeros
- The negative of a number A is always $-A = 1 + {\sim}A$
- A number is positive if the msb is 0:
  - If so, just add up non-zero digits by weight as you do for unsigned
- A number is negative if msb is 1:
  - If so add weight of msb, then for all bits below that subtract off the weight of any non-zero digits:

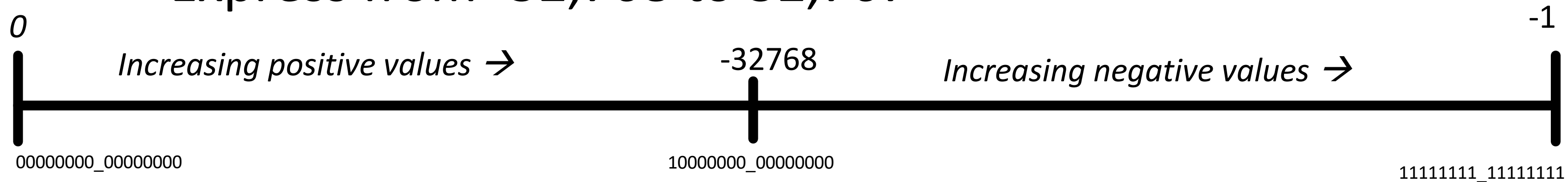
*msb = most significant bit

Signed Values:

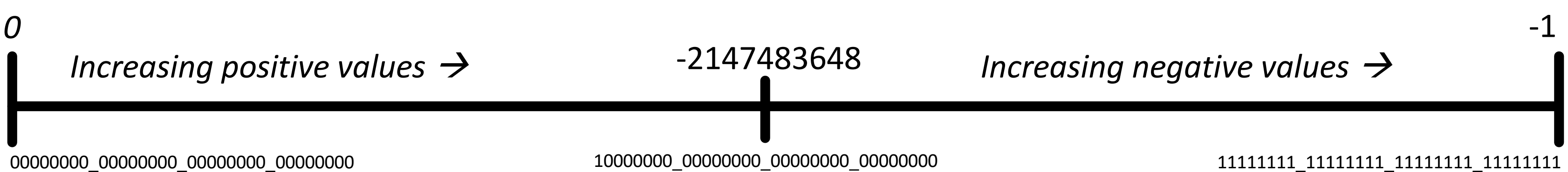
- 1 byte (8 bits): 2^8 values: 256 numbers to rep
  - Express from -128 to 127



- 2 bytes (16 bits): 2^{16} values: 65,536 numbers
  - Express from -32,768 to 32,767



- 4 bytes (32 bits): 2^{32} values: 4,294,967,296 nums
  - Express from -2,147,483,648 to 2,147,483,647



Math Operations Still Work

- Two's Complement is pretty nice because you can still do all your regular math operations pretty easily
- Also No double-zero!
- Pretty much all modern digital systems use two's complement math to represent signed integers

Signed Arithmetic in Verilog

Just add “signed” modifier to your variable declaration. \s

```
logic [15:0] a; // Unsigned
logic signed [16:0] signed_a; //signed
```

Using Signed Arithmetic in Verilog

ALL OF THE FOLLOWING ARE TREATED AS ***UNSIGNED*** IN VERILOG!!!

- *Any* operation on two operands, unless **both operands are signed**
- Based numbers (e.g. 12'd10), unless the explicit "s" modifier is used)
- Bit-select results `a[5]`
- Part-select results `a[4:2]`
- Concatenations

```
logic [15:0] a; // Unsigned
logic signed [15:0] b;
logic signed [16:0] signed_a;
logic signed [31:0] a_mult_b;

assign signed_a = a;//Convert to signed
assign a_mult_b = signed_a * b
```

Example of multiplying signed by unsigned

http://billauer.co.il/blog/2012/10/signed-arithmetics-verilog/

For example, consider these two test bench examples:

```
module test_one;
  logic signed [3:0] x;
  logic [3:0] y;
  logic signed [8:0] z;
  initial begin
    x = -2;
    y=3;
    z = x*y;
    $display(x, y, z);
    $finish;
  end
endmodule
```

Result:

-2 3 42

```
module test_two;
  logic signed [3:0] x;
  logic signed [3:0] y;
  logic signed [8:0] z;
  initial begin
    x = -2;
    y=3;
    z = x*y;
    $display(x, y, z);
    $finish;
  end
endmodule
```

Result:

-2 3 -6

Not really synthesizable here ($finish, $display, etc)…but shows what Verilog is thinking

Sign extension

Consider the 8-bit 2’s complement representation of:

```
42 = 00101010        -5 = ~00000101 + 1
                        =  11111010 + 1
                        =  11111011
```

What is their 16-bit 2’s complement representation?

```
42 = 0000000000101010

-5 = 1111111111111011
```

Extend the MSB (aka the “sign bit”) into the higher-order bit positions

Using Signed Arithmetic in Verilog

Shifts in Verilog do not base themselves off of the type they are working on. >> is always binary shift.

“**<<<**“ and “**>>>**” tokens result in arithmetic (signed) left and right shifts: multiple by 2 and divide by 2.

Right shifts will maintain the sign by filling in with sign bit values during shift

```
logic signed [3:0] x;
logic signed [3:0] value = 4'b1000; // -8
x = value >> 2 // results in 0010 or 2
x = value >>> 2 // results in 1110 or -2

logic [3:0] value = 4'b1000; // -8
x = value >> 2 // results in 0010 or 2
x = value >>> 2 // results in 0010 or -2 (is unsigned…extends with 0’s)
```

Few Other Things

- When specifying numbers/constants you cand put a **s** in front to specify it as signed.

```
logic signed [7:0] x;
initial begin
  x = -'d5;
  $display("%d %8b", x,x); //prints: -5 11111011
  x = -'sd5;
  $display("%d %8b", x,x); //prints: -5 11111011
  x = 'd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'sd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'd234;
  $display("%d %8b", x,x); //prints: -22 11101010
  x = 'sd128;
  $display("%d %8b", x,x); //prints: -128 10000000
  #100;
  $finish;
end
```

```
logic [7:0] x;
initial begin
  x = -'d5;
  $display("%d %8b", x,x); //prints: 251 11111011
  x = -'sd5;
  $display("%d %8b", x,x); //prints: 251 11111011
  x = 'd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'sd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'd234;
  $display("%d %8b", x,x); //prints: 234 11101010
  x = 'sd128;
  $display("%d %8b", x,x); //prints: 128 10000000
  #100;
  $finish;
end
```

Need to make a thing signed?

- Either use $signed
- Or declared signed types to route through:

```
logic signed [3:0] x = 4'b1110; // -2 also -4'
logic [3:0] y = 4'b1100; //12 unsigned, (-4 signed)
logic signed [4:0] z
assign z = x*$signed(y);//interpret y as signed
//results in z having 5'b11000 in it (-8)
//OR:
logic signed [3:0] y_signed;
assign y_signed = y;
assign z = x*y_signed; //multiplication of two signed things is signed
//results in z having 5'b11000 in it (-8)
```

Signed Numbers Guideline

- Once you start using signed Verilog in a module or a signal path, just make everything you're using is signed. If you do that, you should be ok.
- Make sure everything upstream of a calculation has been done in only a signed environment (held in signed logics and used with signed logics.
- Signed/Unsigned bugs are some of the hardest to find so be cautious
- When in doubt also use $signed

A variable being signed does NOT change the bits the variable contains!

- The signedness of a variable only impacts how operators are interpreted. It does not impact the bits themselves.
- Some operators are relatively robust and act kinda the same regardless if you are signed or unsigned! (+, -, bitwise operators, even *)
- But the setup and interpretation of these operations often needs slightly different framing based on the signedness

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	logical equality	equality
!=	logical inequality	equality
===	case equality	equality
!==	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
^~ or ~^	bit-wise XNOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Few Other Things

- When specifying numbers/constants you cand put a **s** in front to specify it as signed.

```
logic signed [7:0] x;
initial begin
  x = -'d5;
  $display("%d %8b", x,x); //prints: -5 11111011
  x = -'sd5;
  $display("%d %8b", x,x); //prints: -5 11111011
  x = 'd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'sd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'd234;
  $display("%d %8b", x,x); //prints: -22 11101010
  x = 'sd128;
  $display("%d %8b", x,x); //prints: -128 10000000
  #100;
  $finish;
end
```

```
logic [7:0] x;
initial begin
  x = -'d5;
  $display("%d %8b", x,x); //prints: 251 11111011
  x = -'sd5;
  $display("%d %8b", x,x); //prints: 251 11111011
  x = 'd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'sd5;
  $display("%d %8b", x,x); //prints: 5 00000101
  x = 'd234;
  $display("%d %8b", x,x); //prints: 234 11101010
  x = 'sd128;
  $display("%d %8b", x,x); //prints: 128 10000000
  #100;
  $finish;
end
```

- In all comparative cases above we’ve put identical bits into variable. When we ask Verilog to perform an operation with those bits, its interpretation differs.
- This can bleed into sign extension and other peripheral tasks, for example…

Consider Multiplication

- Consider two variables. One has ‘b101 in it another has ‘b110 in it.
- If you invoke unsigned multiplication on these bits…stuff just sort of works:

```
    ‘b101        (5)
  * ‘b110        (6)
  ----------------------
    ‘b000        (0)
   ‘b1010       (10)
 +’b10100       (20)
  ----------------------
 ‘b011110       (30)
```

- In actuality because the multiplication of a 3 bit by 3 bit number could result in 6 bits of result, you should “extend” but it can be just 0’s

Consider Multiplication

- So for unsigned you're really doing this:

```
     ‘b000101        (5)
*  ‘b000110        (6)
---------------------------
     ‘b000000        (0)
    ‘b0001010       (10)
+  ’b00010100       (20)
---------------------------
     ‘b011110       (30)
```

Consider Multiplication

- Consider two variables. One has ‘b101 in it another has ‘b110 in it
- If you invoke **signed** multiplication…stuff does not “just work”. You *really* need to bit extend ahead of time to the worst case width:

```
     ‘b111101        (-3)
   * ‘b111110        (-2)
   ---------------------
     ‘b000000        (0)
    ‘b1111010        (-6)
   ‘b11110100        (-12)
  ‘b111101000        (-24)
 ‘b1111010000        (-48)
+’b11110100000       (-96)
---------------------------
‘b110011000110        (6)
```

Discard overflow

Other Operations…

- Things like equality/inequality checks as well as division, mod, etc…
- These obviously are dependent on whether we interpret the bits as signed or unsigned (no surprise there)

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	logical equality	equality
!=	logical inequality	equality
===	case equality	equality
!==	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
^~ or ~^	bit-wise XNOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Operator Precedence

- There is and always has been a very clear order in which operators get analyzed
- However some of these operators are **sign** dependent in precedence
- Sign dependence may lead to differing sign extension.
- And then weird things can happen.

<table>
<tr><th>Verilog Operator</th><th>Name</th><th>Functional Group</th></tr>
<tr><td>[ ]</td><td>bit-select or part-select</td><td></td></tr>
<tr><td>( )</td><td>parenthesis</td><td></td></tr>
<tr><td>!
~
&
|
~&
~|
^
~^ or ^~</td><td>logical negation
negation
reduction AND
reduction OR
reduction NAND
reduction NOR
reduction XOR
reduction XNOR</td><td>logical
bit-wise
reduction
reduction
reduction
reduction
reduction
reduction</td></tr>
<tr><td>+
-</td><td>unary (sign) plus
unary (sign) minus</td><td>arithmetic
arithmetic</td></tr>
<tr><td>{ }</td><td>concatenation</td><td>concatenation</td></tr>
<tr><td>{{ }}</td><td>replication</td><td>replication</td></tr>
<tr><td>*
/
%</td><td>multiply
divide
modulus</td><td>arithmetic
arithmetic
arithmetic</td></tr>
<tr><td>+
-</td><td>binary plus
binary minus</td><td>arithmetic
arithmetic</td></tr>
<tr><td><<
>></td><td>shift left
shift right</td><td>shift
shift</td></tr>
<tr><td>>
>=
<
<=</td><td>greater than
greater than or equal to
less than
less than or equal to</td><td>relational
relational
relational
relational</td></tr>
<tr><td>==
!=</td><td>logical equality
logical inequality</td><td>equality
equality</td></tr>
<tr><td>===
!==</td><td>case equality
case inequality</td><td>equality
equality</td></tr>
<tr><td>&</td><td>bit-wise AND</td><td>bit-wise</td></tr>
<tr><td>^
^~ or ~^</td><td>bit-wise XOR
bit-wise XNOR</td><td>bit-wise
bit-wise</td></tr>
<tr><td>|</td><td>bit-wise OR</td><td>bit-wise</td></tr>
<tr><td>&&</td><td>logical AND</td><td>logical</td></tr>
<tr><td>||</td><td>logical OR</td><td>logical</td></tr>
<tr><td>?:</td><td>conditional</td><td>conditional</td></tr>
</table>

Also using a “-” does not make a thing signed

- The unary operator “-” just does $-A = 1 + \sim A$
- The result is not inherently signed.
- So don’t expect `-(2’d2)` to be a signed thing (for the purposes of operator determination)

Conclusions

- It seems like Verilog is strongly inclined towards unsigned numbers. Any of the following yield an unsigned value:
  - Any operation on two operands, unless both operands are signed.
  - Numbers given with an explicit base (e.g. 12’d10), unless the explicit "s" modifier is used)
  - Results of bit-select
  - Results of part-select
  - Concatenations
- Be careful of hidden sign extensions!
- Be careful of small one bit or two bit signed numbers…the patterns of two’s complement stuff gets fuzzy at one bit.
- Use **$signed** as needed…results in ugly code, but can make things safe.

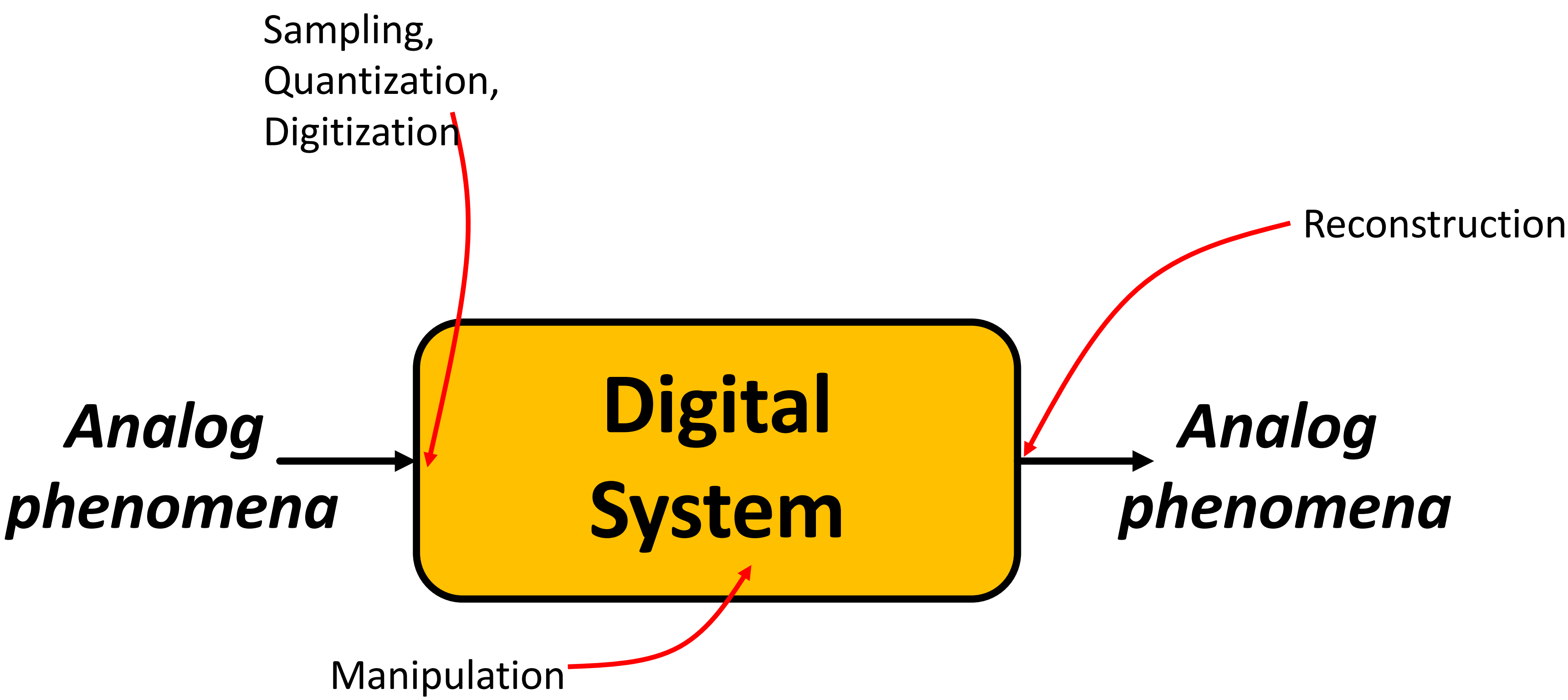
(https://www.01signal.com/verilog-design/arithmetic/signed-wire-reg/)

DSP Concepts

Digital Signal Processing

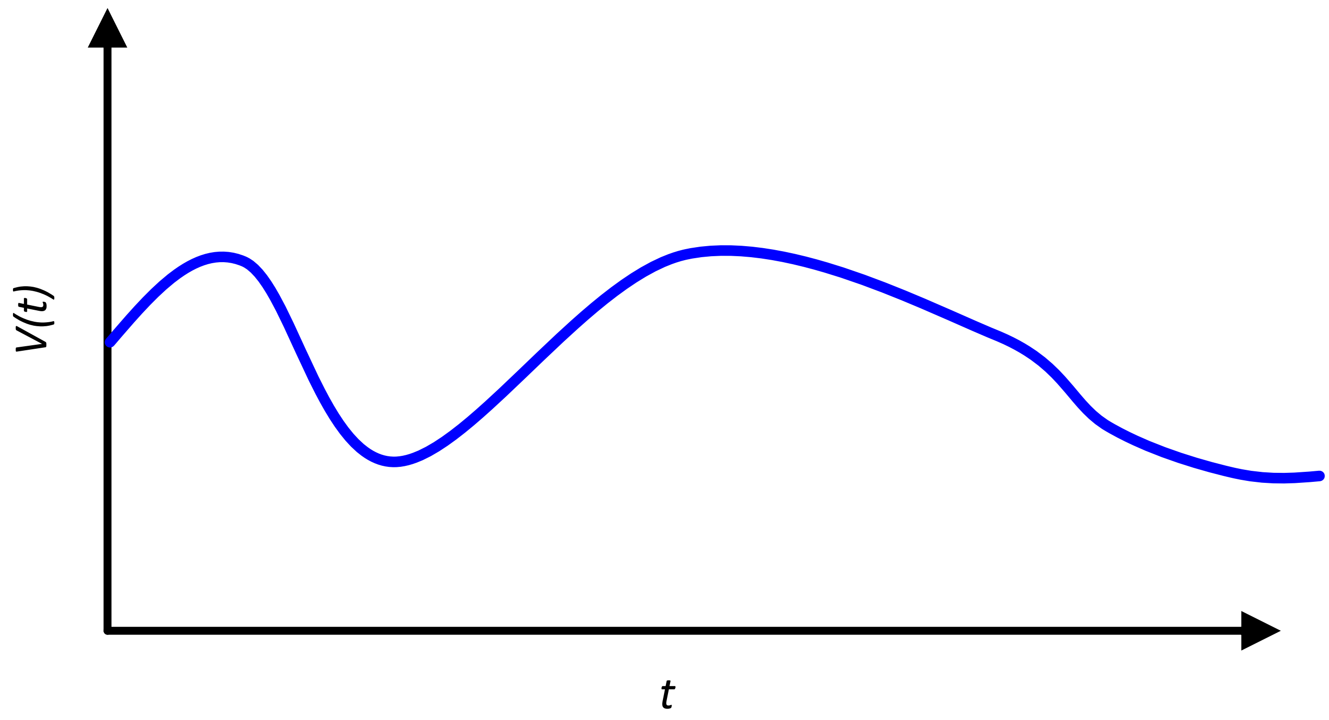
A Digital System in an Analog World

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

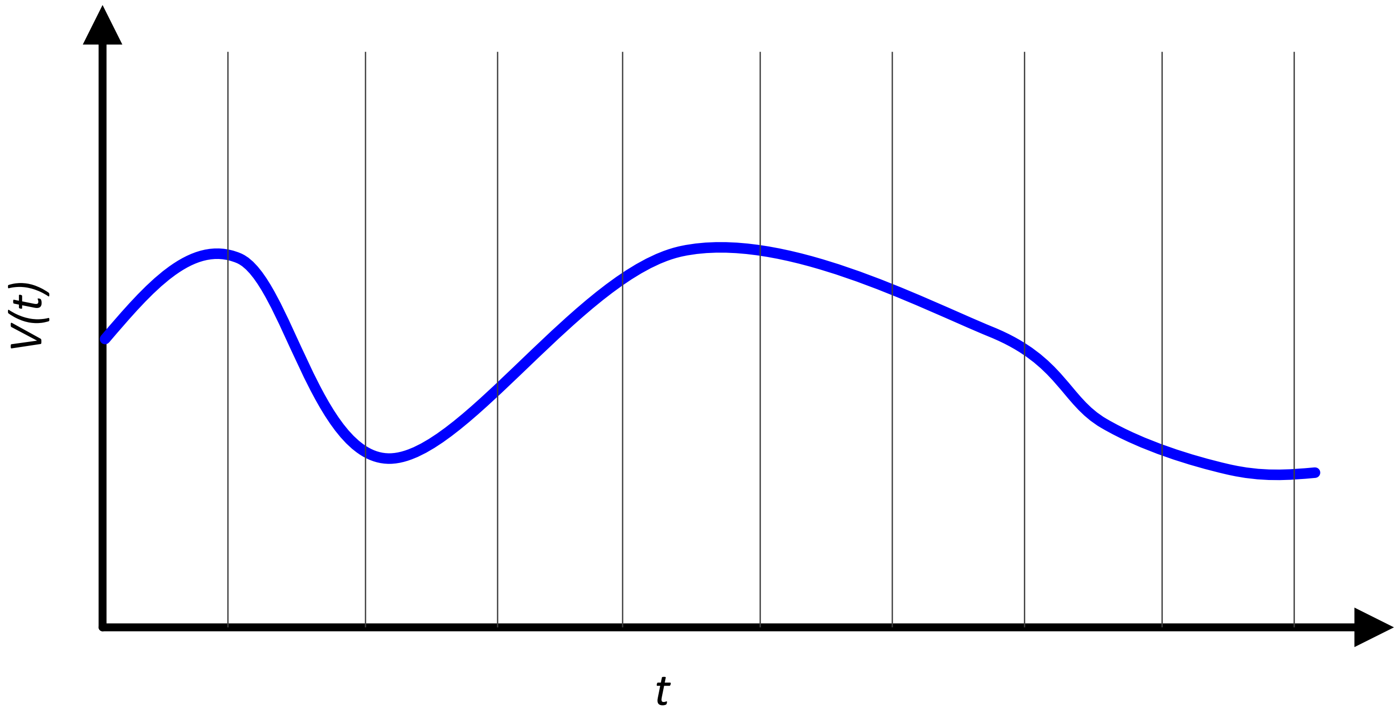


Visualizing Sampling

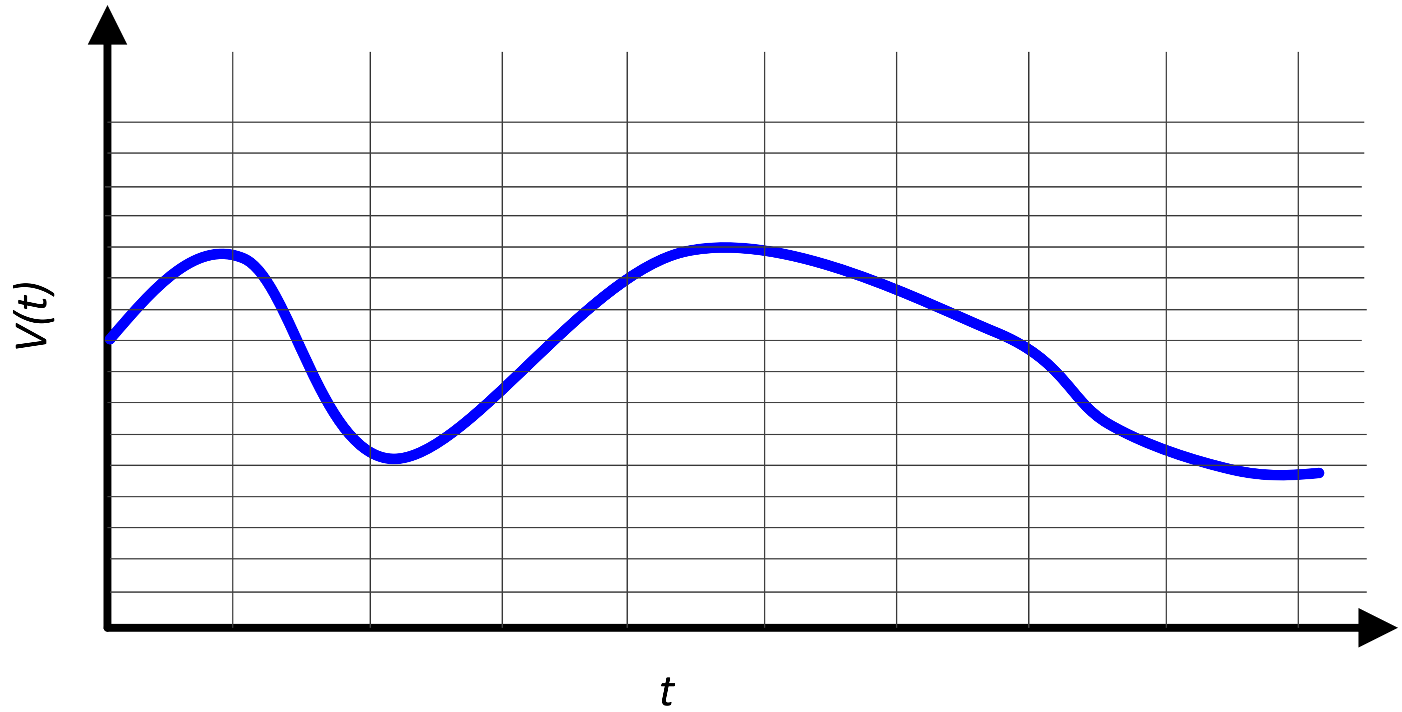
Continuous in Value and in Time



Discretization in Time

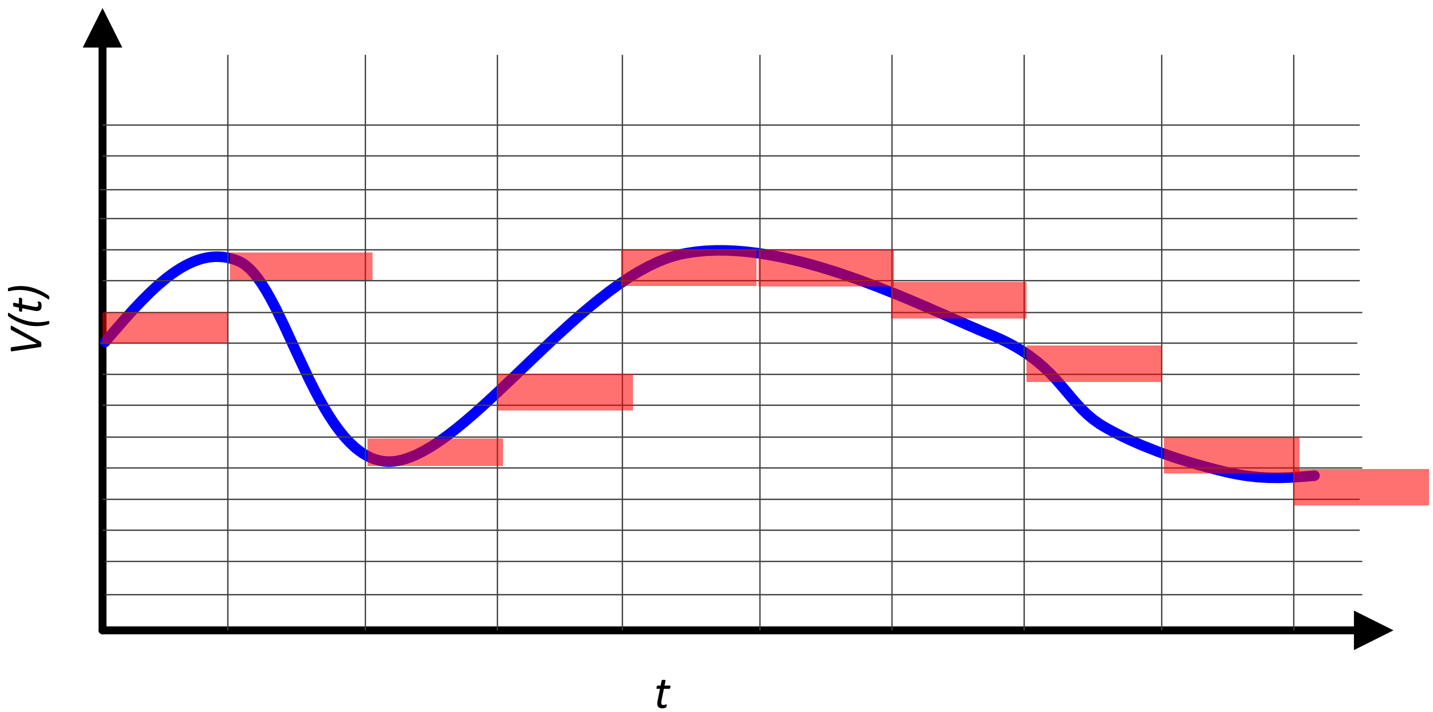


Discretization in Time and **Quantization** in Value



4 bit value encoding

Discretization in Time and **Quantization** in Value



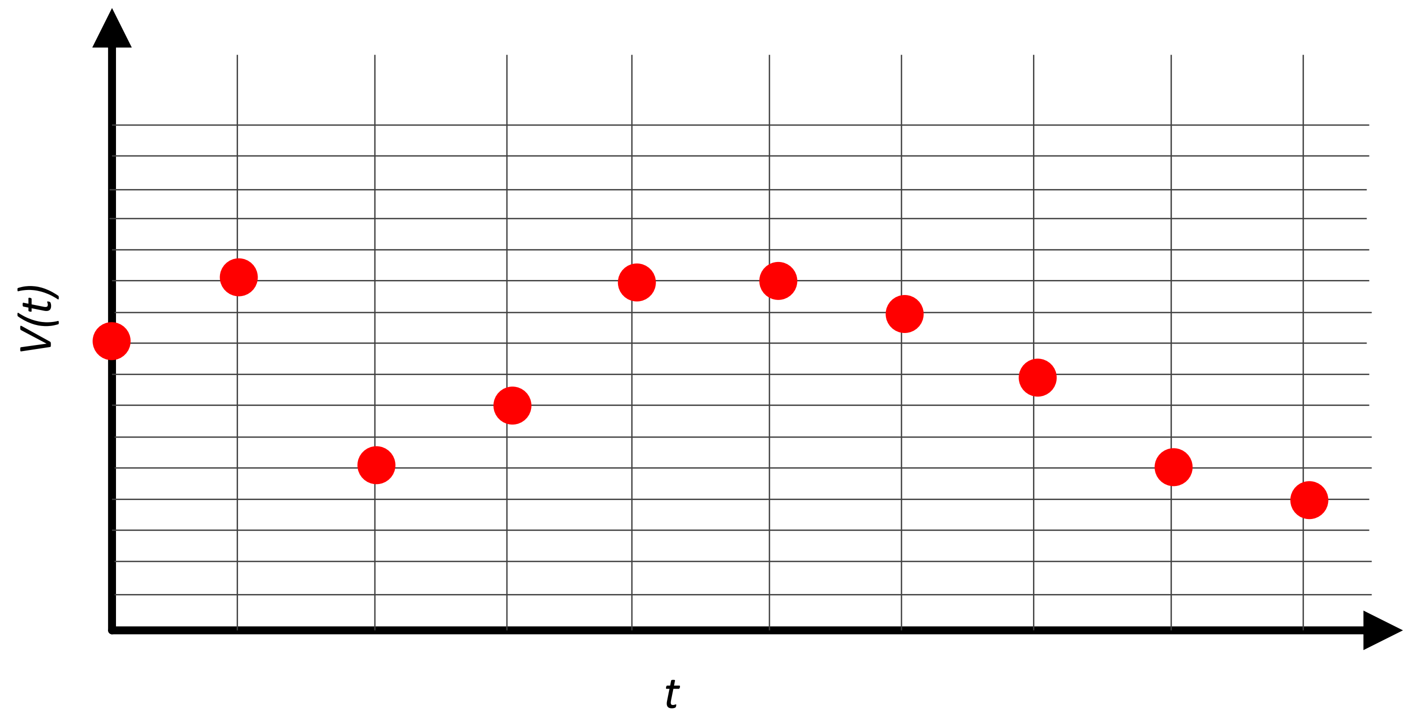
v[n] = [9,11,5,7,11,11,10,8,5,4,]

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!
- Good stuff. That's not a lot!

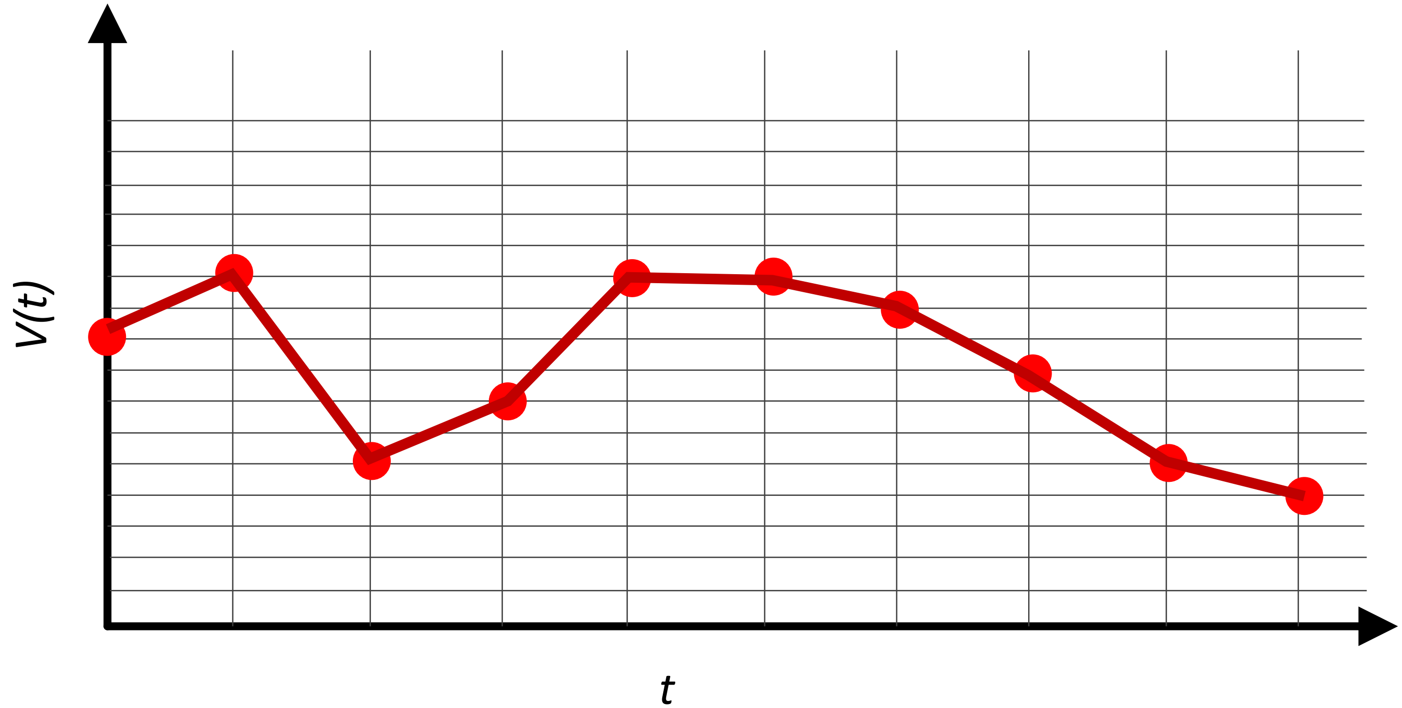
Reconstruction of Signal



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

4 bit value encoding

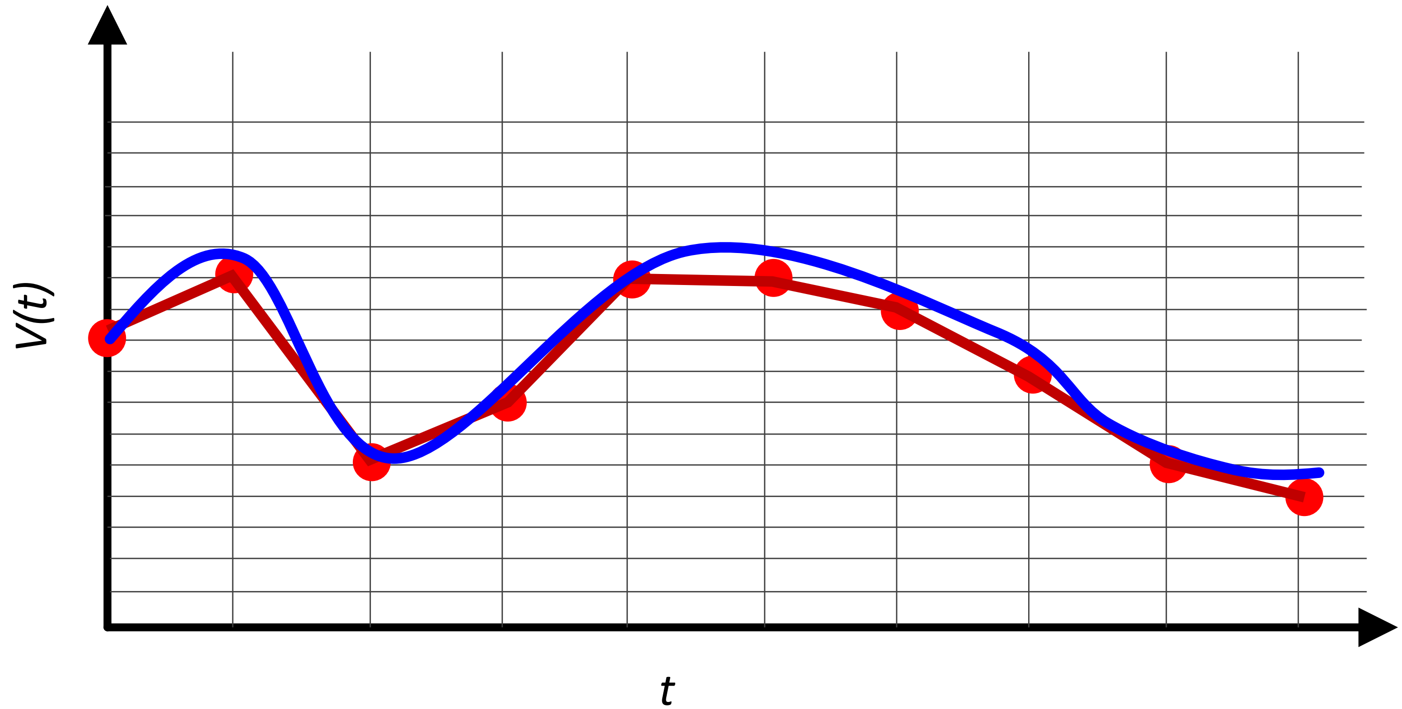
Reconstruction (with first-order hold interpolation)



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,]

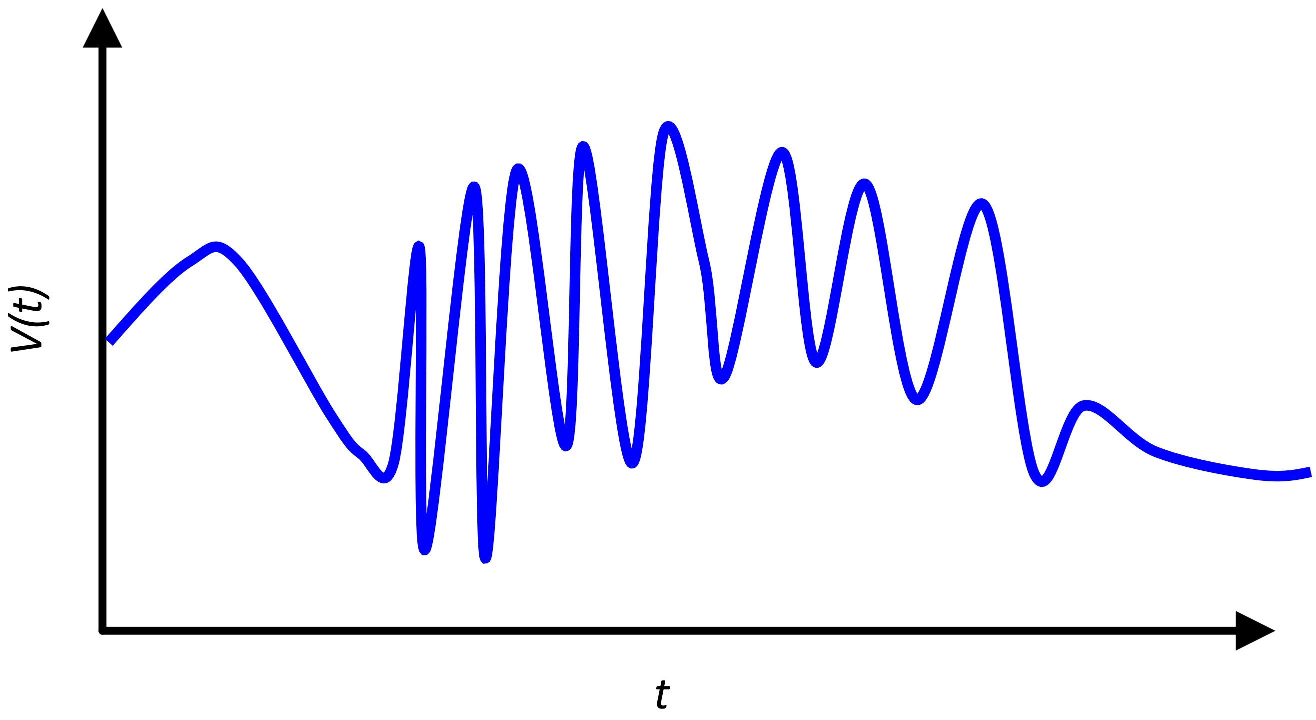
4 bit value encoding

Errors

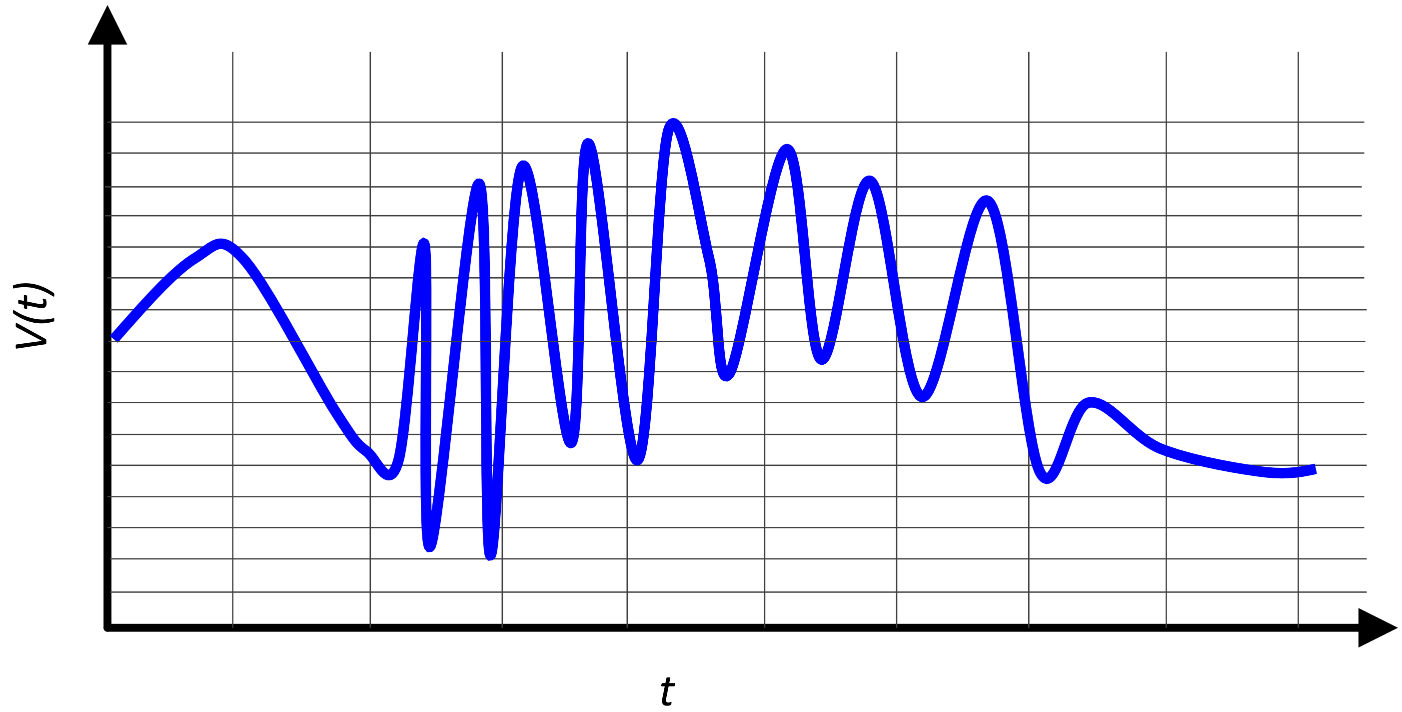
- **Discretization Error:** How ”off” our readings are in time due to sampling at discrete intervals

- **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

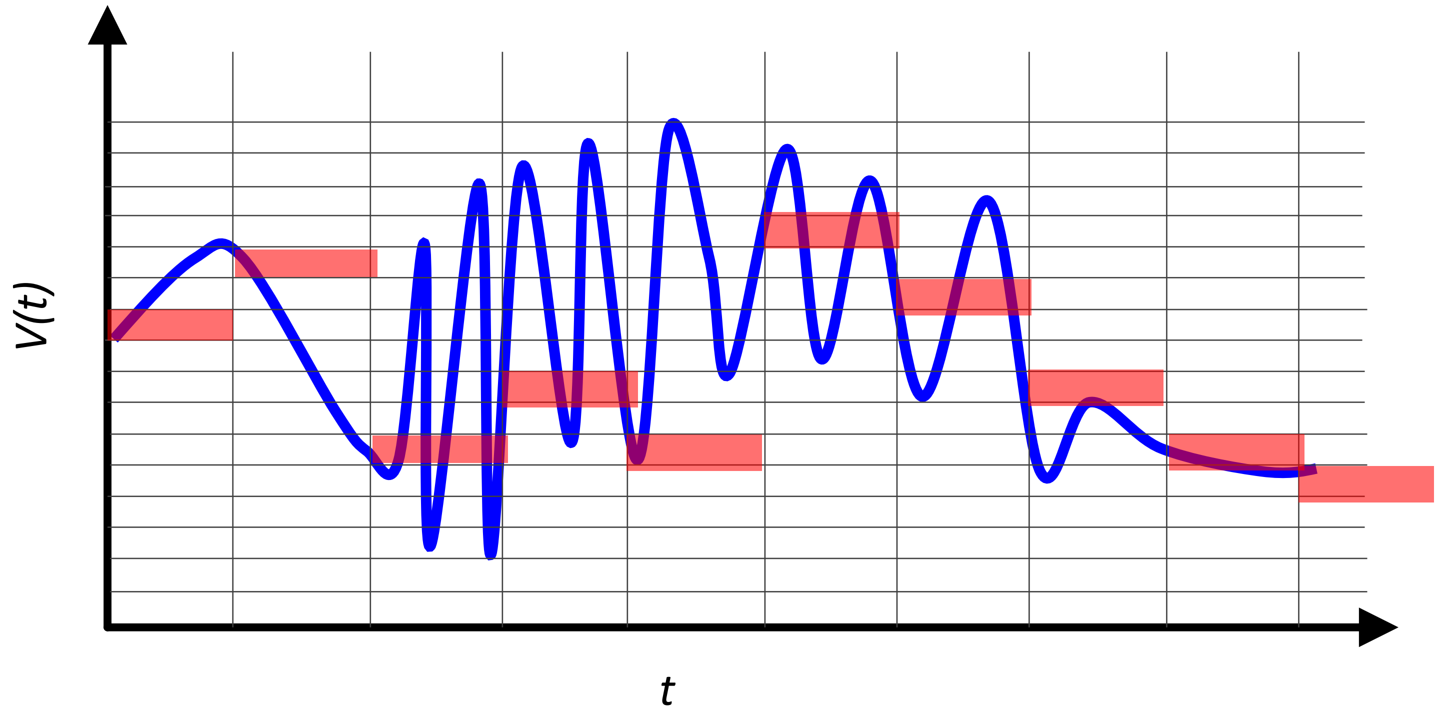


Discretization in Time and **Quantization** in Value



4 bit value encoding

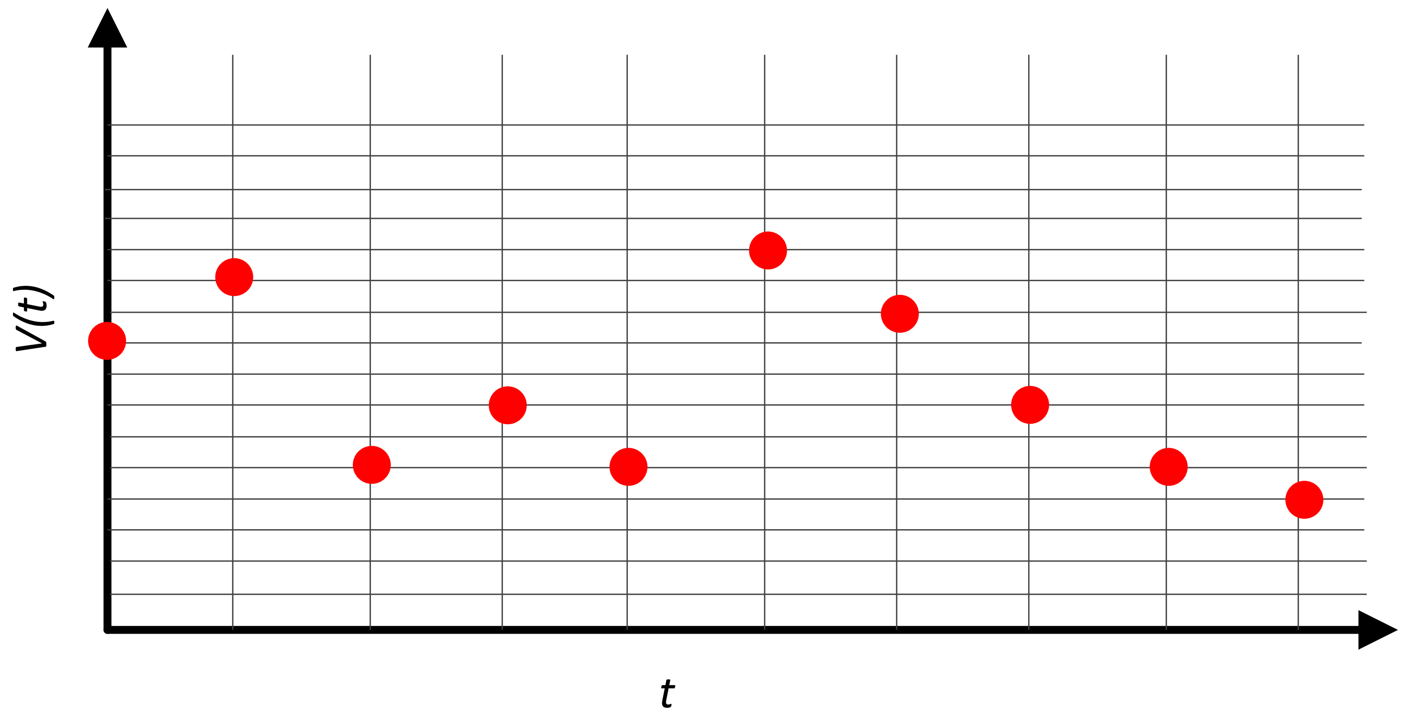
Discretization in Time and **Quantization** in Value



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

4 bit value encoding

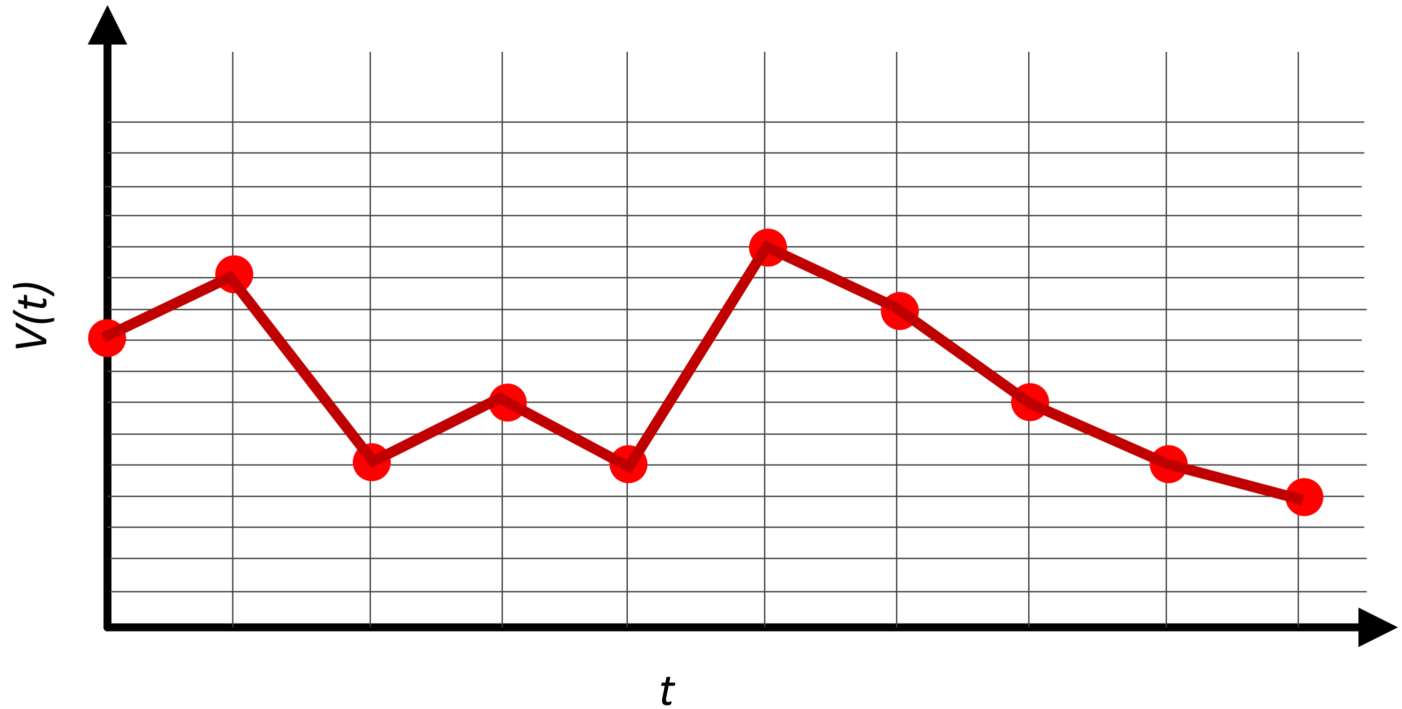
Reproduce



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

4 bit value encoding

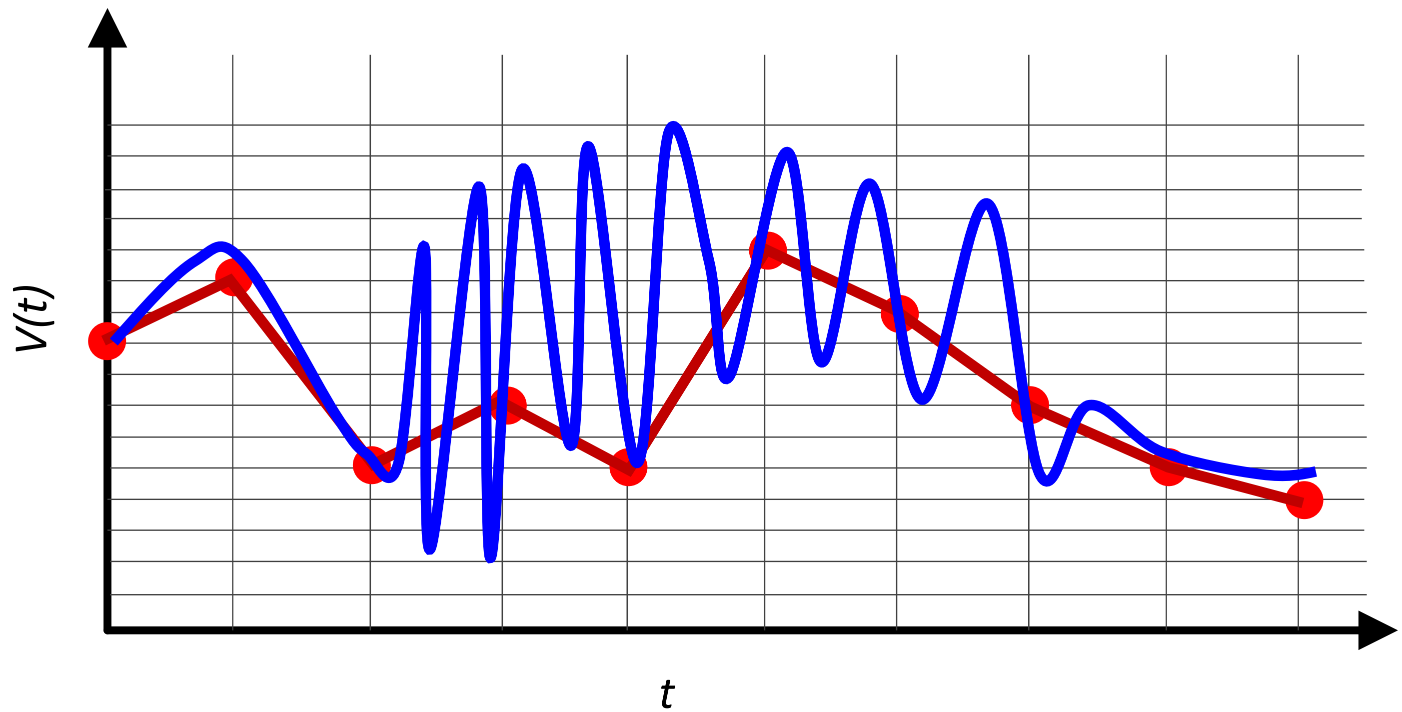
Reproduce



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

4 bit value encoding

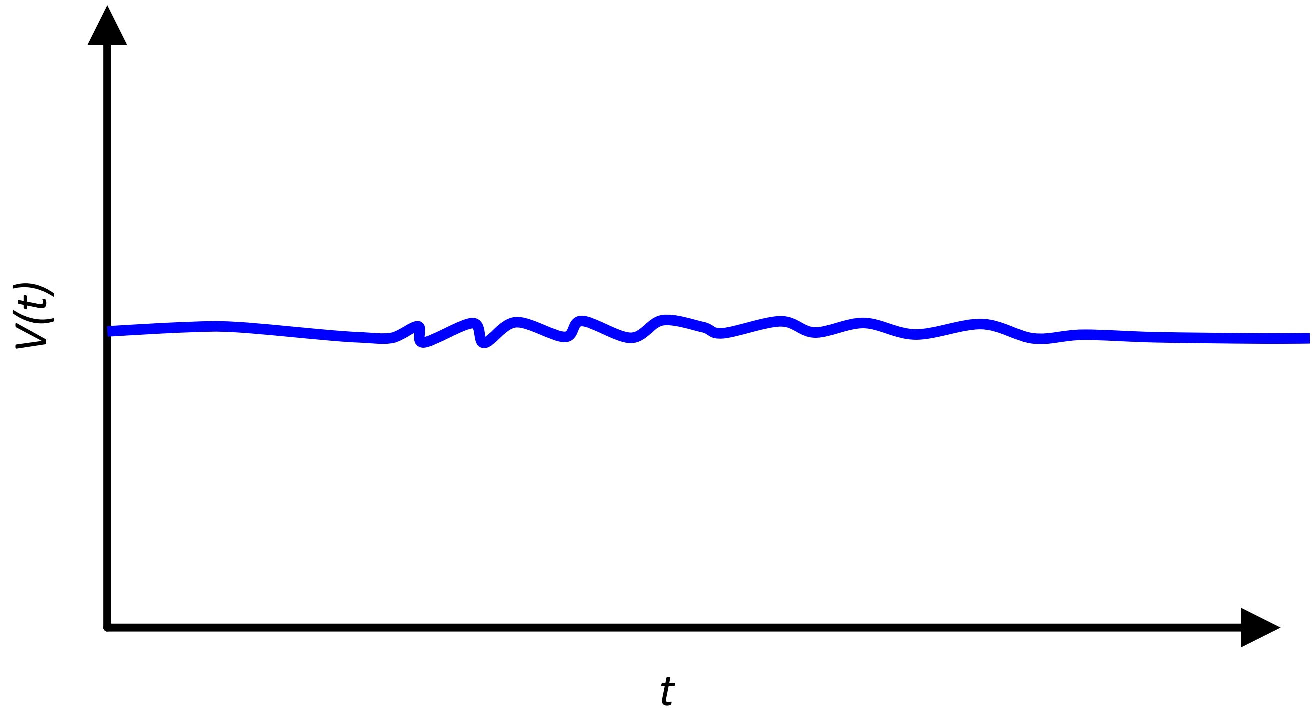
Compare to original... Did not Capture the high-frequency Wiggles!



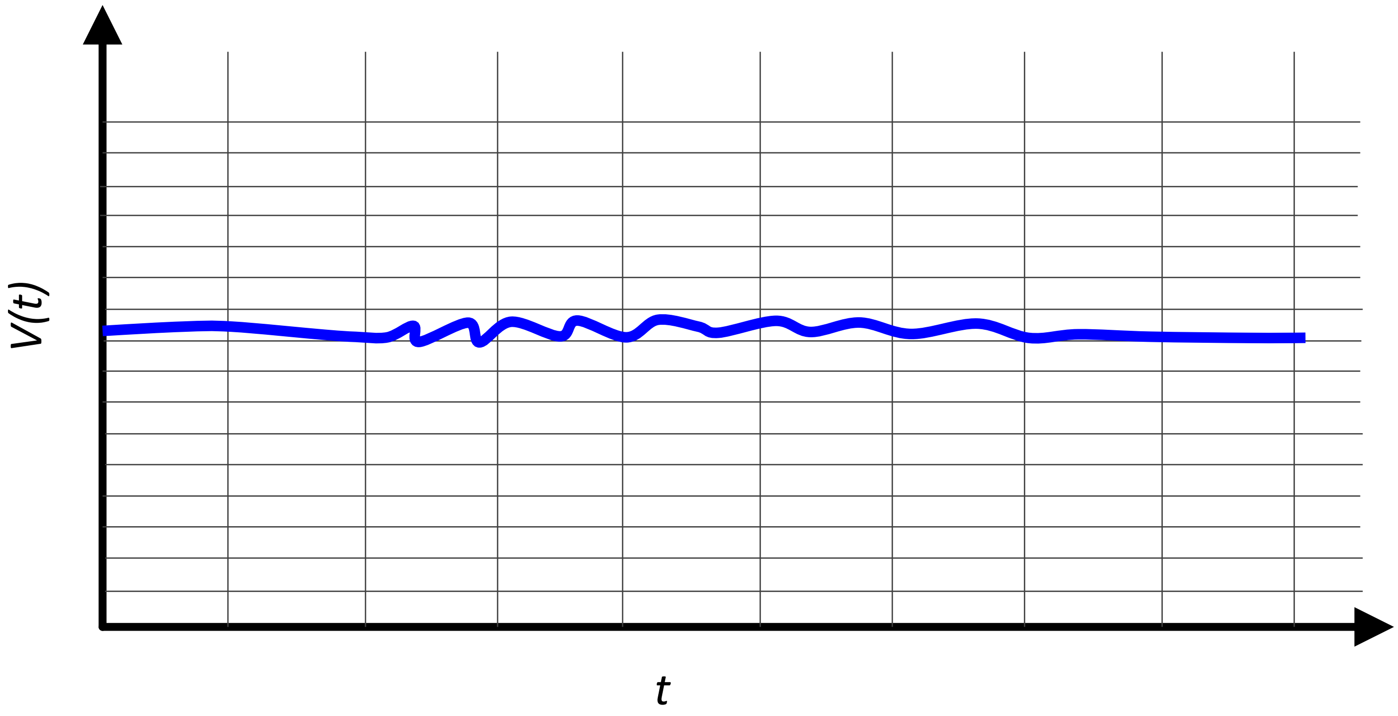
v[n] = [9,11,5,7,5,12,10,7,5,4,]

Potentially Bad Discretization Error

Continuous in Value and in Time

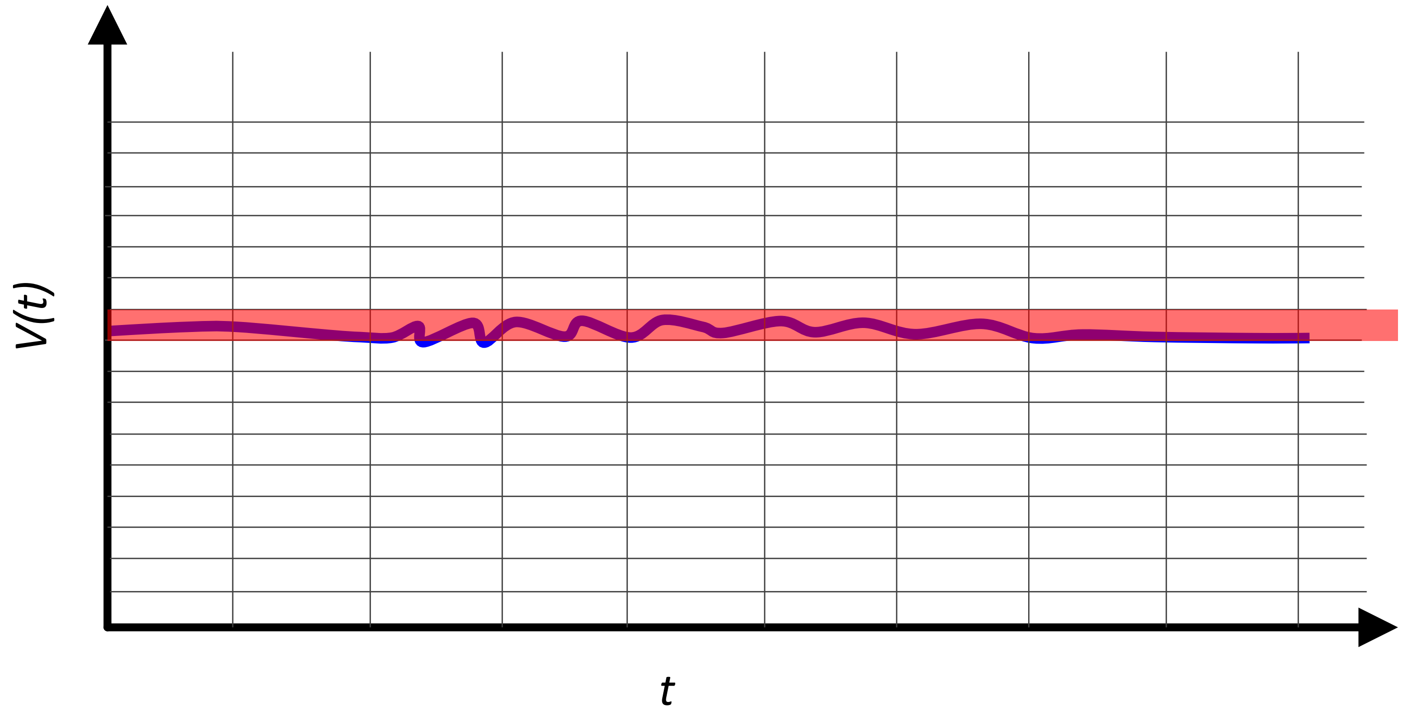


Discretization in Time and **Quantization** in Value



4 bit value encoding

Discretization in Time and **Quantization** in Value



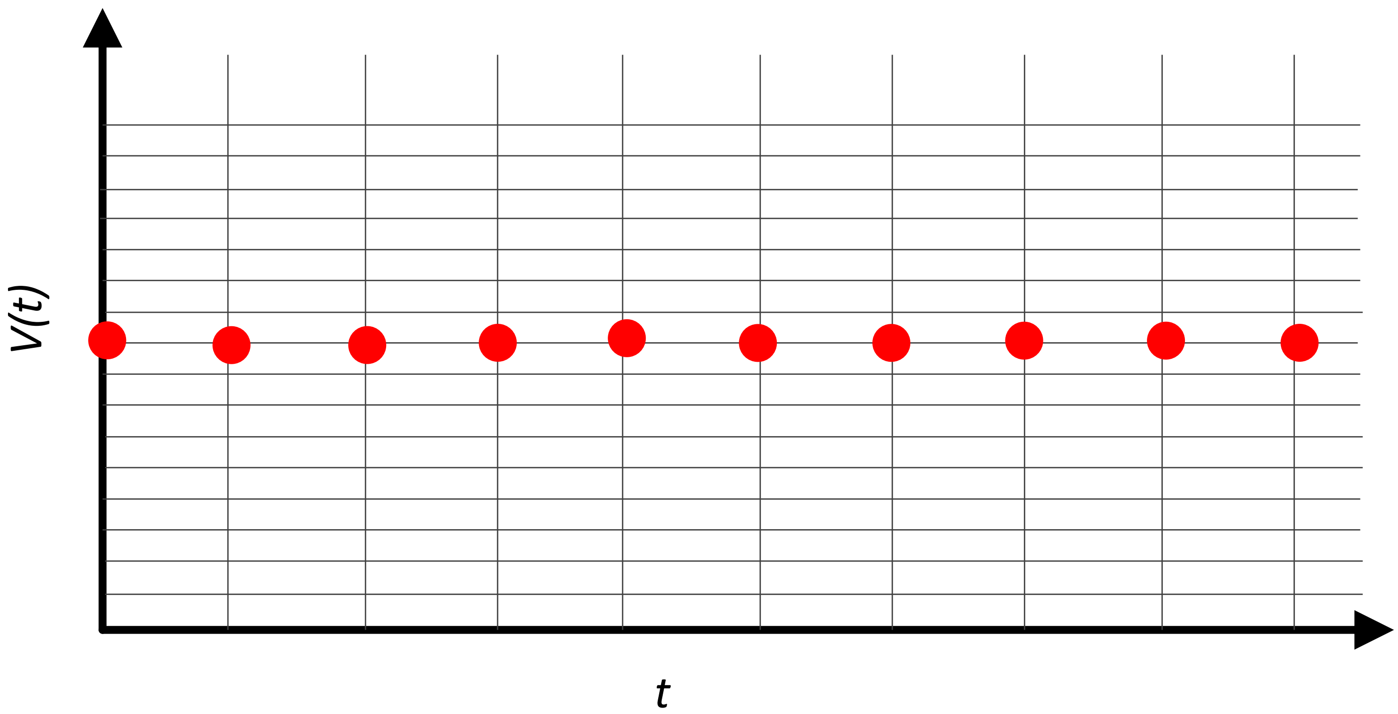
v[n] = [9,9,9,9,9,9,9,9,9,9]

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.

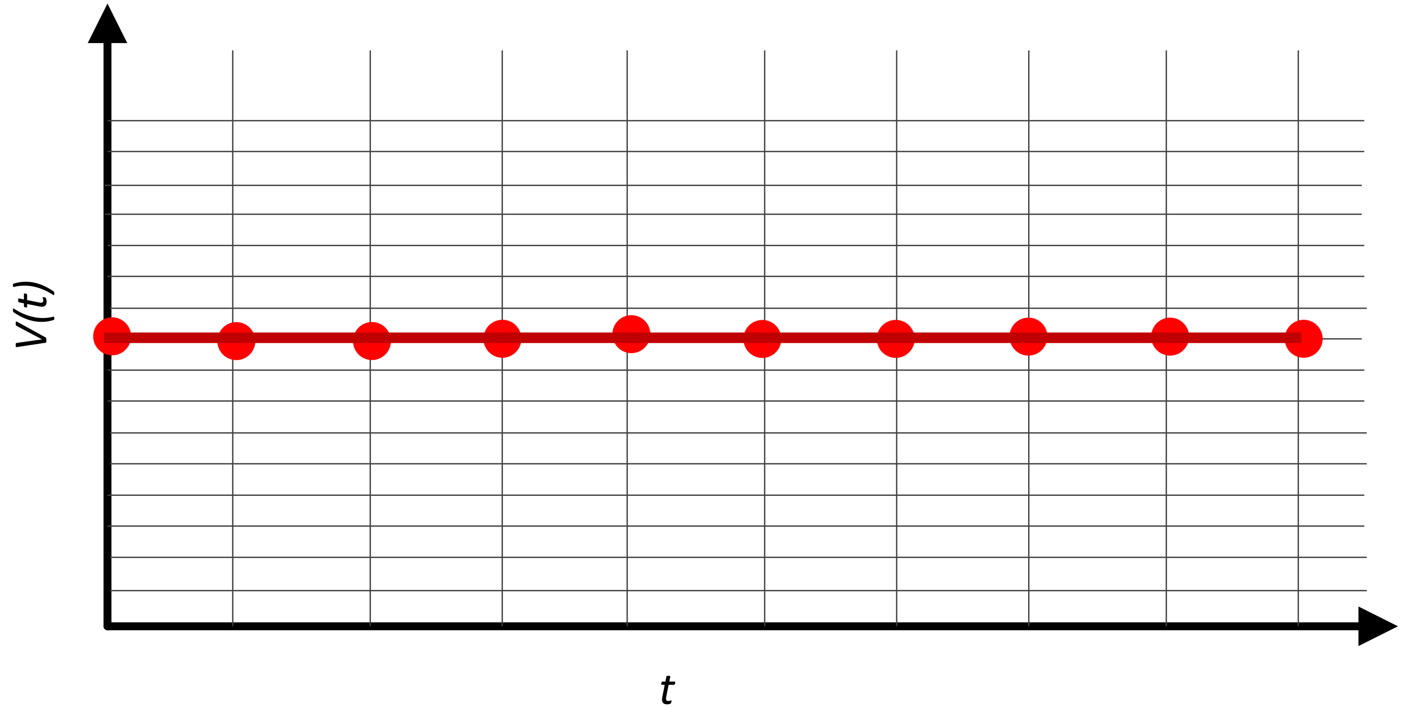
Reproduce



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

4 bit value encoding

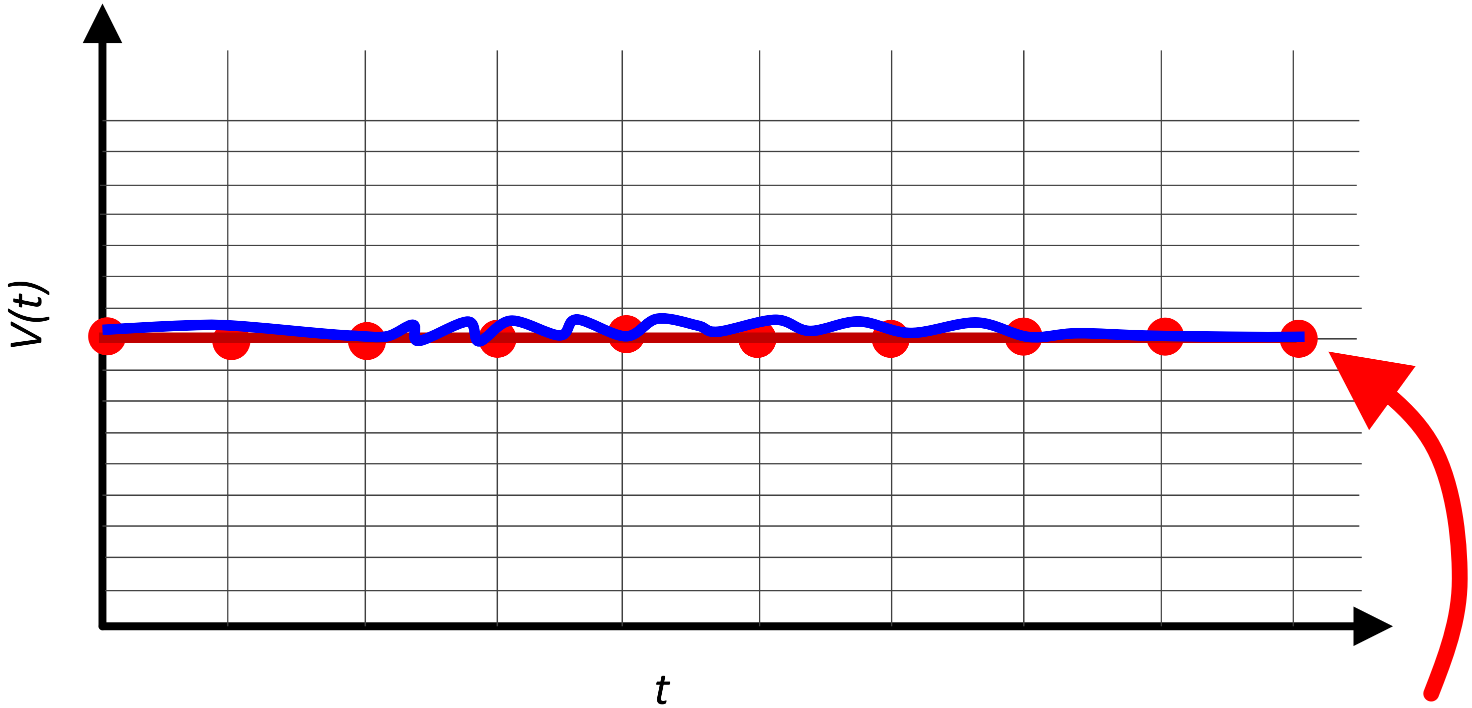
Reproduce



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

<u>Quantization Error!</u>

Those tiny wiggles might be really important in certain contexts!

Rodent heartbeats!

Conclusions

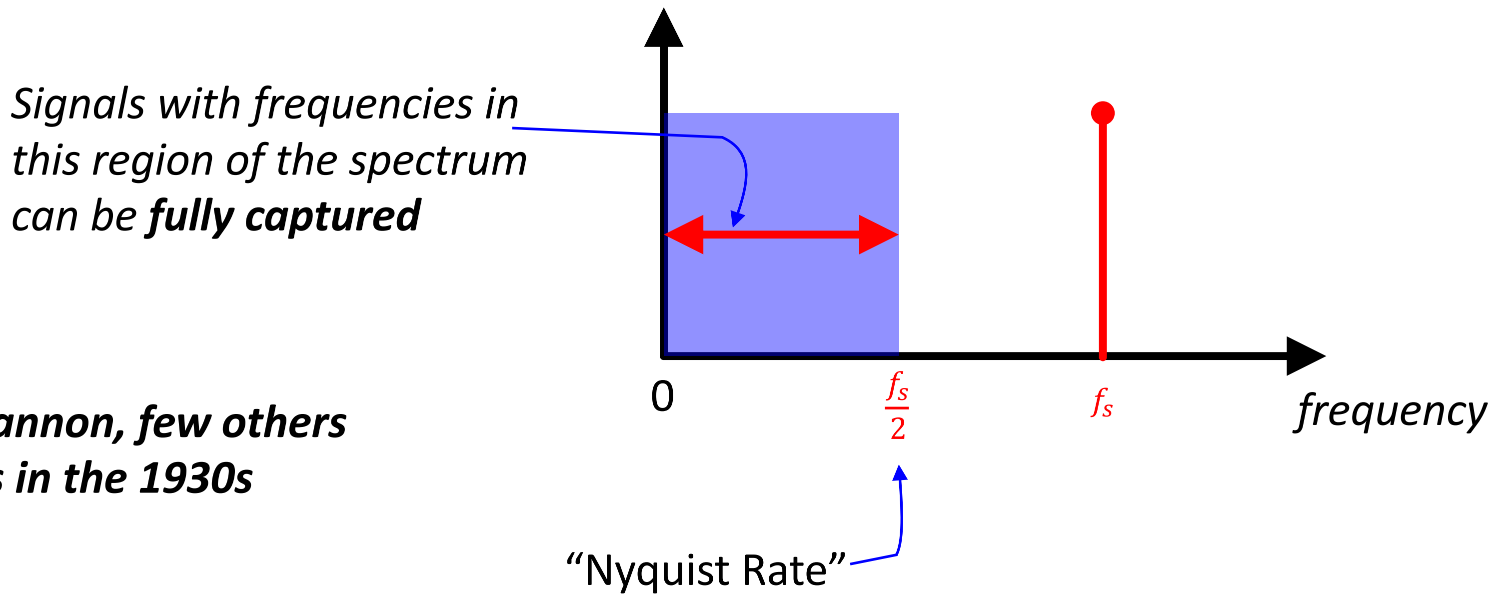
- Care must be taken when choosing what rate you sample (**discretize**) your signal and at what bit-depth you **quantize** your sample
- There's no right answer, since it depends on context/use cases.
- Ideally want to sample at high rate and quantize with many bits…
- But taken to the extreme this 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 (we’ll revisit quantization later)

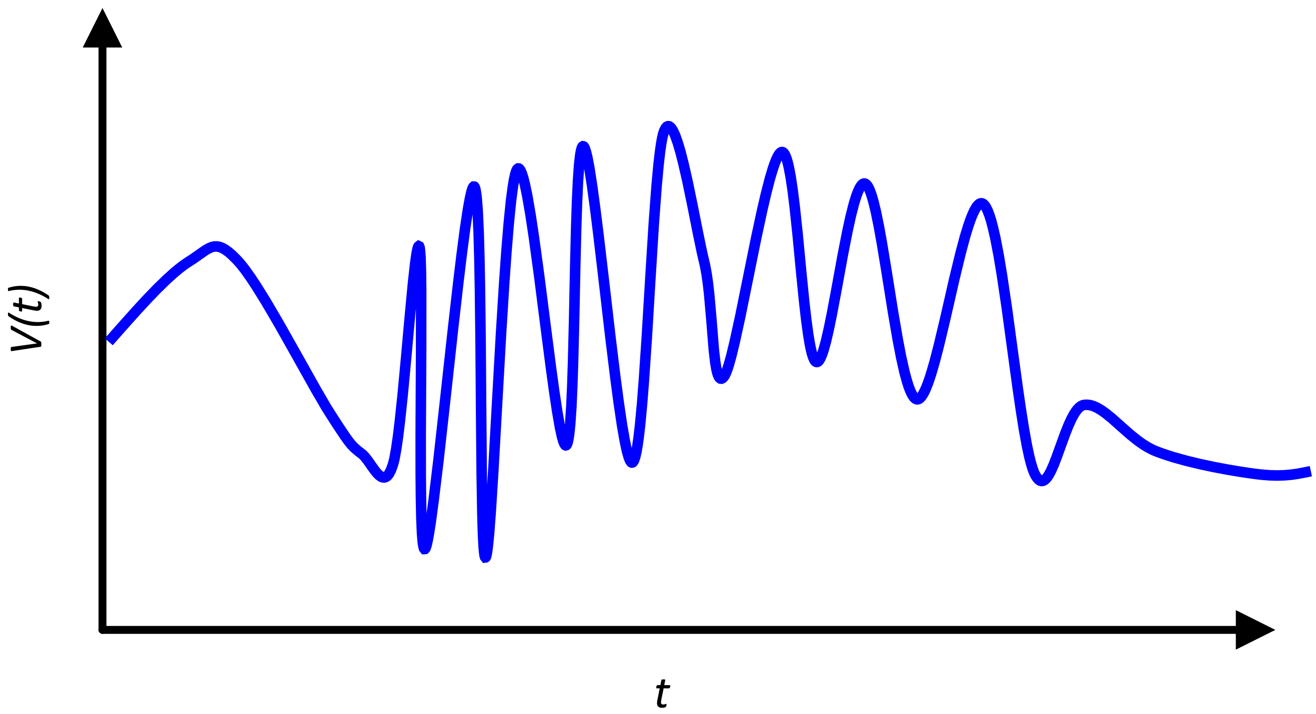
Sample Rate

- 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 $\frac{f_s}{2}$

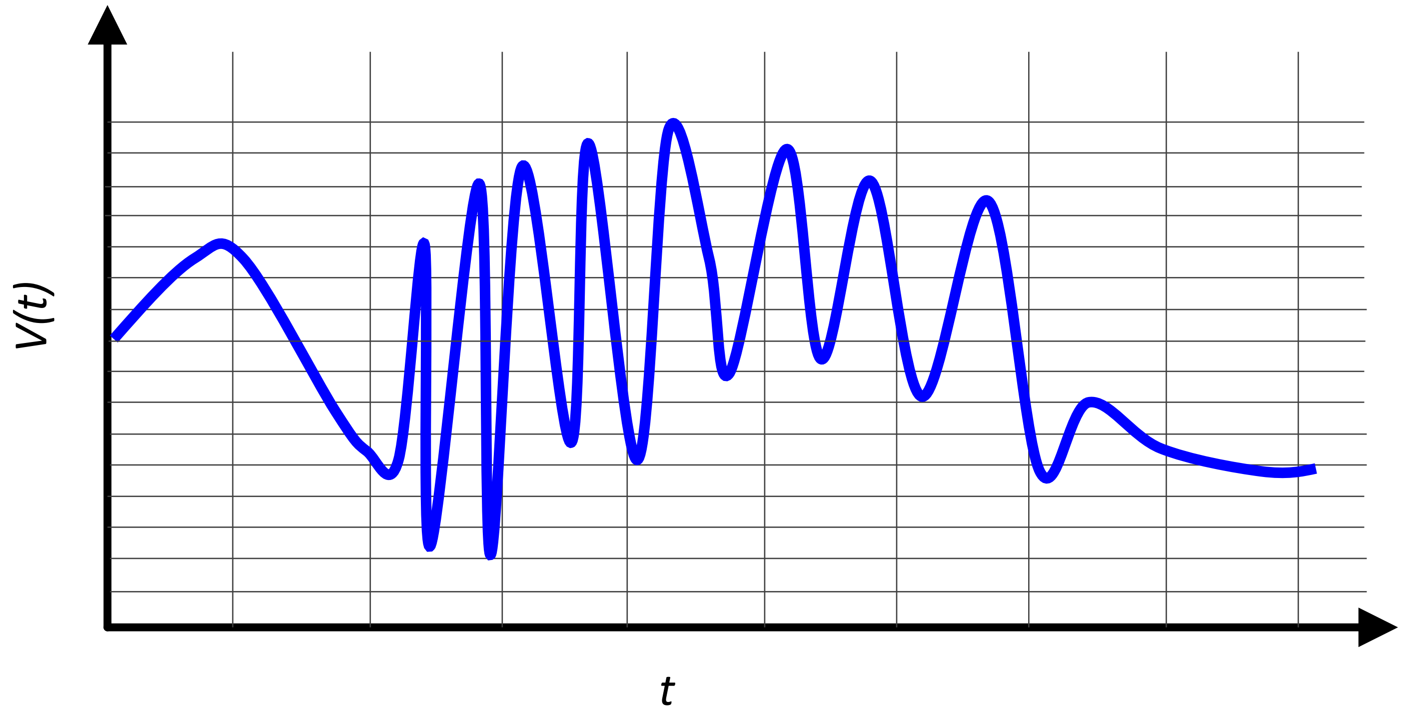


Nyquist, Shannon, few others showed this in the 1930s

Let's consider this situation though....

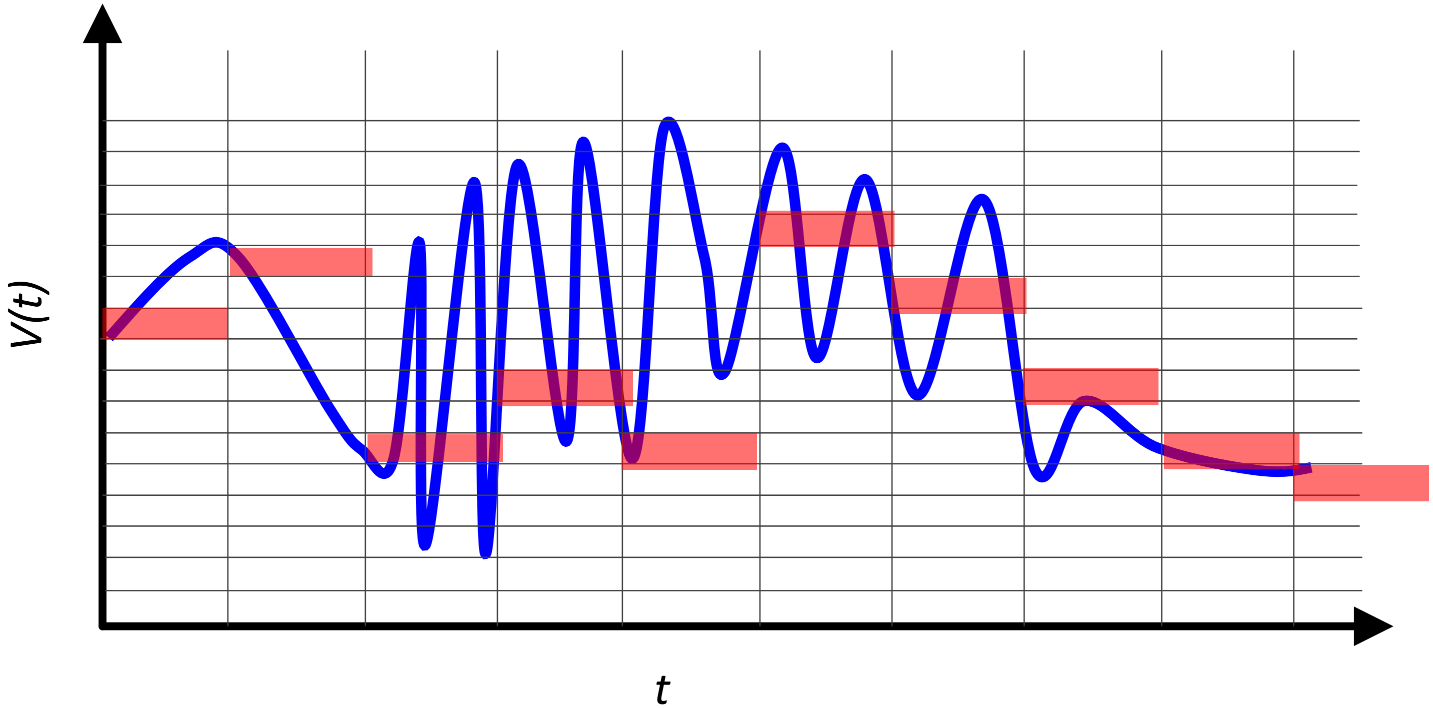


Let’s digitize it…at this sample rate we shouldn’t be able to capture it



4 bit value encoding

Discretization in Time and Quantization in Value



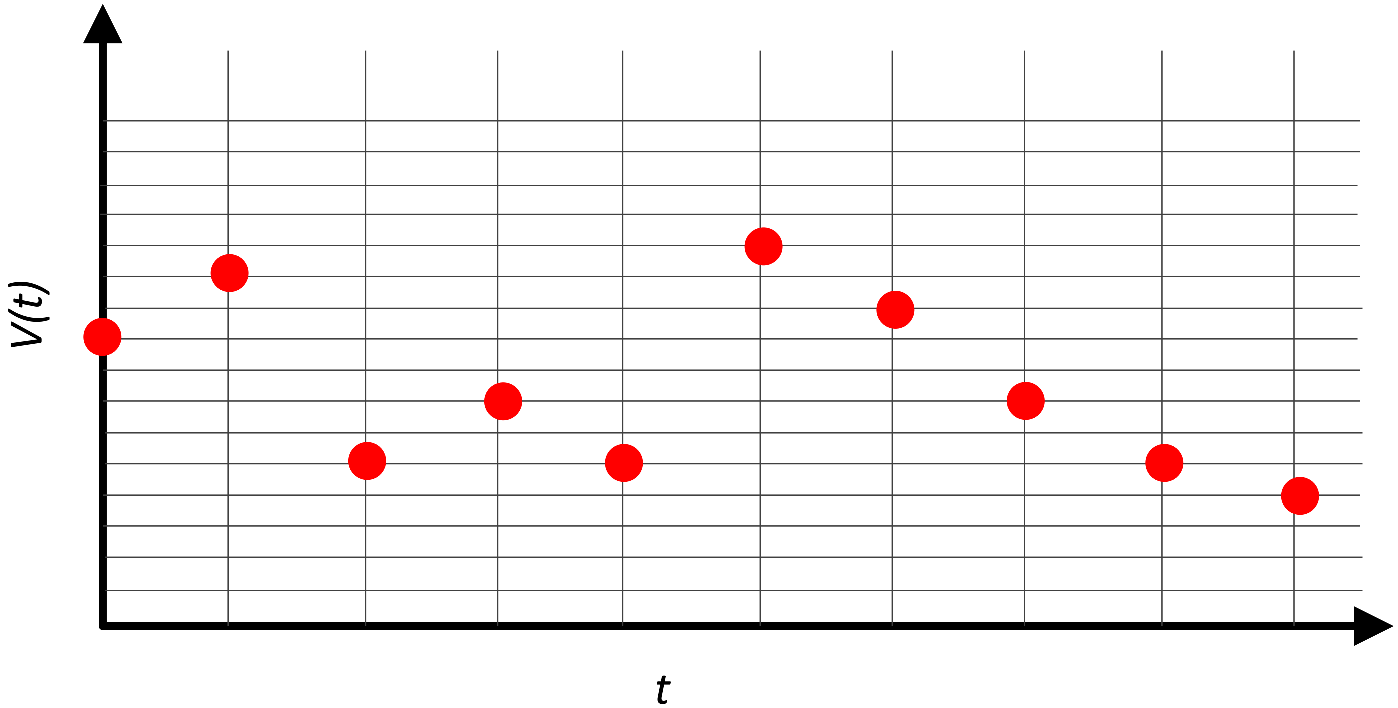
v[n] = [9,11,5,7,5,12,10,7,5,4,]

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

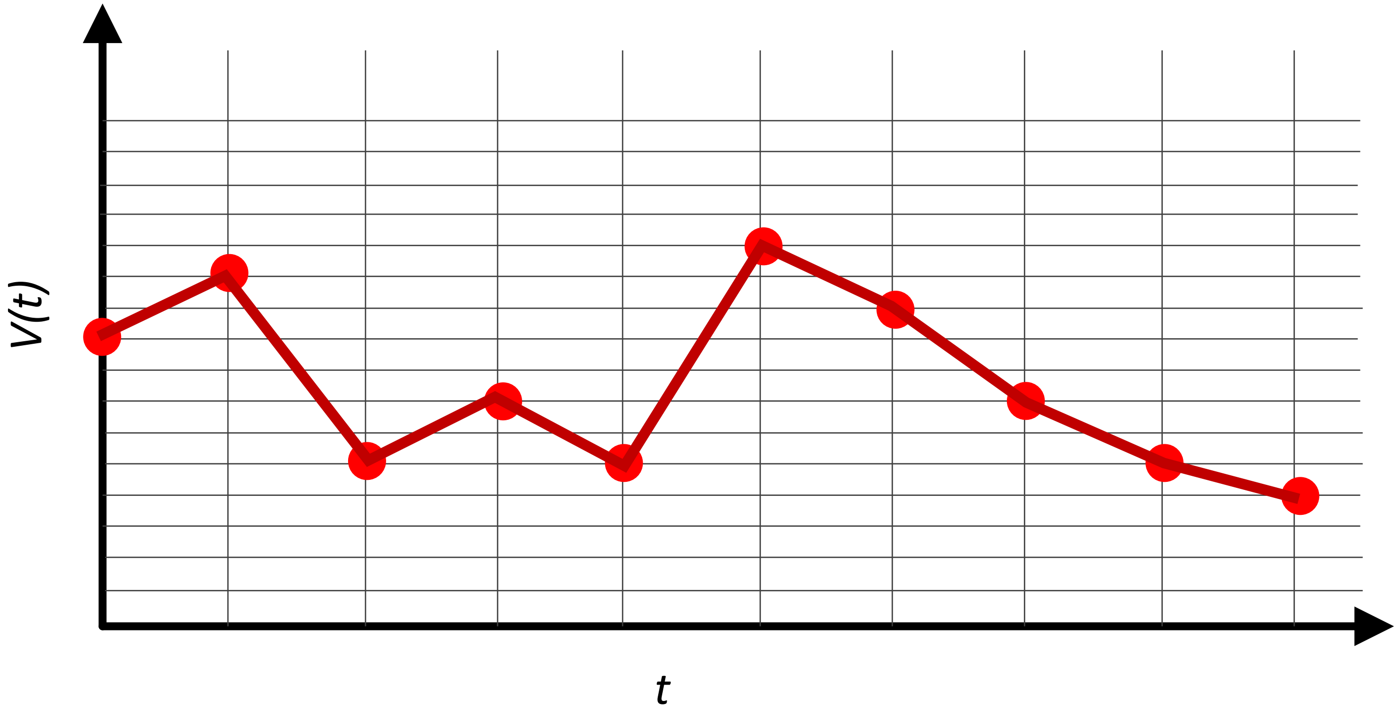
Reconstruct



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

4 bit value encoding

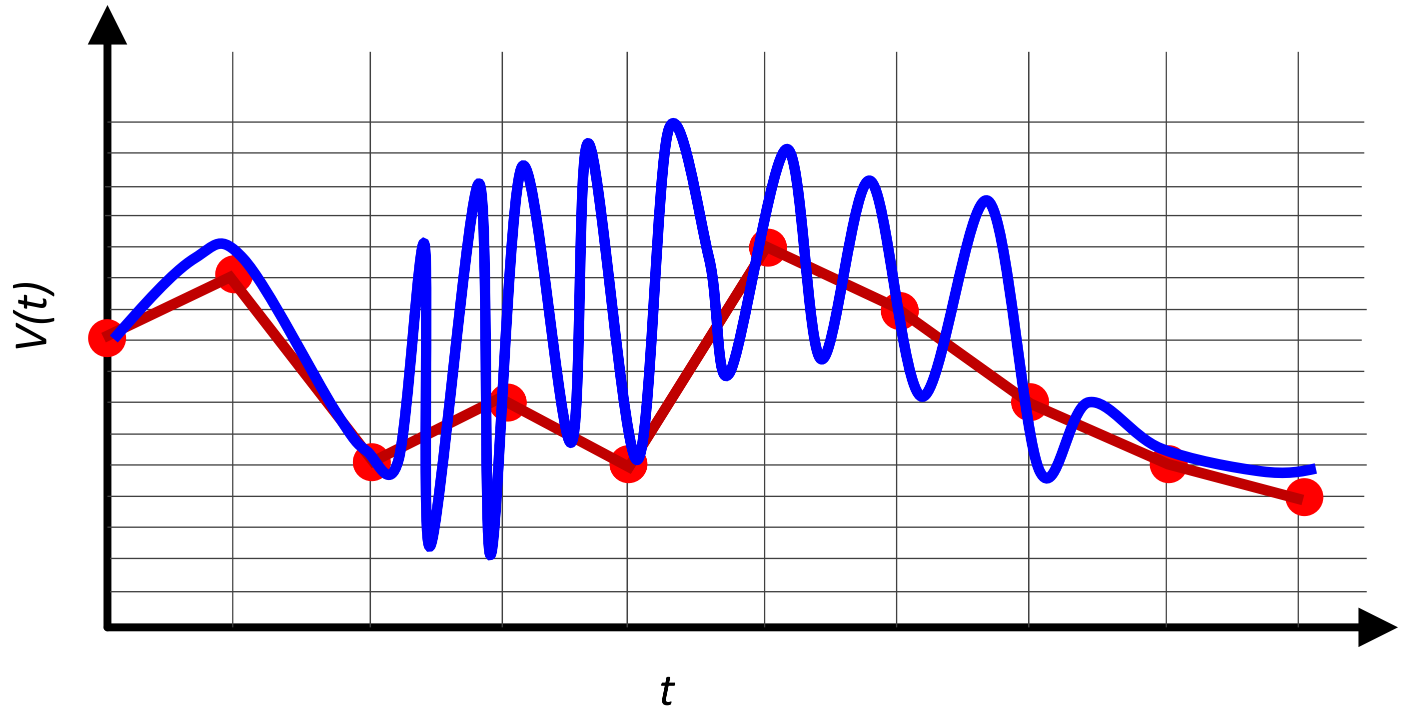
Reproduce



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

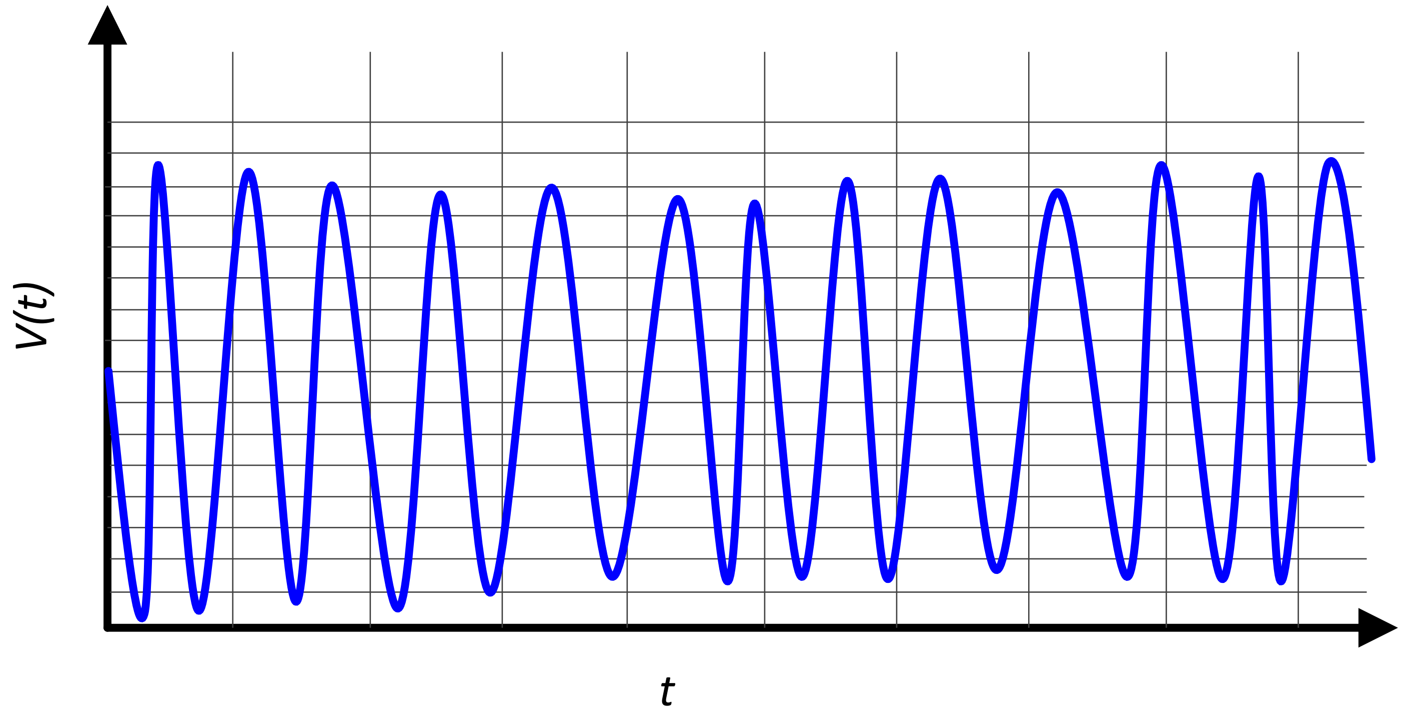
4 bit value encoding

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

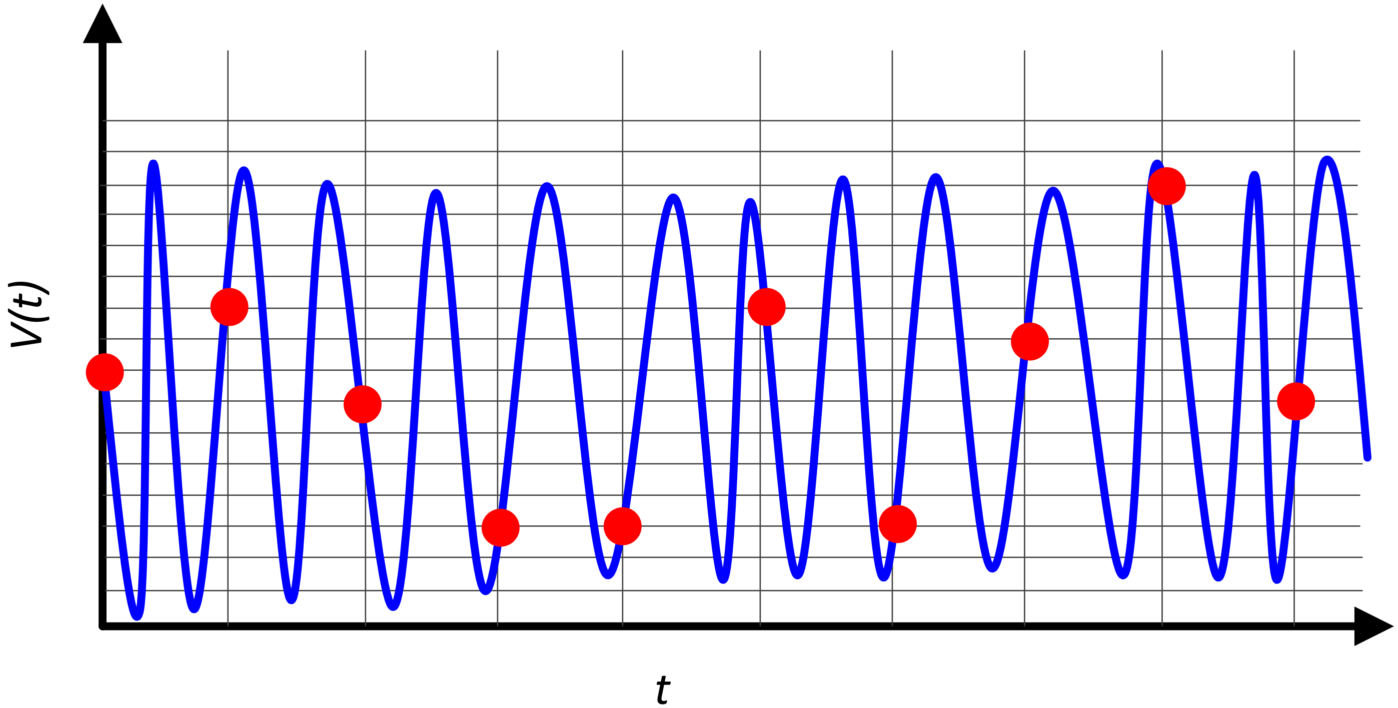


*Great….but we still captured something! What **is** that signal expressed by the red interpolation?*

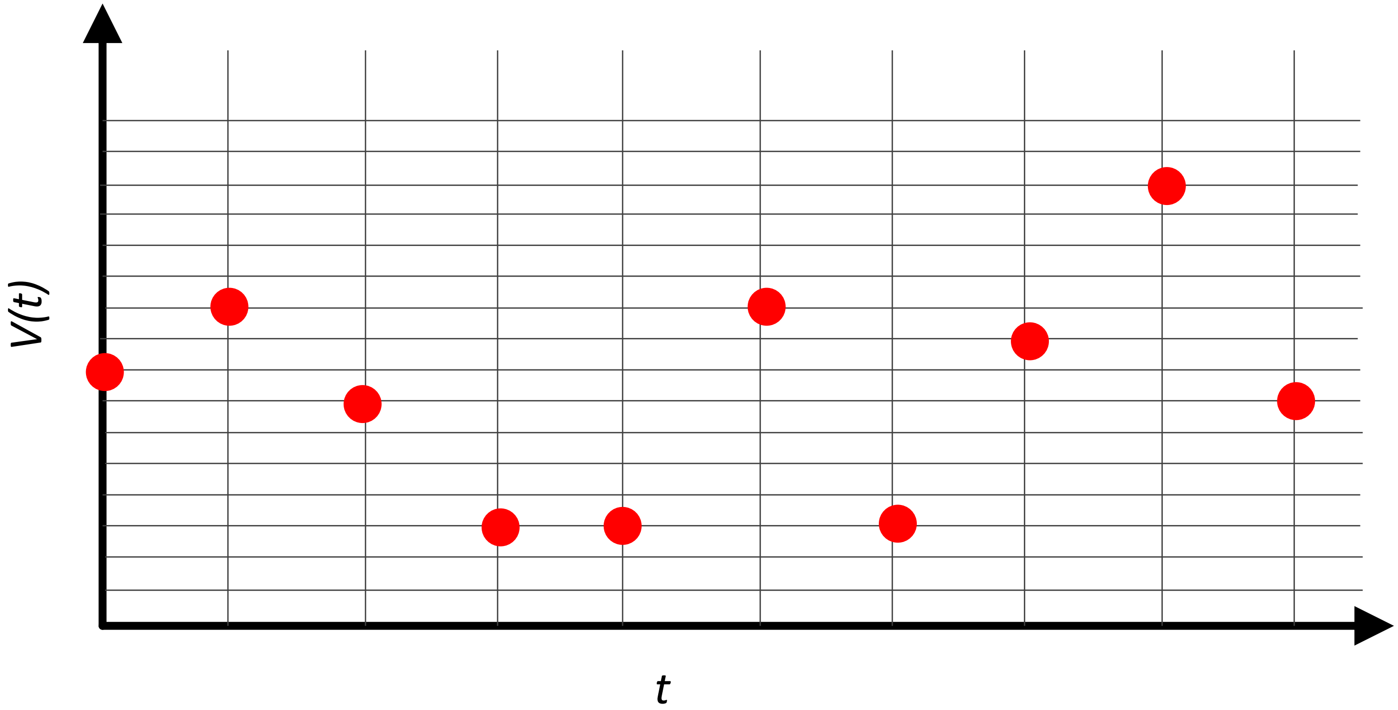
Consider this...



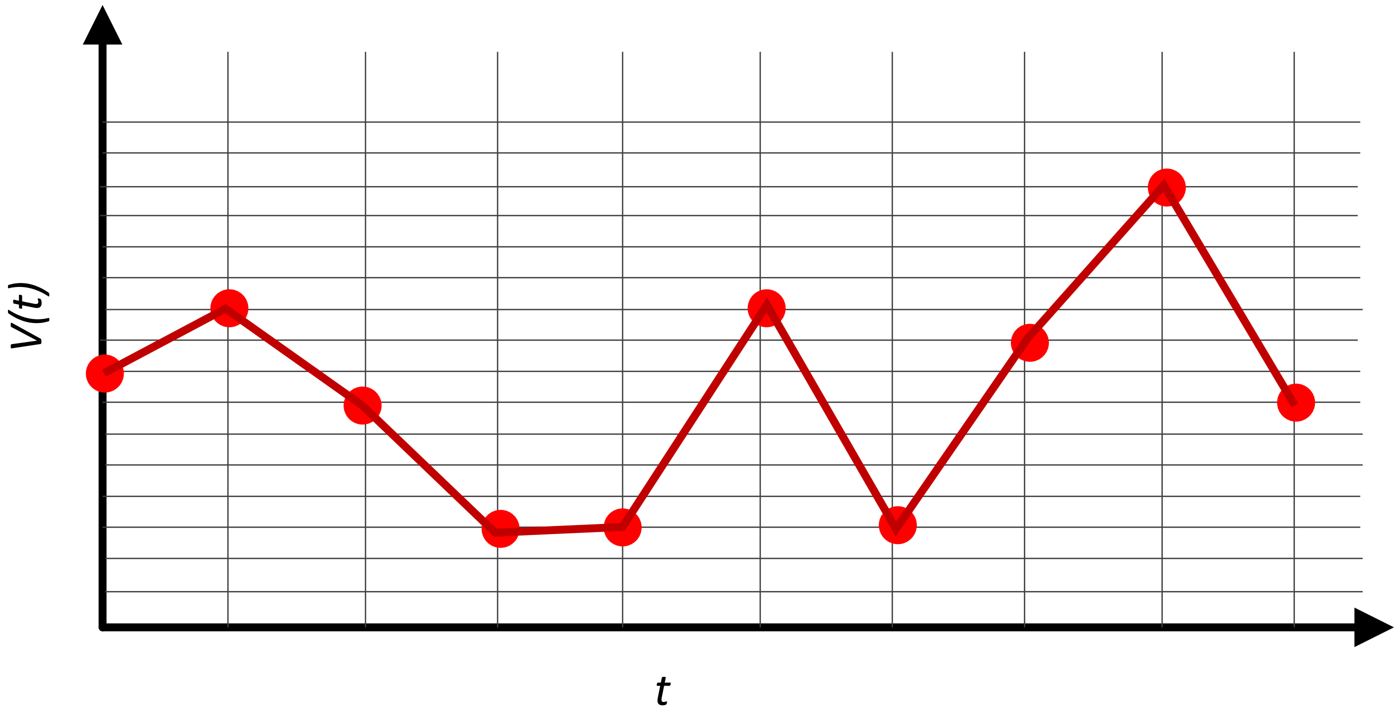
Sample it…



Store it…

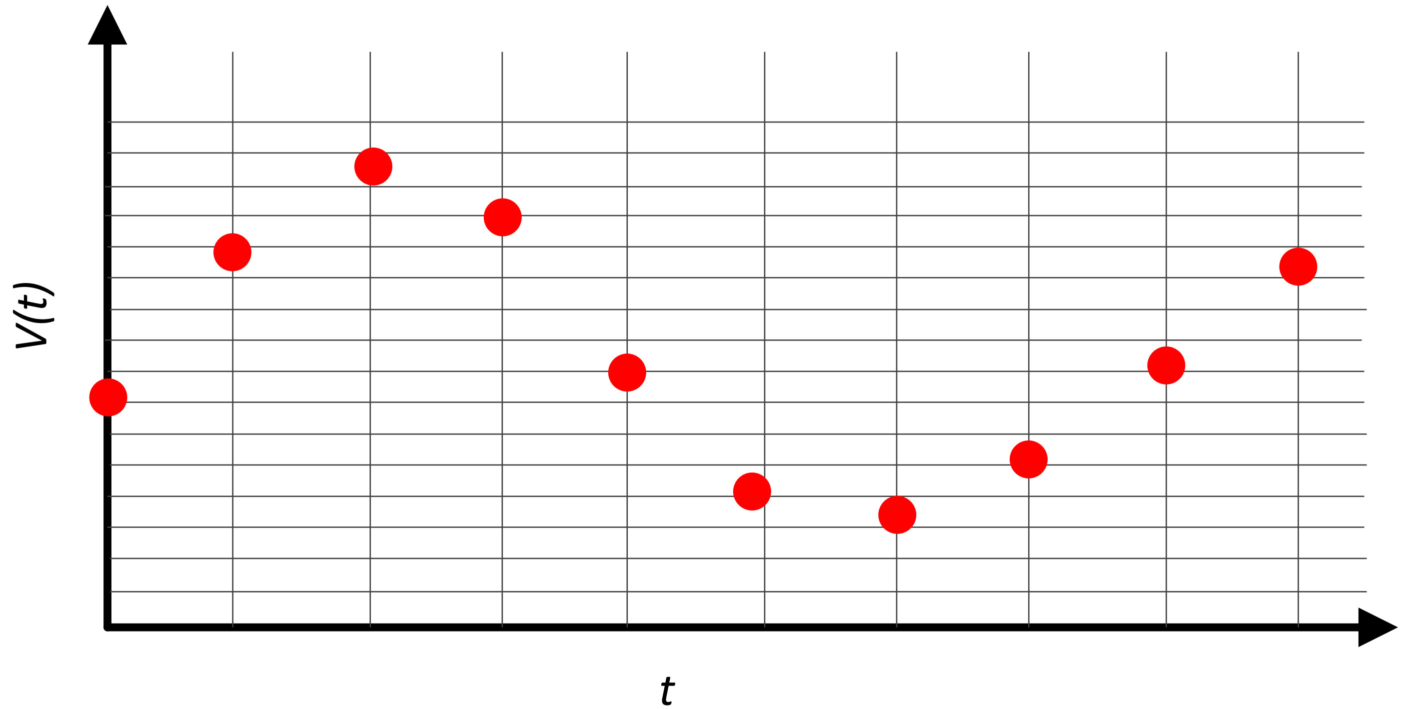


Reconstruct it…

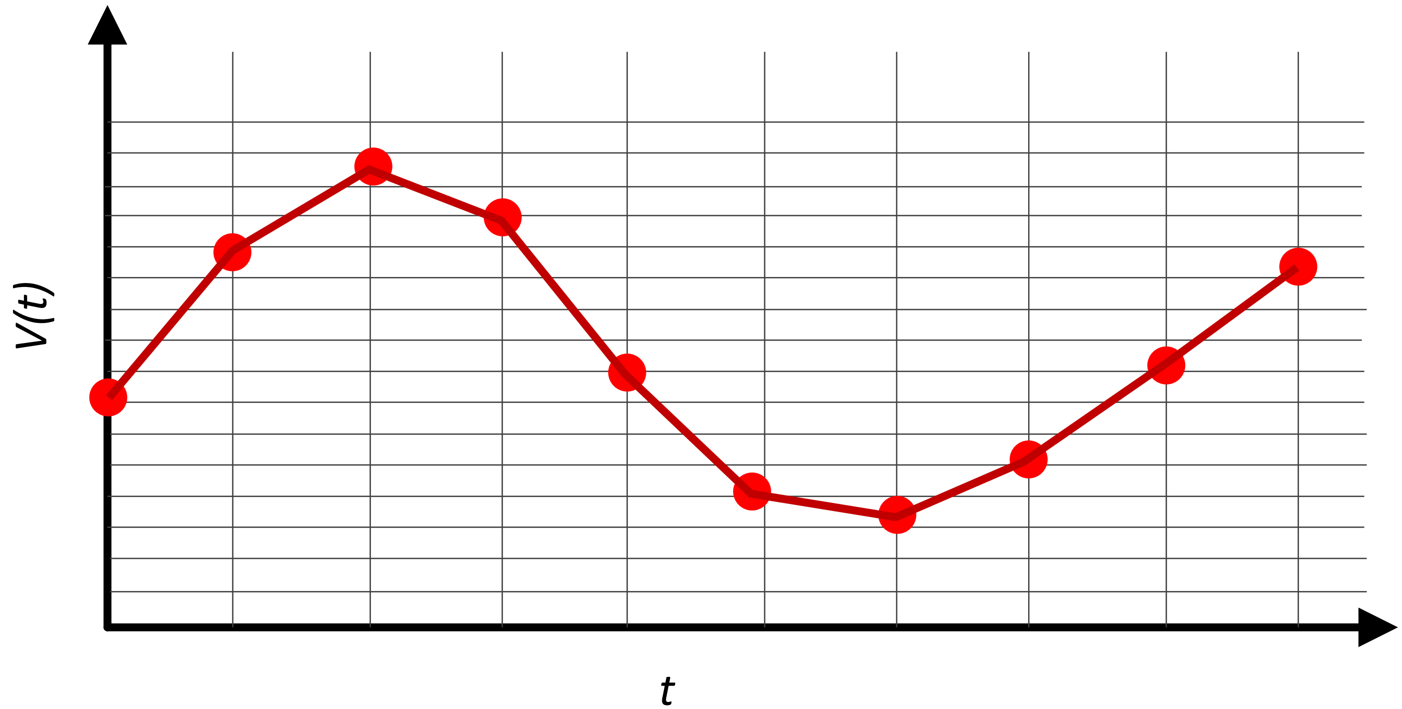


We've created a a different signal from what was before! WTH?

Or Consider this...
if we start with this data...

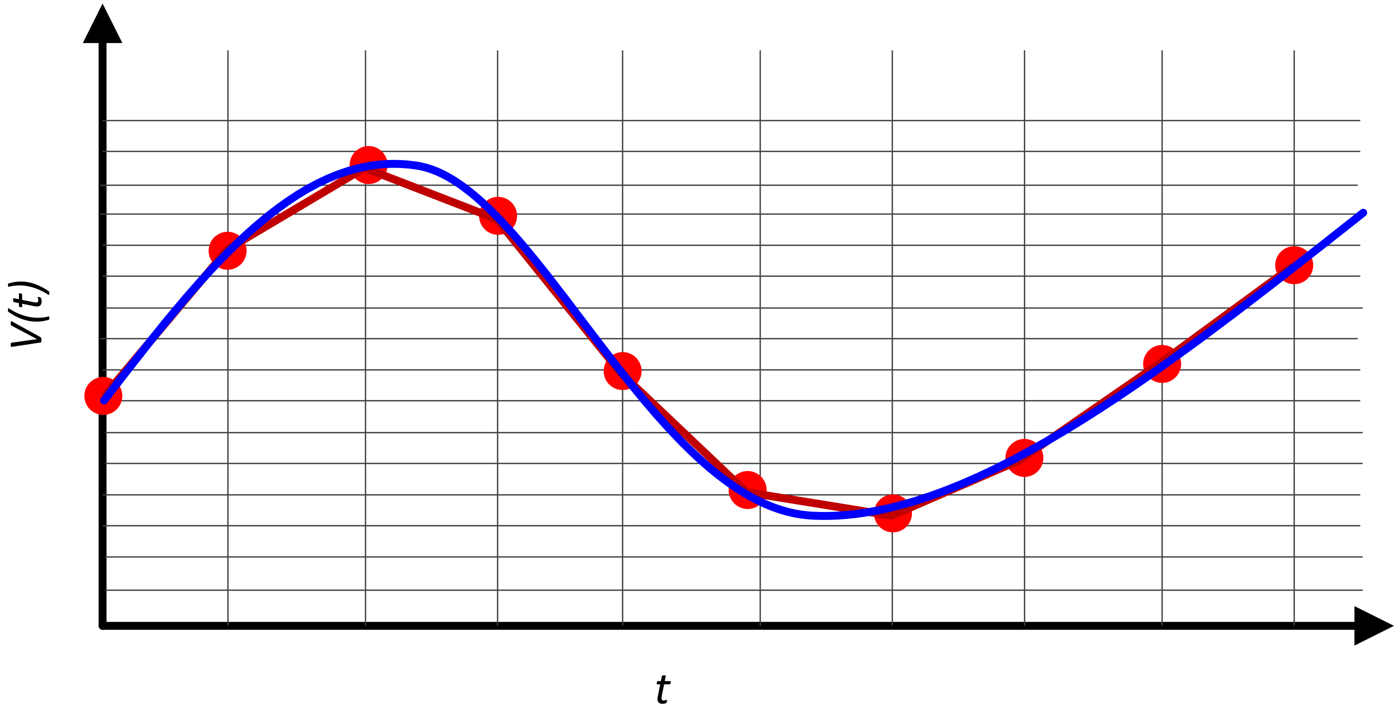


And we Reconstruct the signal…is this ok?

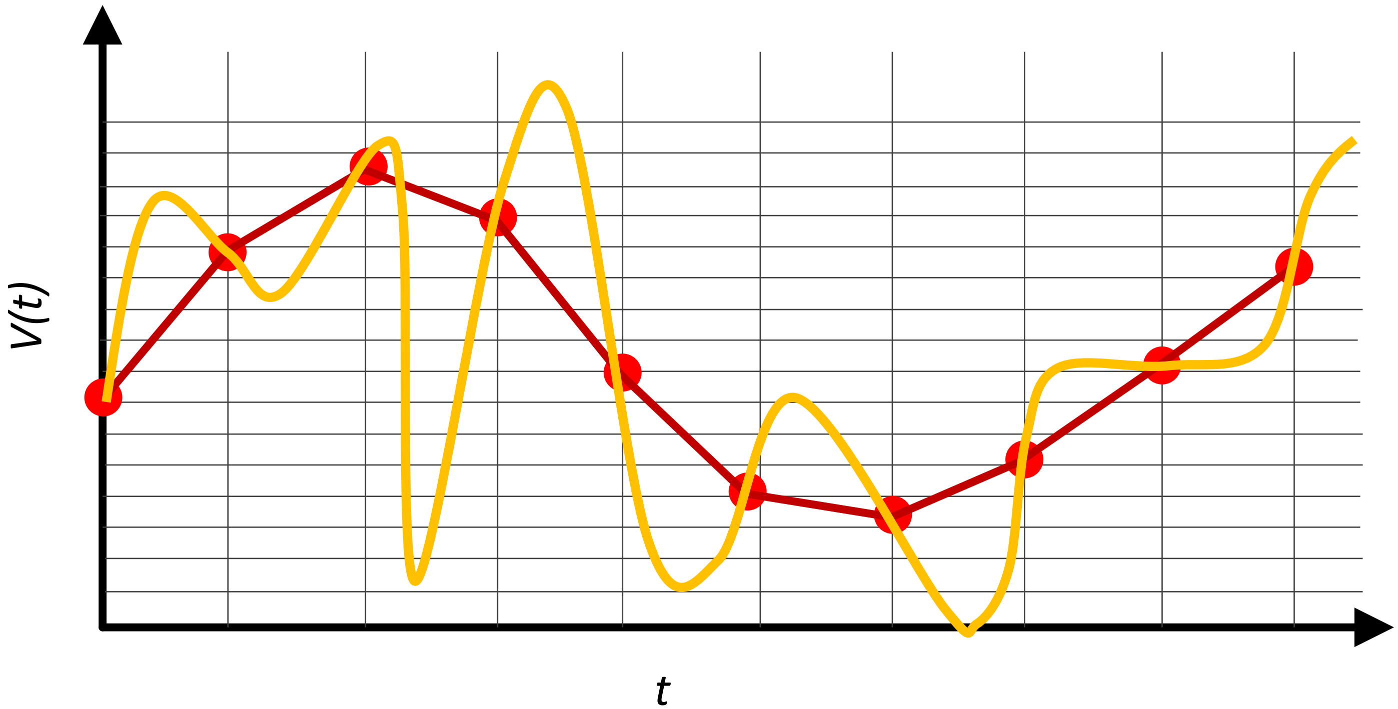


First-order hold (connect-the dots)

If it came from this, ok… but…

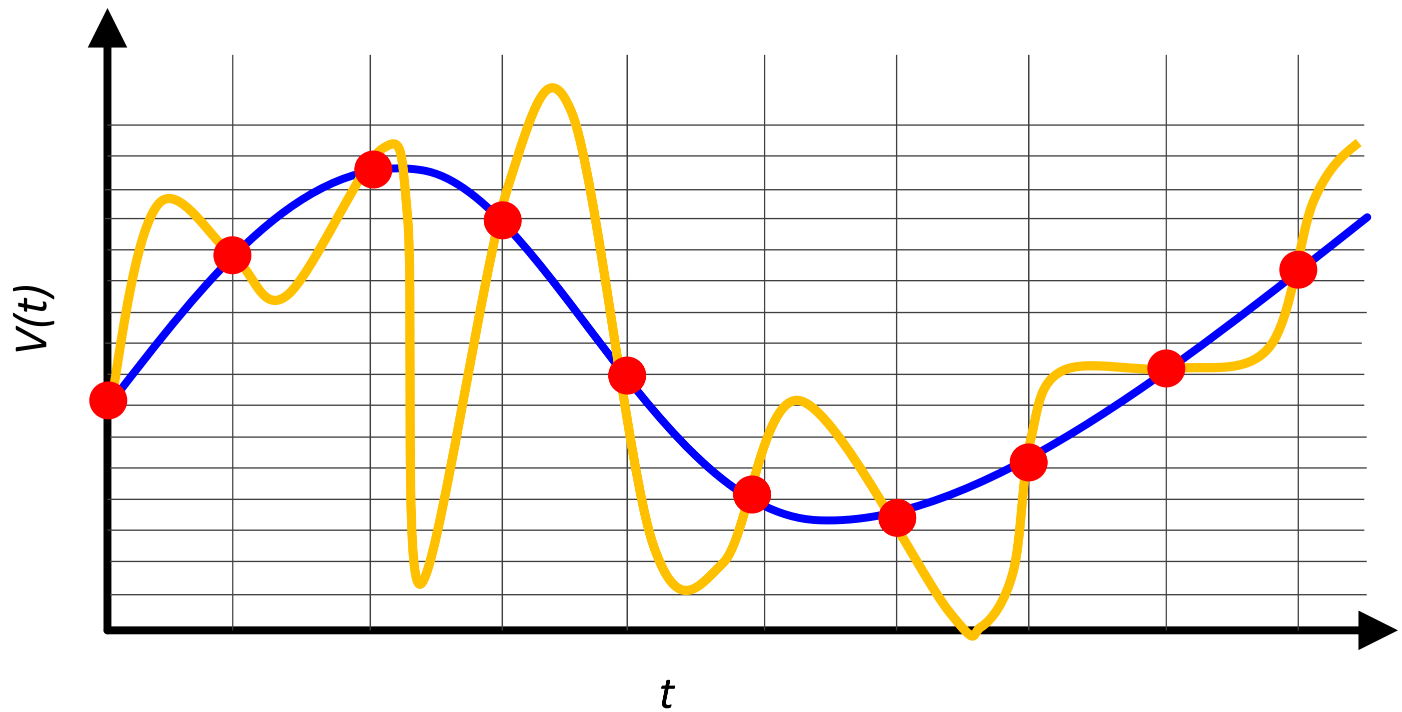


It could have also come from this...Uh oh



First-order hold (connect-the dots)

Which one Made the Signal?



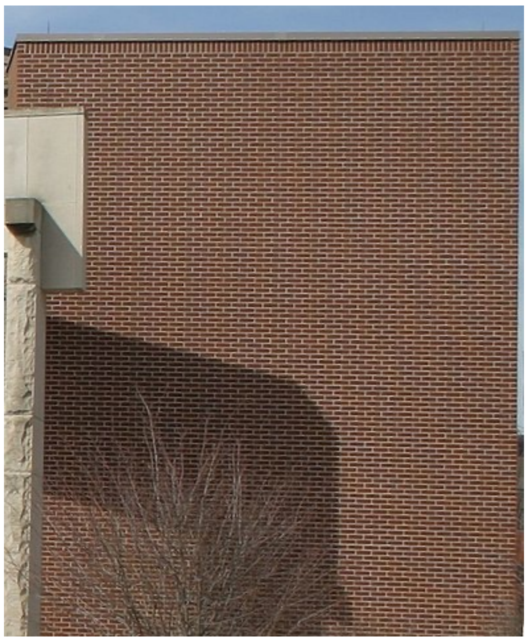
There's ambiguity in what those samples could represent…that means it really doesn't convey much, if any, information

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

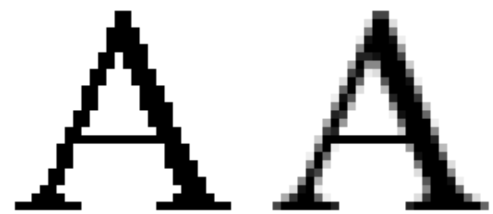
- Just like there are temporal frequencies (in time), images have spatial frequencies.
- Same issues arise!



Anti-alias Filtered

Not Anti-alias Filtered

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



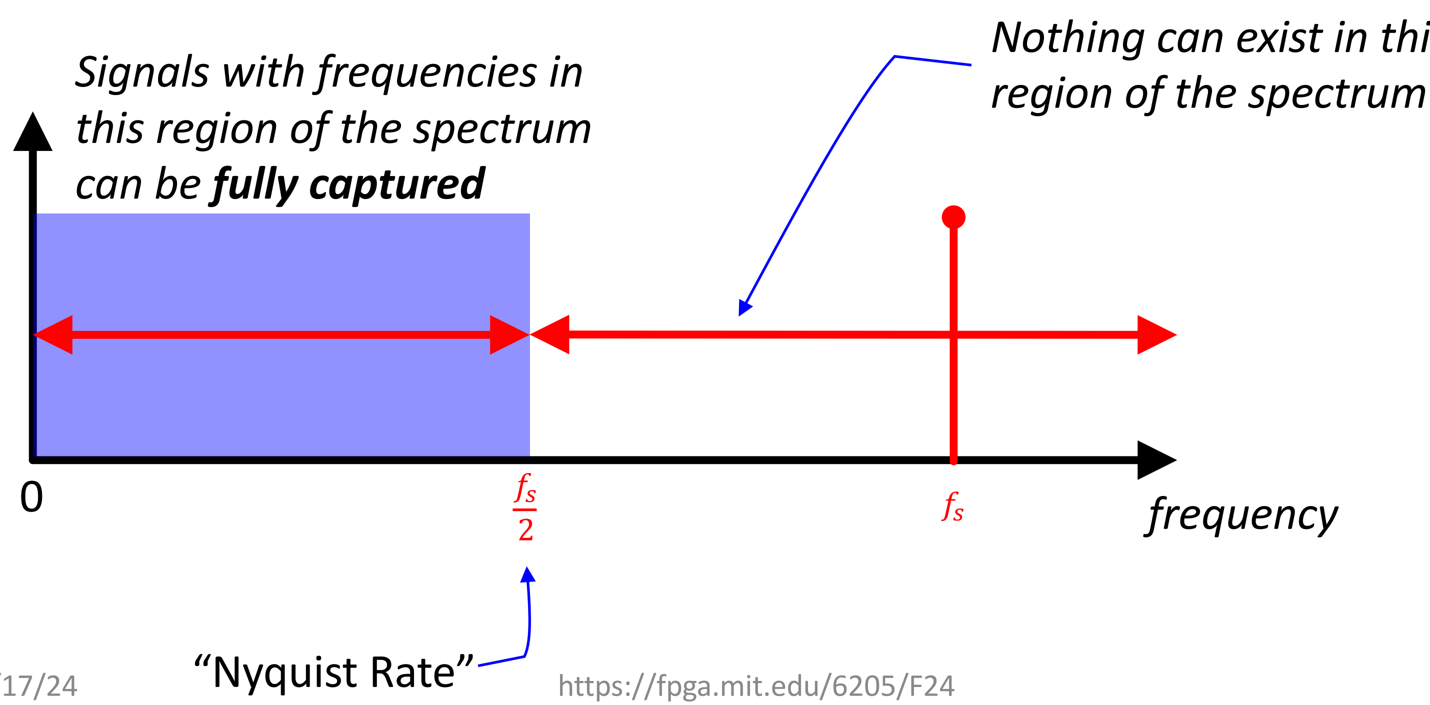
This font has been processed with an anti-alias filter to prevent artifacts when displayed

Aliasing in Audio

https://www.youtube.com/watch?v=UaKho805vCE&ab_channel=MarkAndersonAudio

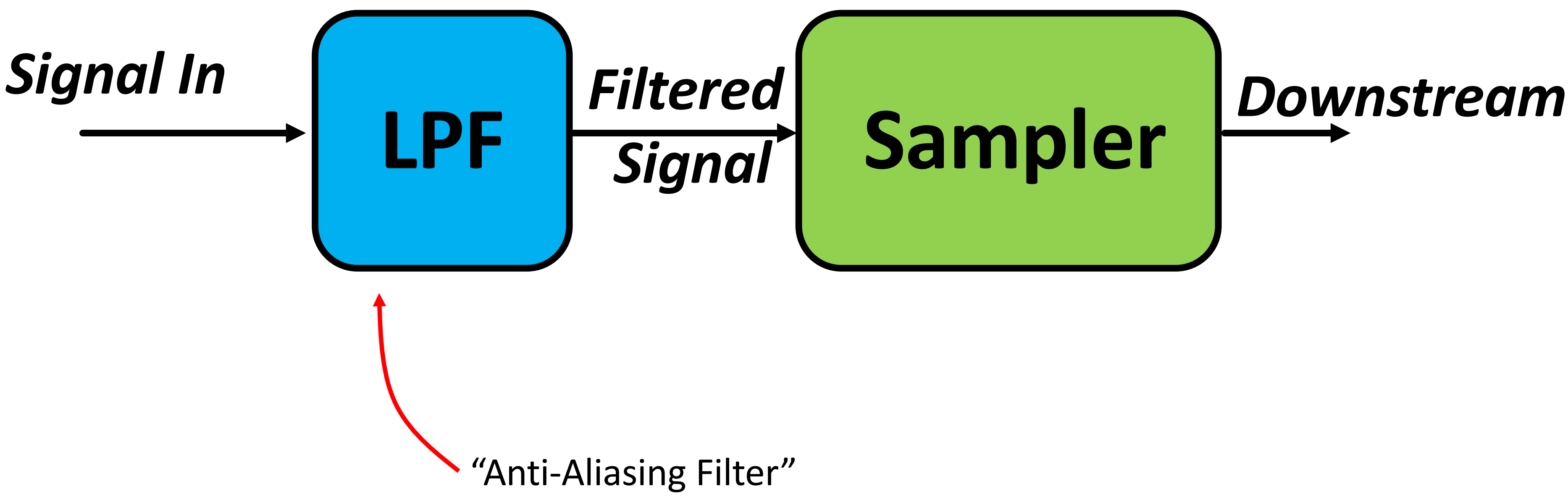
Solution

- 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!



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?

- No time for math…6.003, more so 6.341 spend their time on this stuff*
- 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

*and it is cool stuff!

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

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)

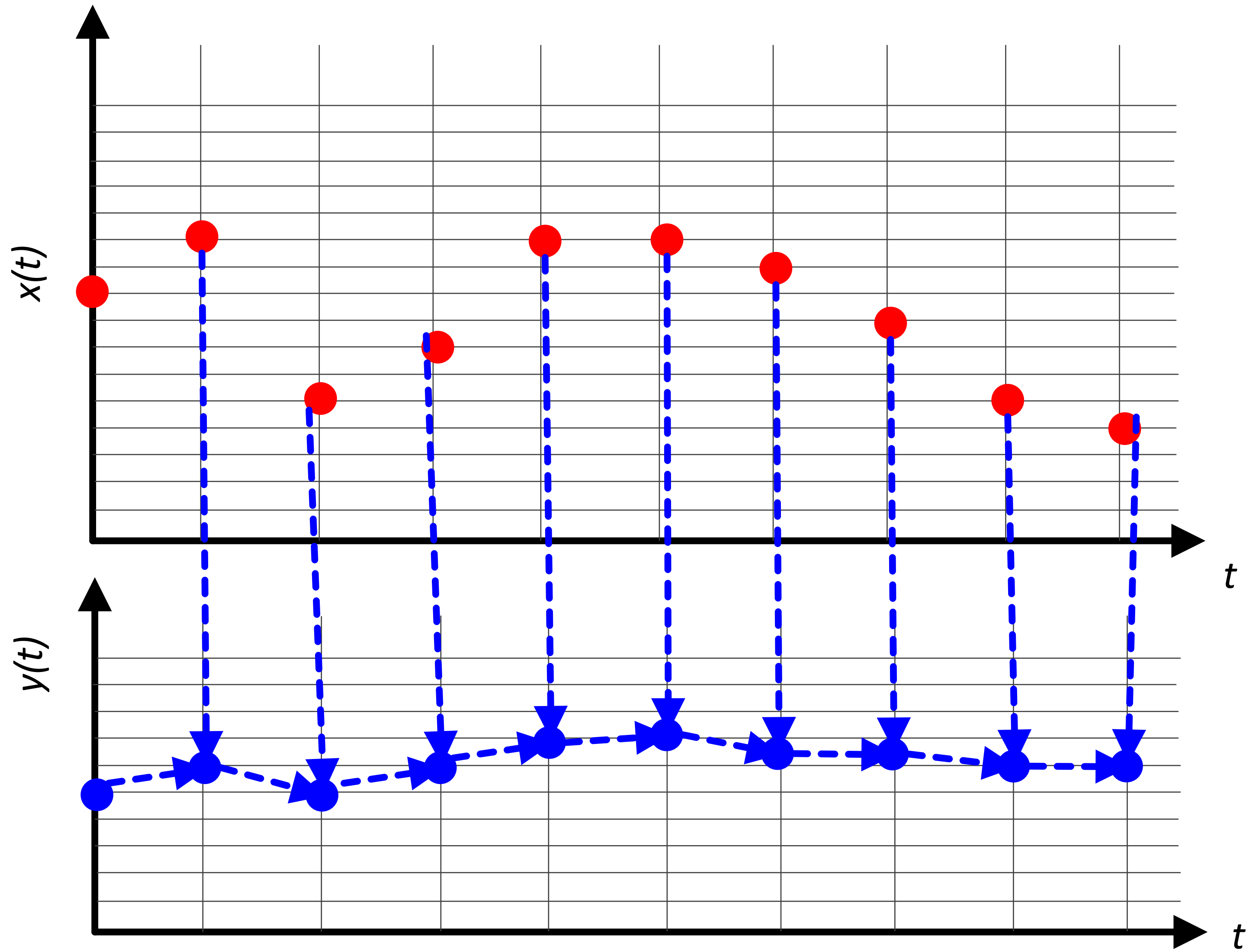
$$y[n] = \alpha \cdot y[n-1] + (1-\alpha) \cdot x[n]$$

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

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

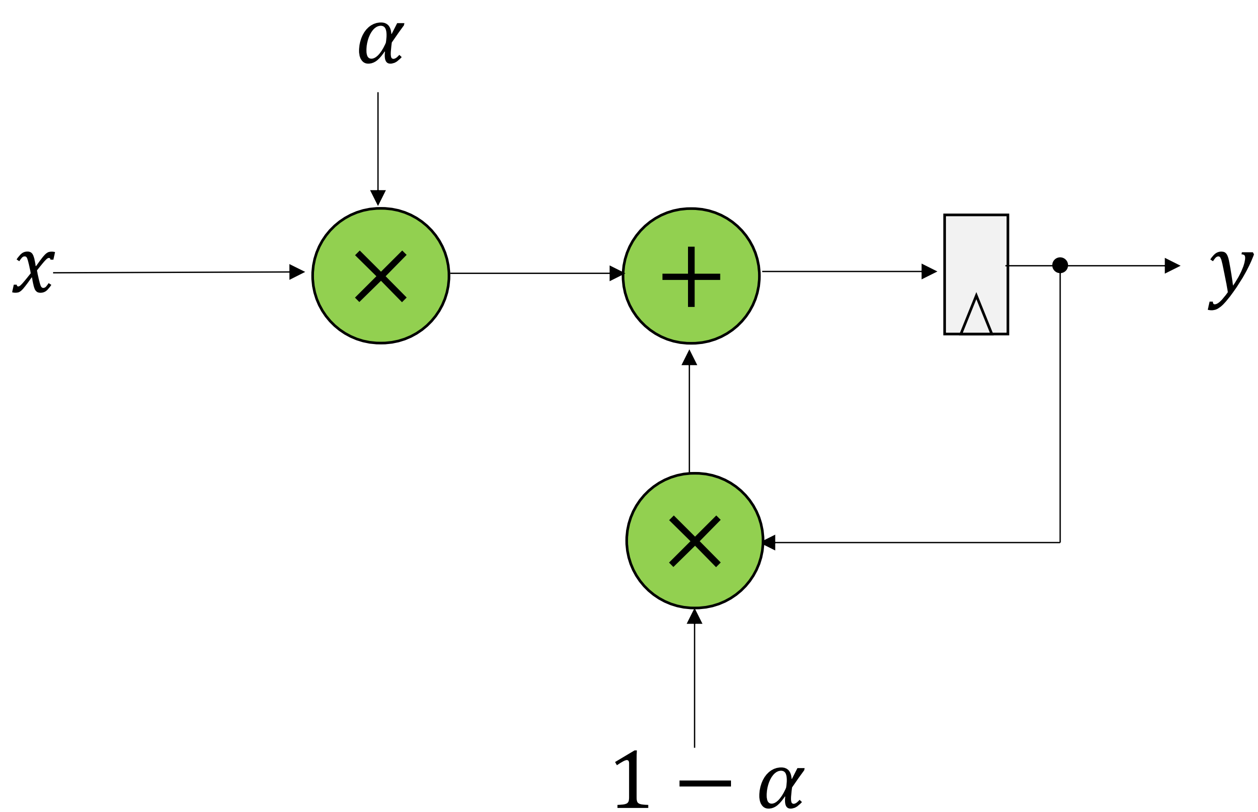
IIR Filter

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



Infinite Impulse Response (Modified)

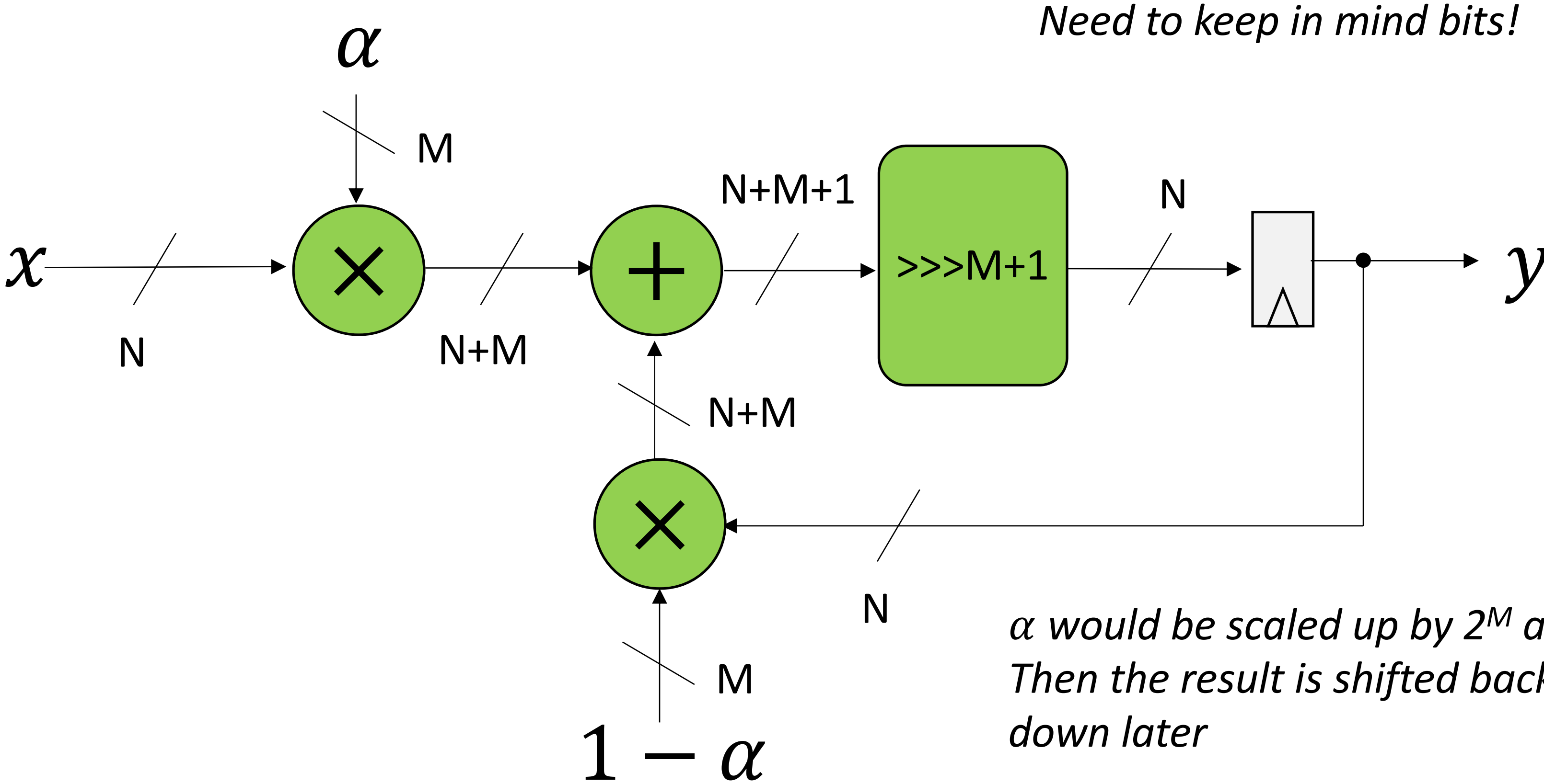
$$y[n] = \alpha \cdot y[n-1] + (1-\alpha) \cdot x[n] \qquad 0 \le \alpha \le 1$$



Infinite Impulse Response (Modified)

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

Need to keep in mind bits!



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

Finite Impulse Response

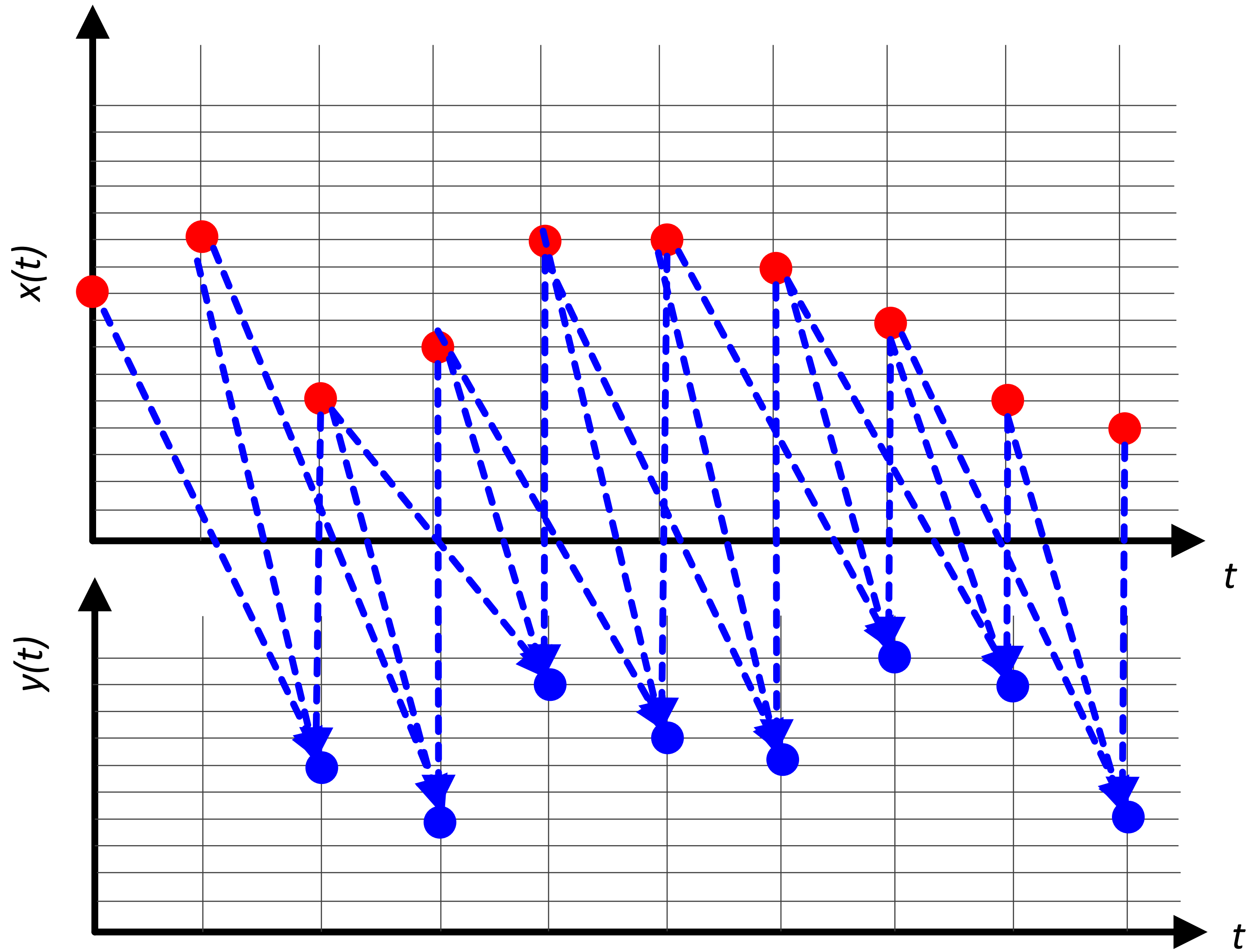
- Have the output be based off of a sliding window of the past history of the input.
- Literally just convolution basically

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

- 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 1000s 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 6.003/6.341, etc)

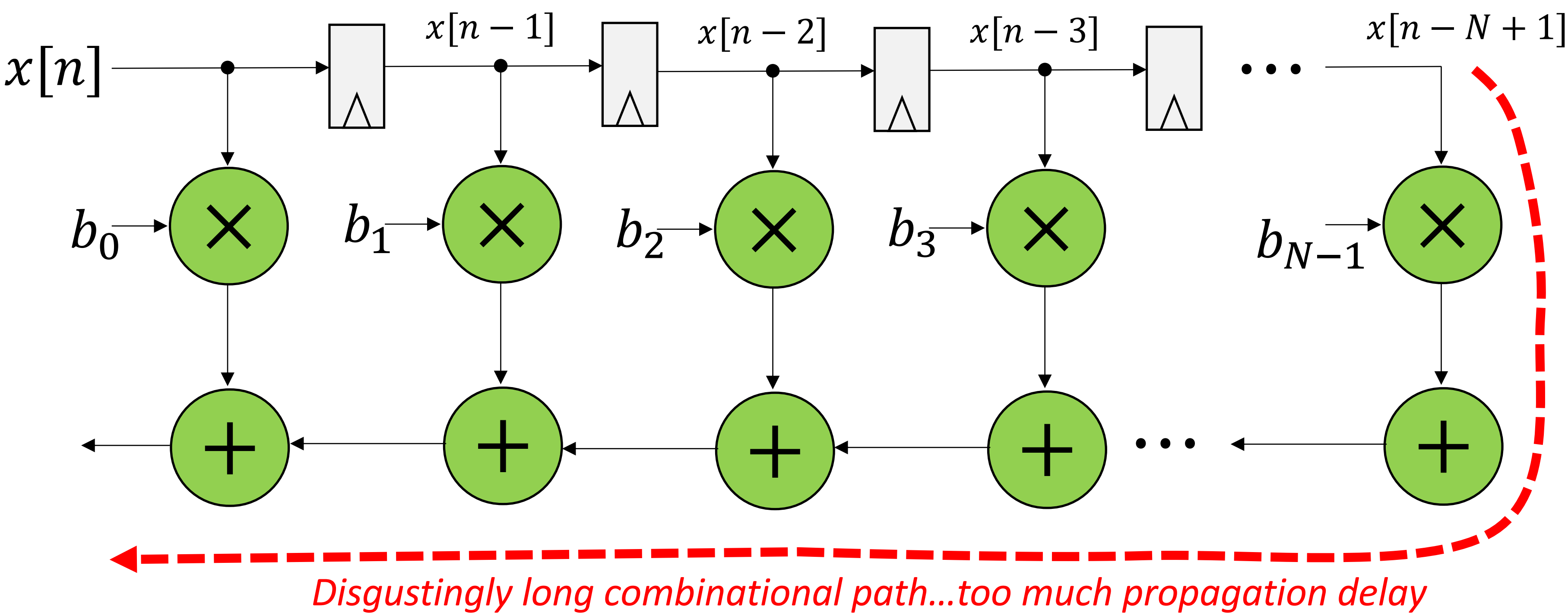
FIR Filters

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

- 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!

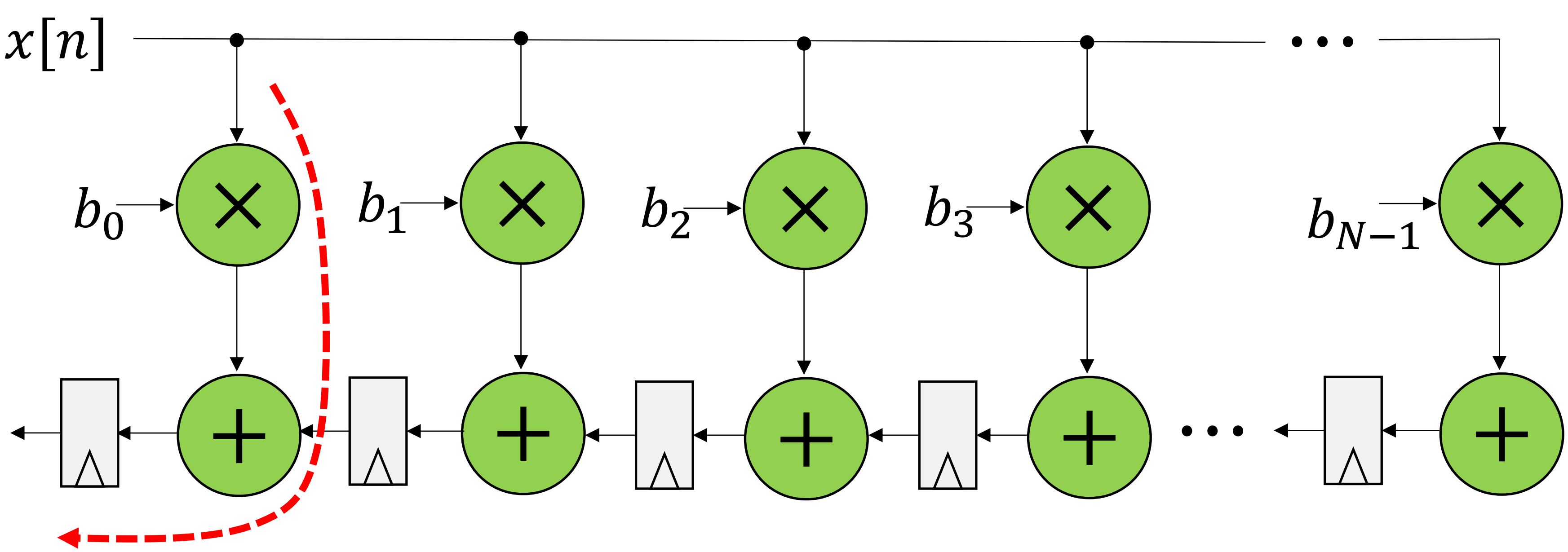
Finite Impulse Response

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



Finite Impulse Response (Modified)

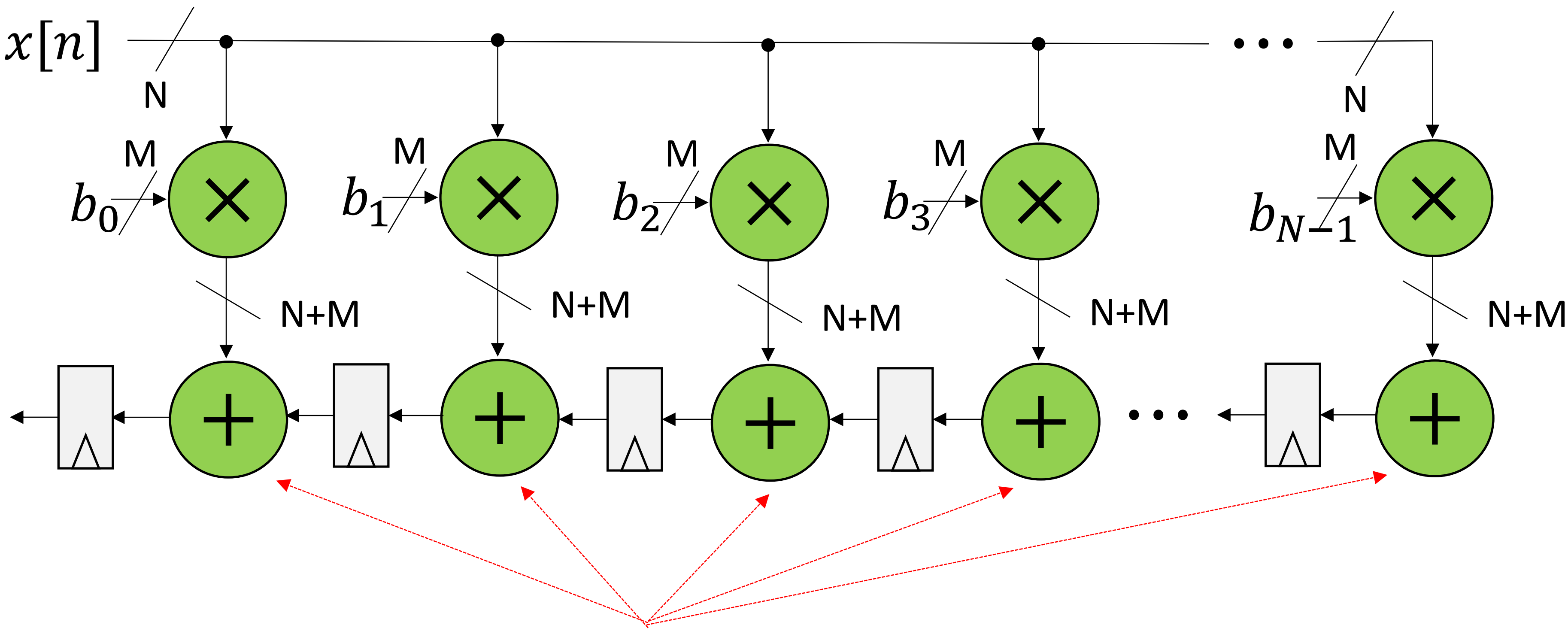
$$y[n] = \sum_{k=0}^{N-1} b_k \cdot x[n-k]$$



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

DSP Blocks?

- These IIR and especially FIR filters sure do have a lot of multiply-then-add operations going on…
- Remember those DSP blocks? That's why they're designed the way they are

DSP Blocks

- Mult-then-add is a common operation chain in many things, particularly Digital Signal Processing
- FPGA has dedicated hardware modules called DSP48 blocks on it
  - 150 of them on Urbana FPGA board
  - 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 tiing accordingly

DSP48 Slice (High Level)

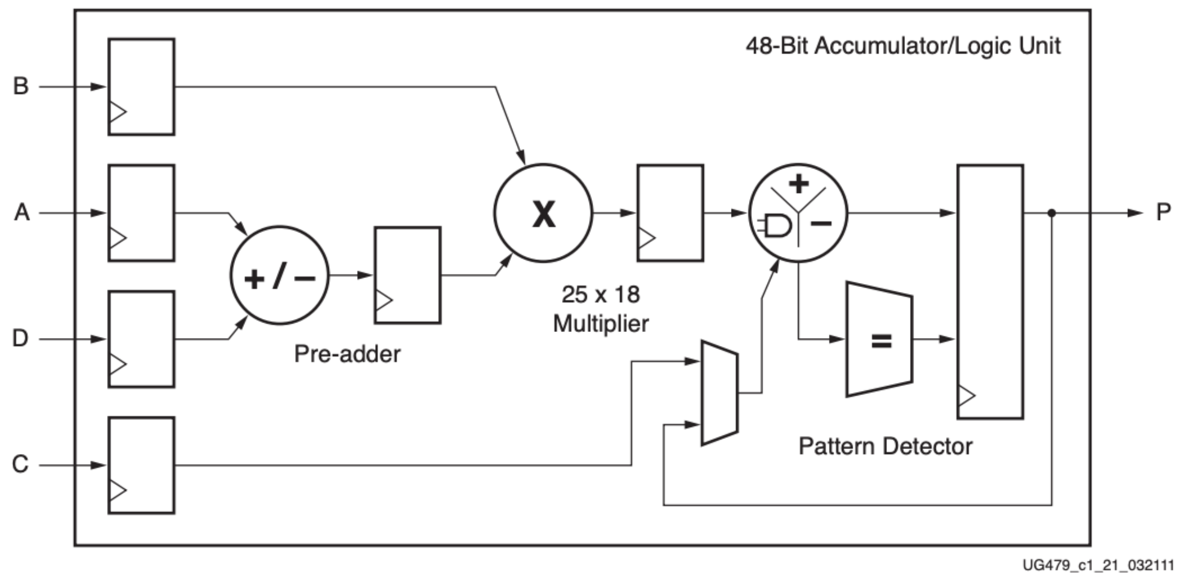


Figure 1-1: **Basic DSP48E1 Slice Functionality**

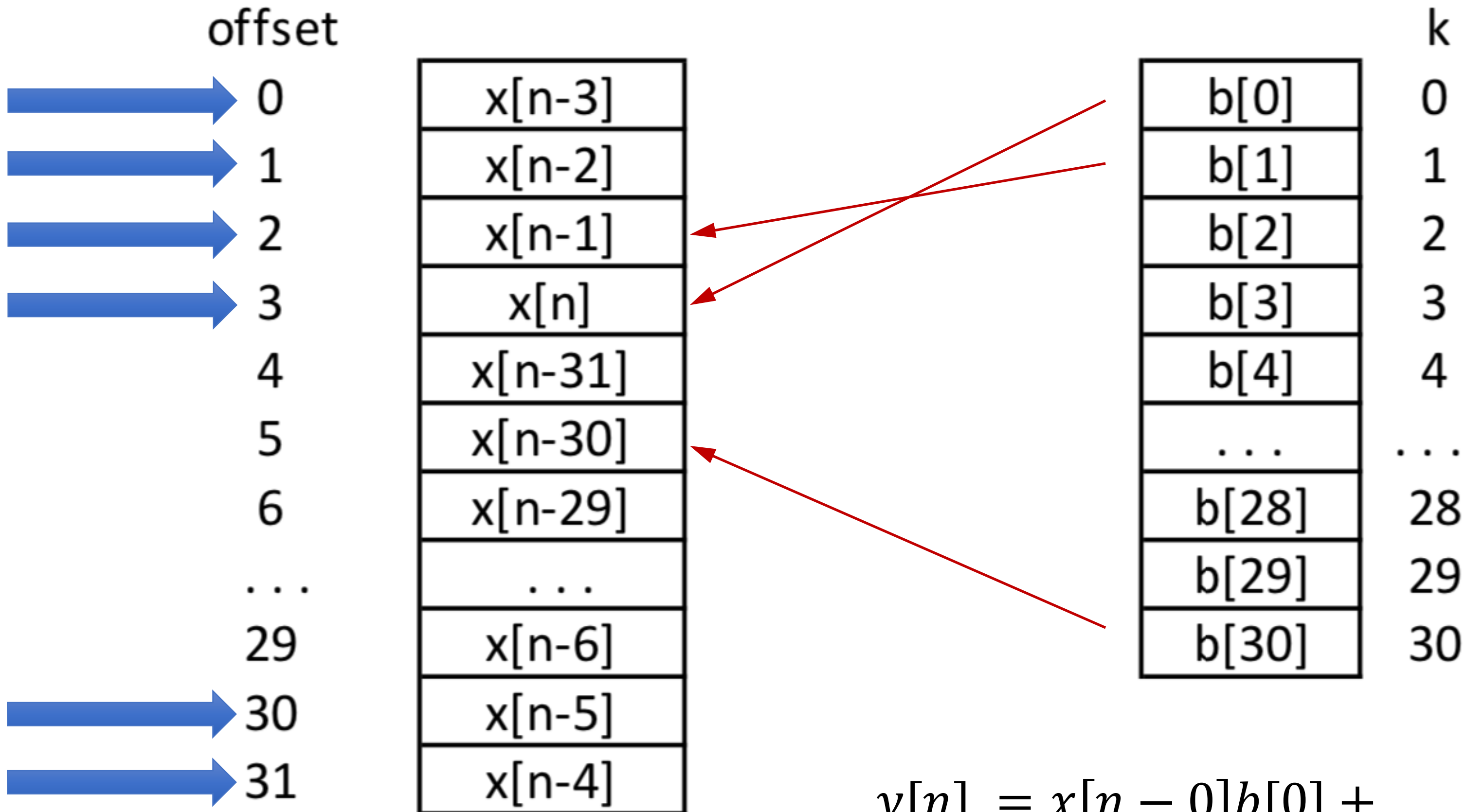
https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

FIR Filter (Iterative Design)

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

- 1000's of taps will use way too much resources. Instead you can also build FSM-based FIR filters
  - Be given new input sample
  - Use one clock-cycle per multiply-add
  - Accumulate the sum
  - After N cycles, your output is calculated
  - Update a circular buffer to keep track of past values of x
- For audio usually plenty of clock cycles between each audio cycle anyways (you have 2000 clock cycles of 100 MHz between each audio sample of 48 ksps audio!)

Circular Buffer/Pointer in Action

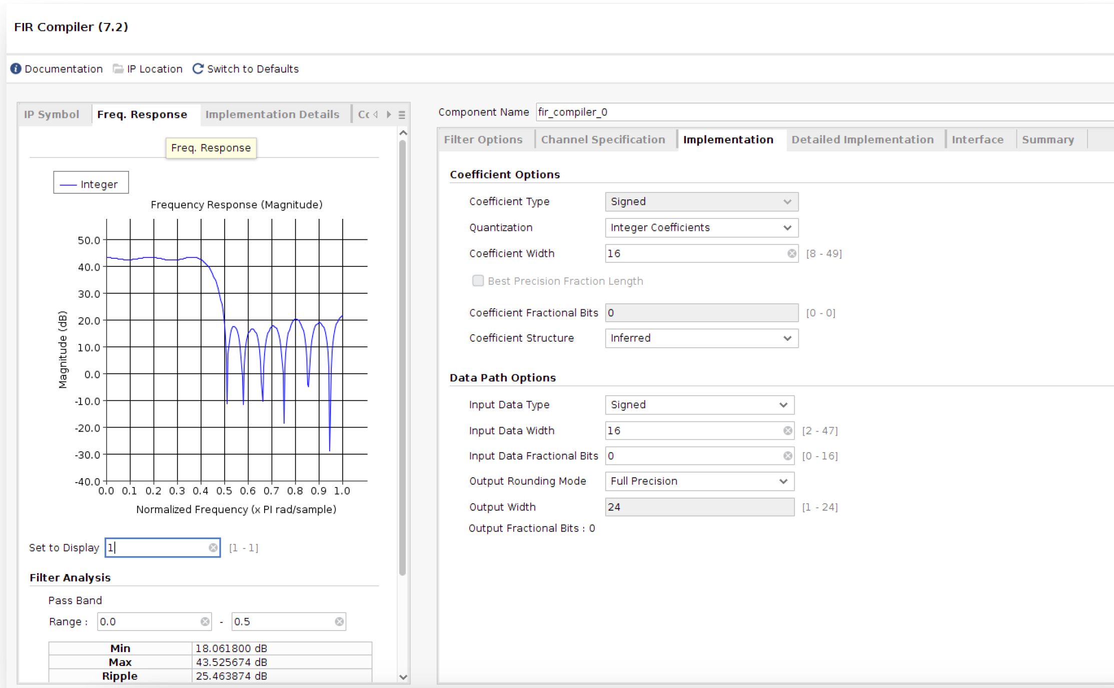


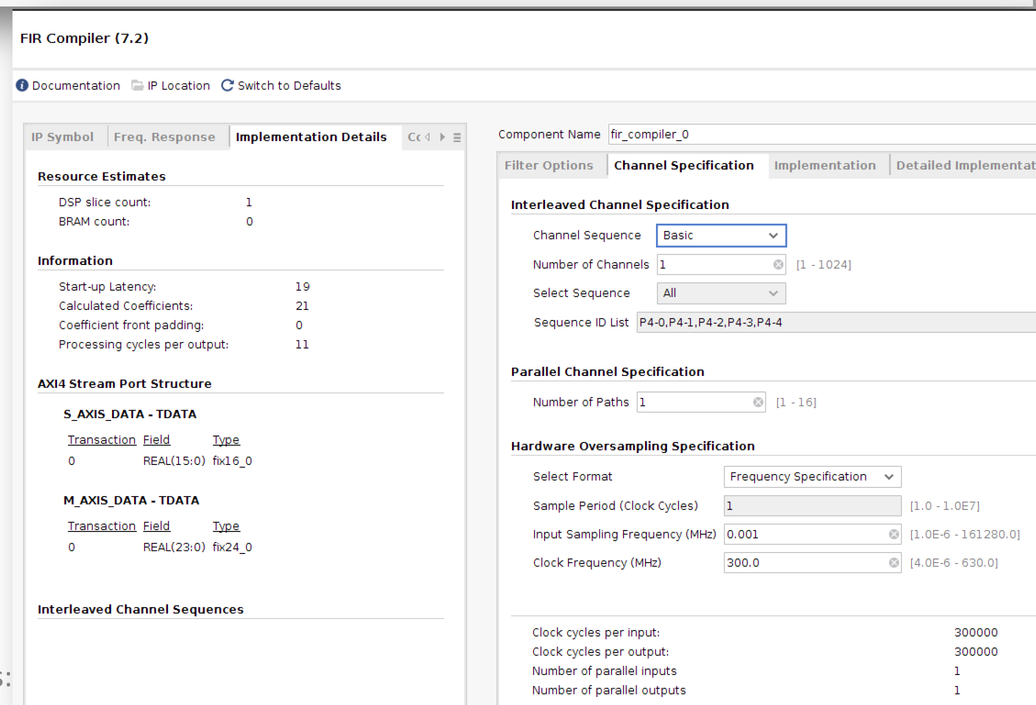
$$y[n] = x[n-0]b[0] + x[n-1]b[1] + \ldots x[n-30]b[30] +$$

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

FIR Wizard

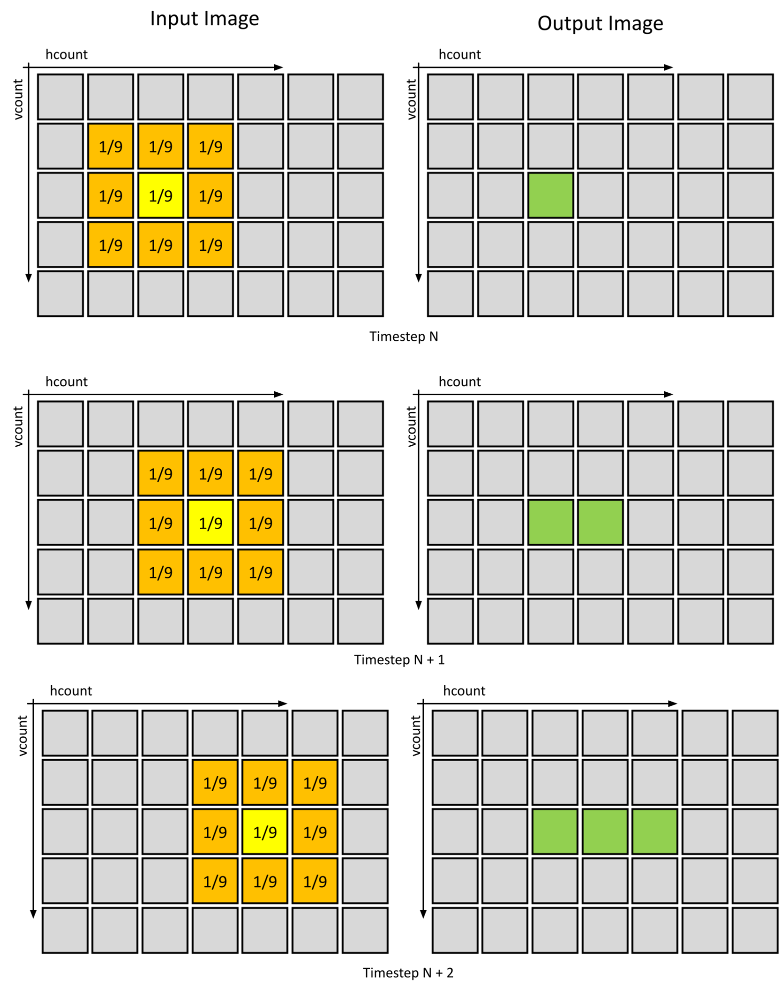
- FIRs are so common, Vivado actually has some IP infrastructure to aid in designing them
- Can tune how pipelined vs. Iterative/FSM you want your FIR!
- Or use Python/numpy to determine coefficients



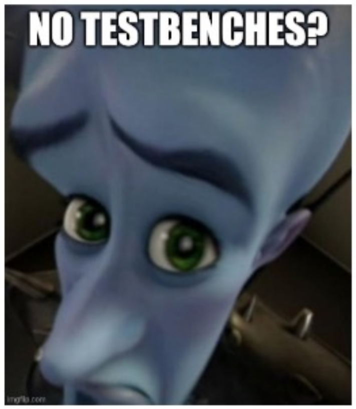


In 2D Space you can also make filters (week 7)

- The common way is a 2D FIR filter, except it exists in 2 dimensions
- Shown here is a 3x3 filter
- The weights of the coefficients make up the "kernel"
- It gets dragged/convolved across the screen

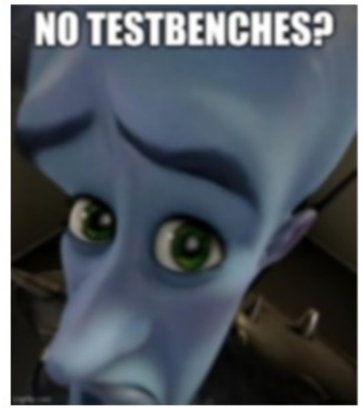


In 2D Space you can also make filters (week 7)



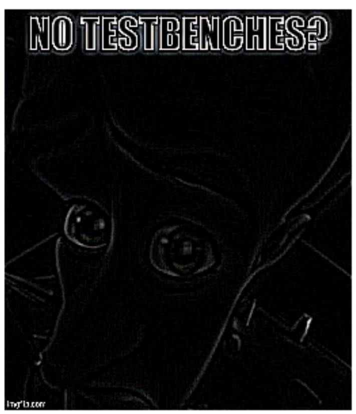
Original Image

Identity Kernel



Gaussian Blur

Sharpen



Ridge Detect

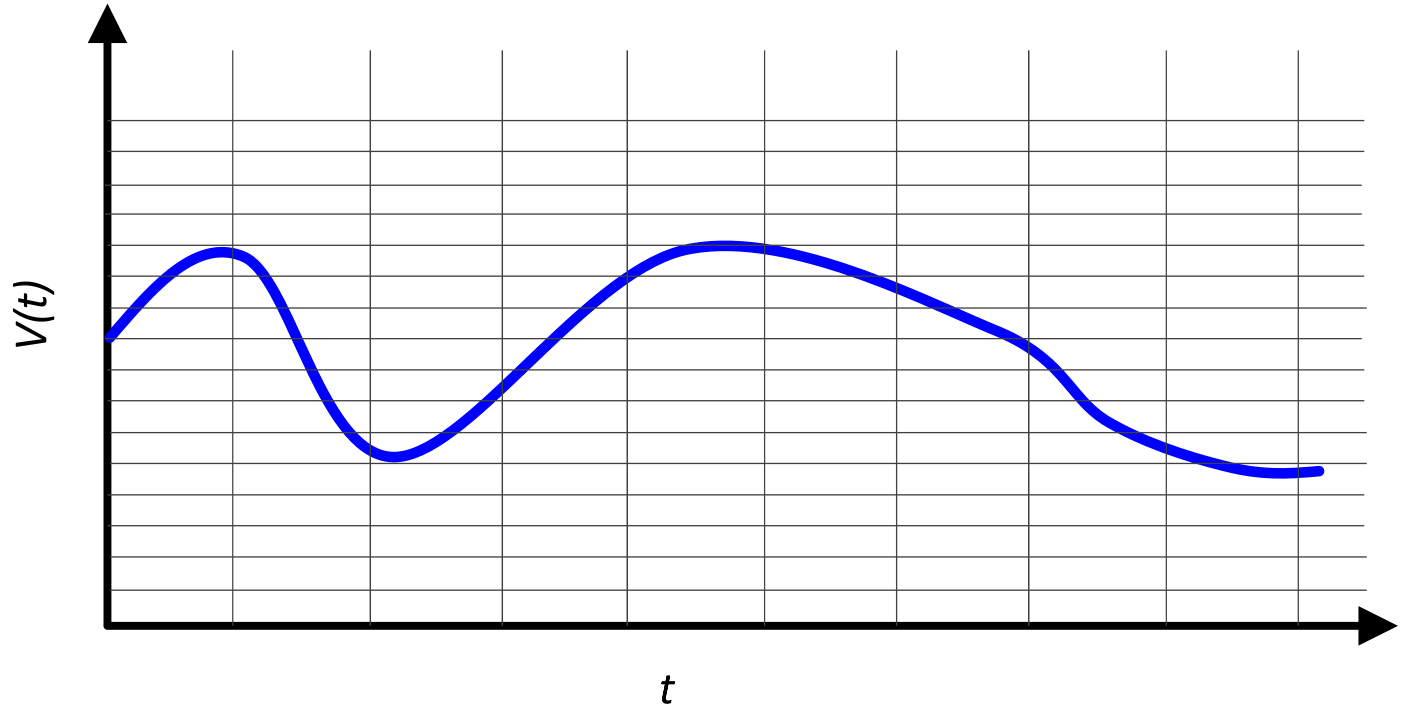

X Sobel Edge Detect


Y Sobel Edge Detect


X Sobel + Y Sobel

Quantization

Discretization in Time and **Quantization** in Value

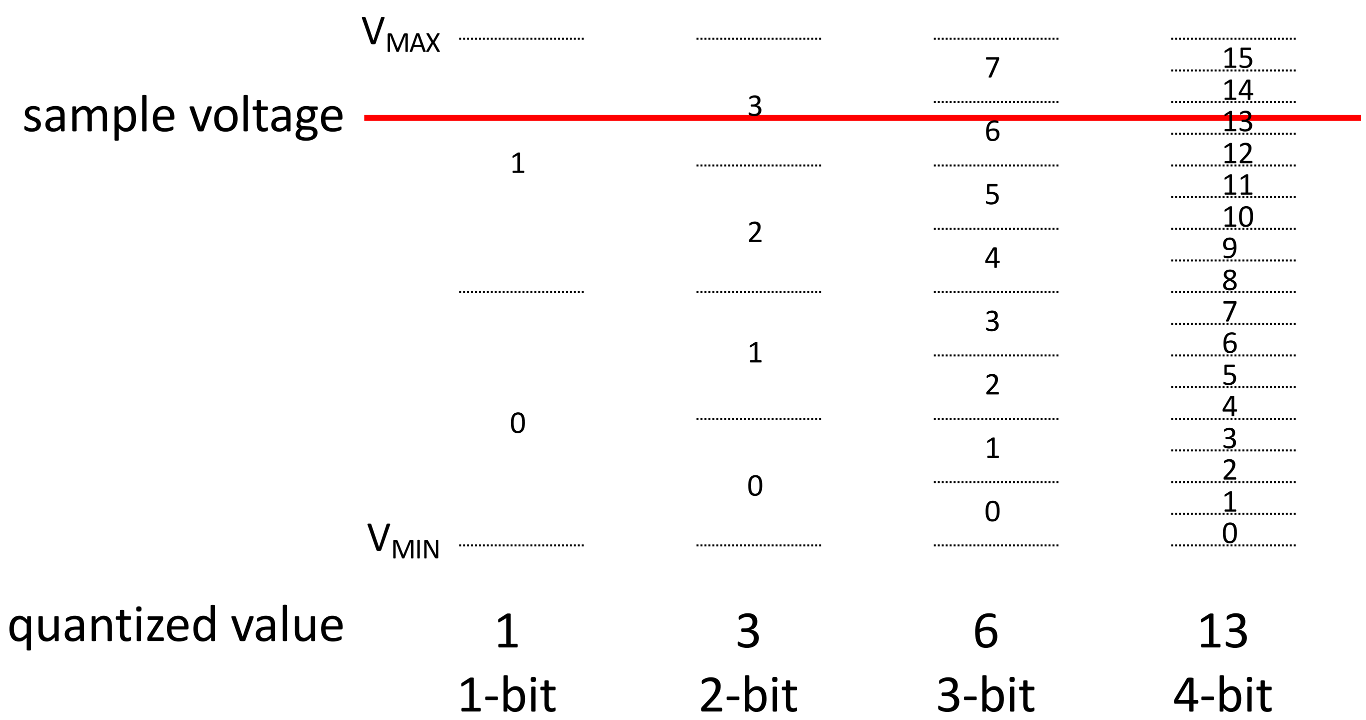


4 bit value encoding

Quantized Values

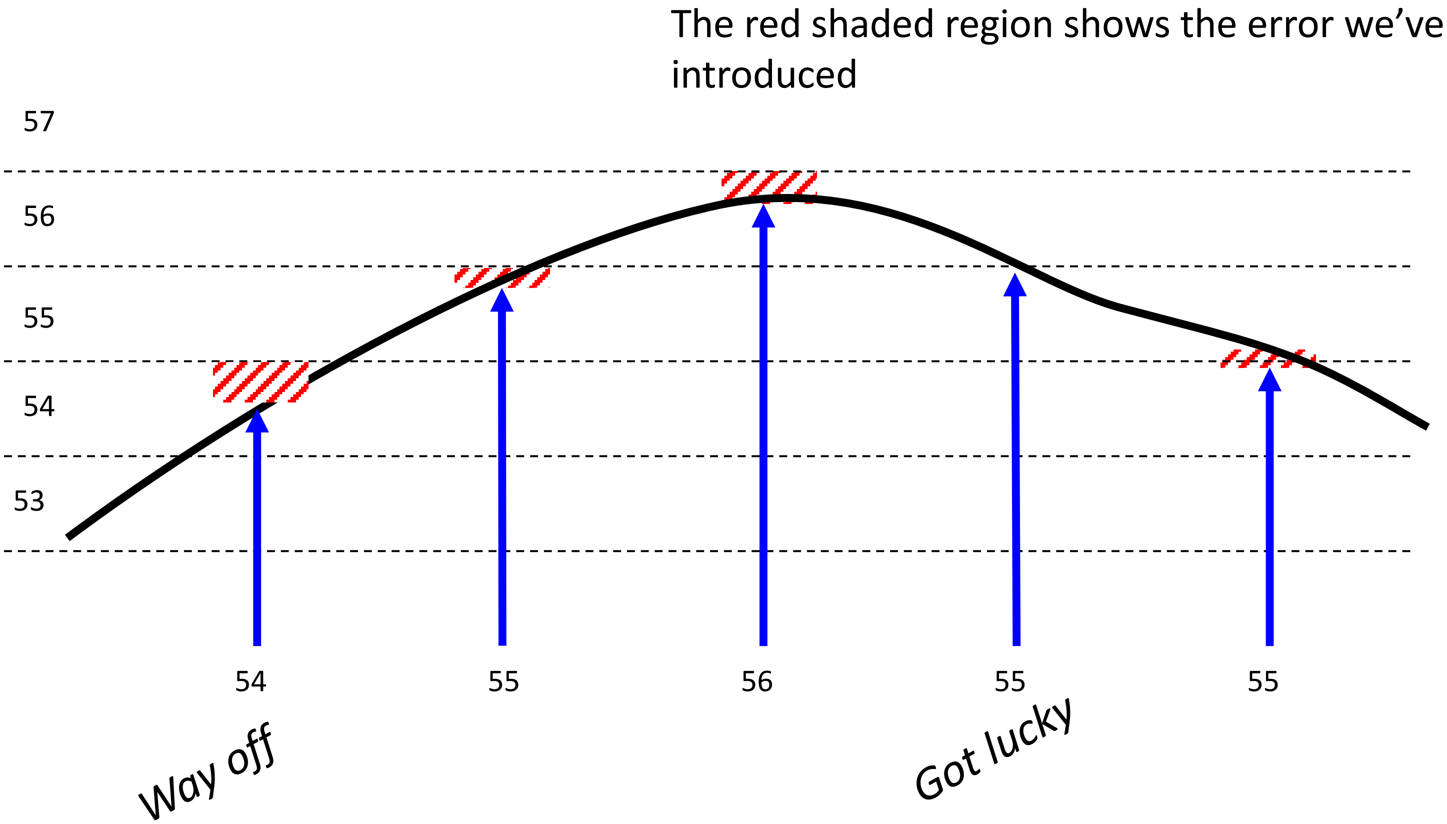
If we use N bits to encode the magnitude of one of the discrete-time samples, we can capture 2^N possible values.

So we'll divide up the range of possible sample values into 2^N intervals and choose the index of the enclosing interval as the encoding for the sample value.

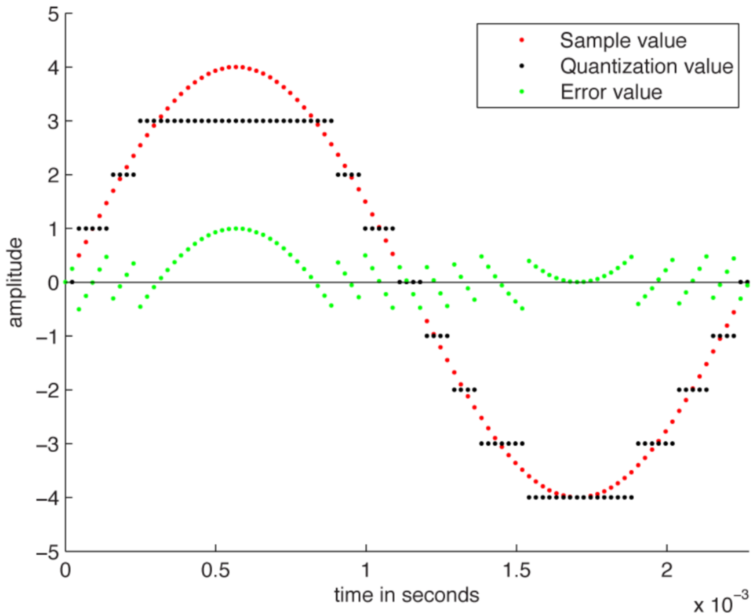


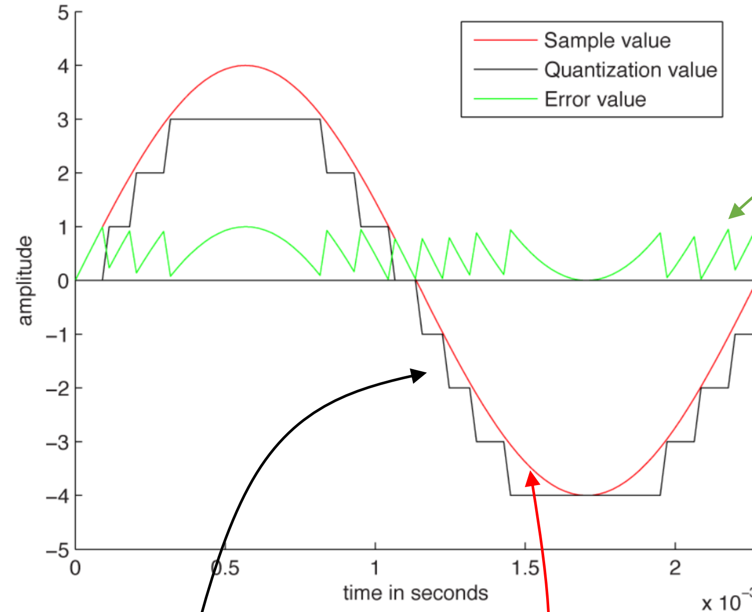
Quantization Error

Note that when we quantize the scaled sample values we may be off by up to ±½ bin from the true sampled values.



During signal reconstruction, Quantization introduces a new signal: Quantization error!



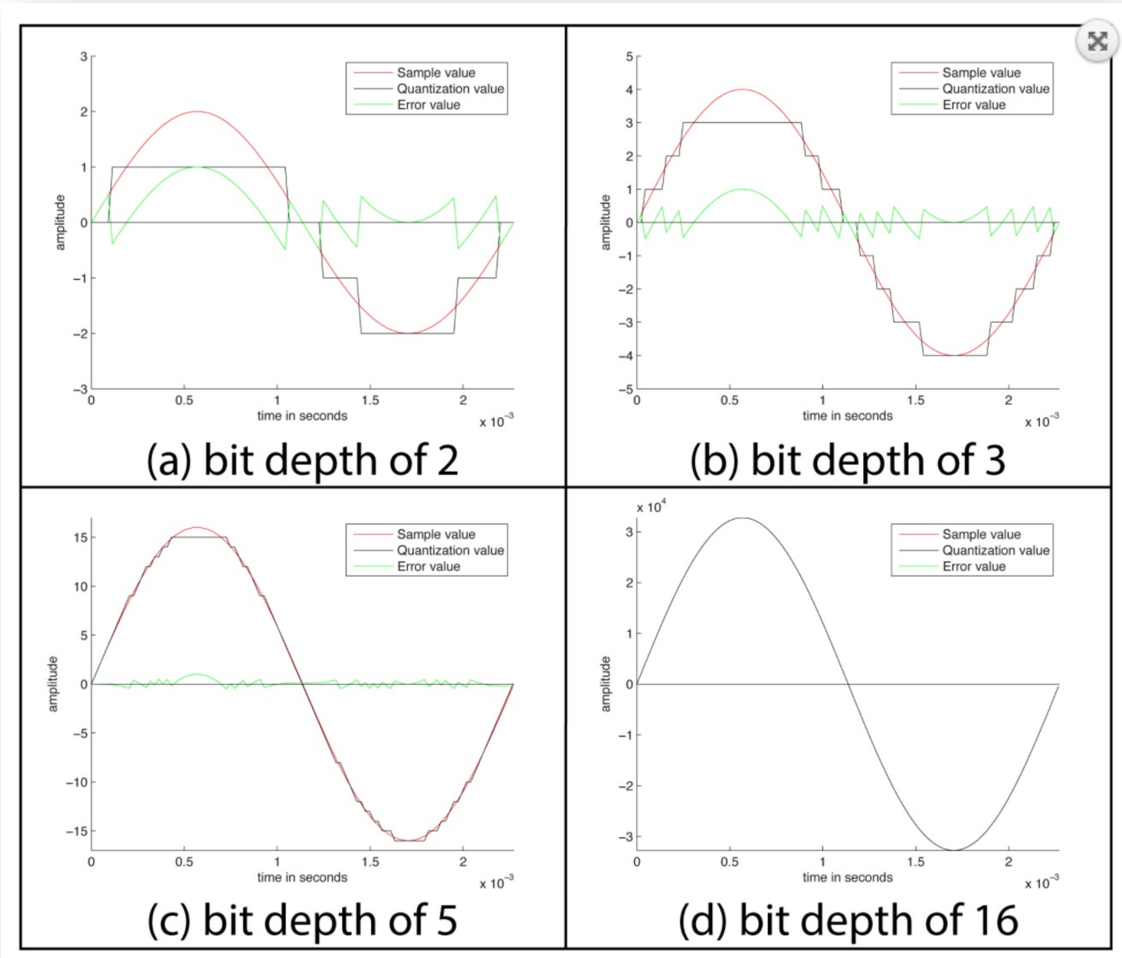


What gets reconstructed is **not** just the original signal,
But the original signal **plus** the quantization error:

$$\boldsymbol{s(t) = s_o(t) + e(t)}$$

http://digitalsoundandmusic.com/chapters/ch5/

Error Signal Drops with Higher Bit-depth

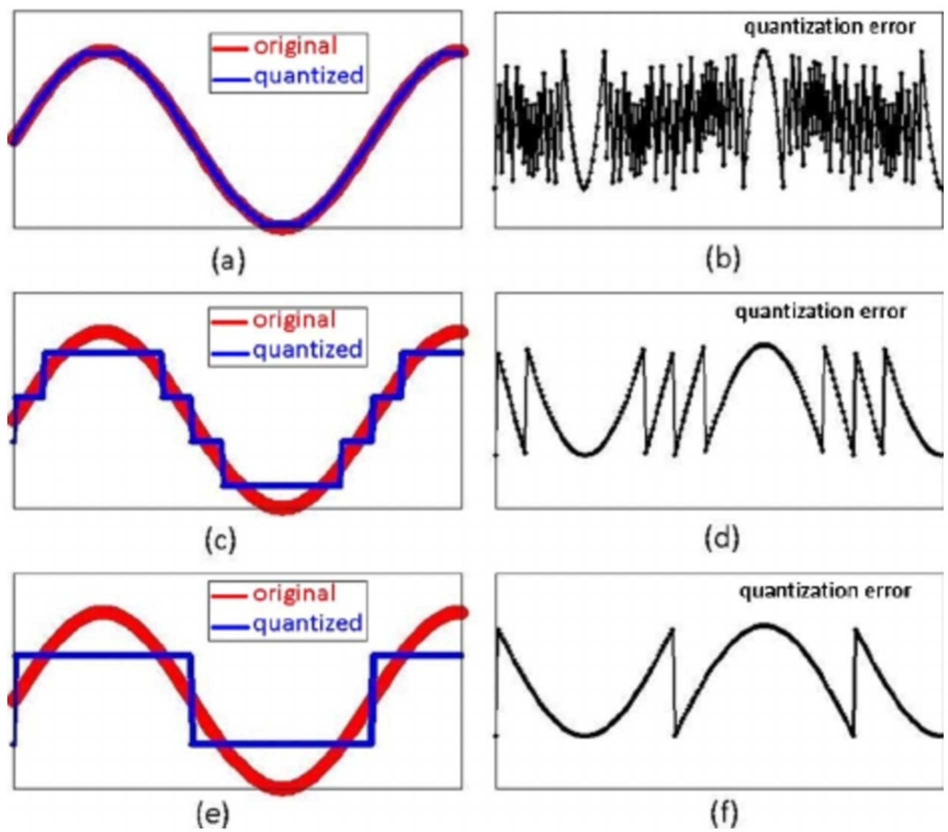


Amplitude of Error Signal Drops with higher bit depth

http://digitalsoundandmusic.com/chapters/ch5/

Structure of Quantization Noise

- The more bits we've used for quantizing:
  - The smaller our error gets
  - **AND**
  - The more "random" our error signal gets
- Fewer bits leads to error signal that actually looks like a signal :/ (NOT good)

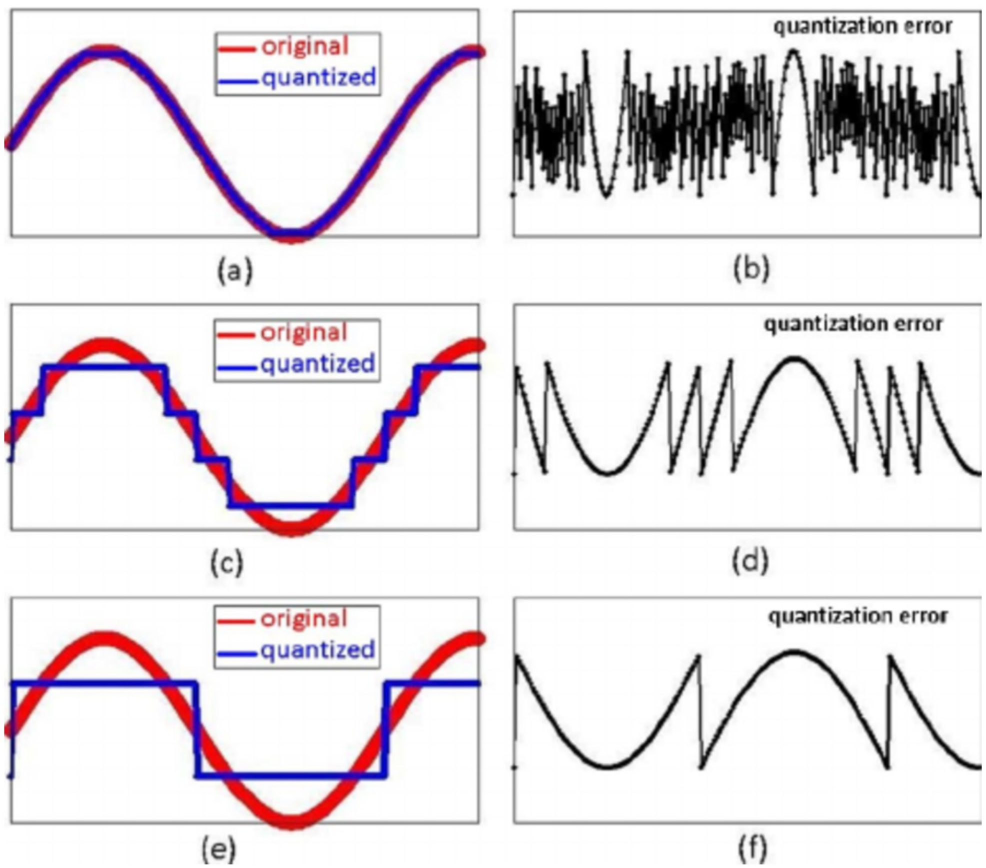


(Color online) Quantization of a sinusoidal signal and the corresponding quantization noise for three different cases. (a) and (b) 5 bits (b) and (c) 2 bits and (d) and (e) 1 bit. It can be seen that for 5 bits the quantization noise is almost randomly varying as in (b), and for the 1 bit case in (e), it has the particular characteristic frequency of the original signal.

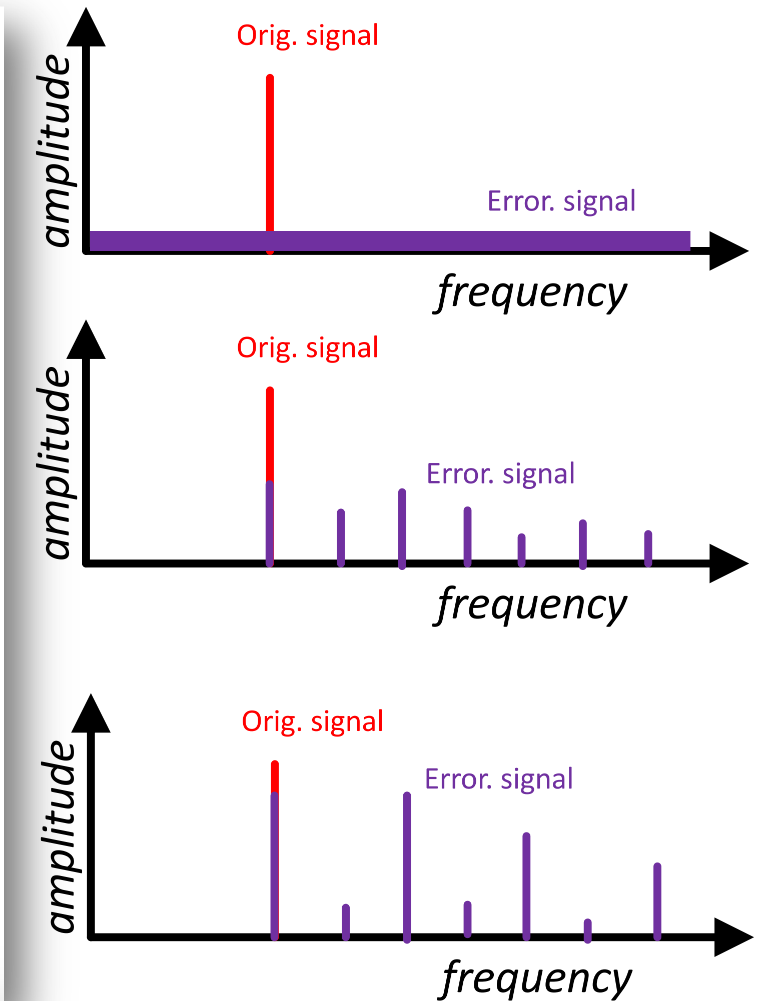
Pandey, Nitesh & Hennelly, Bryan. (2011). Quantization noise and its reduction in lensless Fourier digital holography. Applied optics. 50. B58-70. 10.1364/AO.50.000B58.

More Quantization Obfuscates Original Signal

Frequencies of Error Signal Become more uniform with higher bit depth



(Color online) Quantization of a sinusoidal signal and the corresponding quantization noise for three different cases. (a) and (b) 5 bits (b) and (c) 2 bits and (d) and (e) 1 bit. It can be seen that for 5 bits the quantization noise is almost randomly varying as in (b), and for the 1 bit case in (e), it has the particular characteristic frequency of the original signal.



Pandey, Nitesh & Hennelly, Bryan. (2011). Quantization noise and its reduction in lensless Fourier digital holography. Applied optics. 50. B58-70. 10.1364/AO.50.000B58.

Can’t Distinguish Signal From Error

- Once you’ve lost information, you can never regain it. There is no “enhance” button in real-life
- Motivation to **not** skimp out on quantizing (pick enough bits)
- But if you have to go low in bits…what can you do?

Quantization Error in Audio

@50 sec

https://www.youtube.com/watch?v=UaKho805vCE&ab_channel=MarkAndersonAudio

Quantization*
A Graphical Example

How many bits are needed to represent 256 shades of gray (from white to black)?

Bits	Range
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256

* Acknowledgement: Quantization slides and photos by Prof Denny Freemen 6.003

Quantization: Images

Converting an image from a continuous representation to a discrete representation involves the same sort of issues as with 1D signals (audio)

This image has 280 × 280 pixels, with brightness quantized to 8 bits.

Quantizing Images

8 bit image

7 bit image

Quantizing Images

8 bit image

6 bit image

Quantizing Images

8 bit image

5 bit image

Quantizing Images

8 bit image

4 bit image

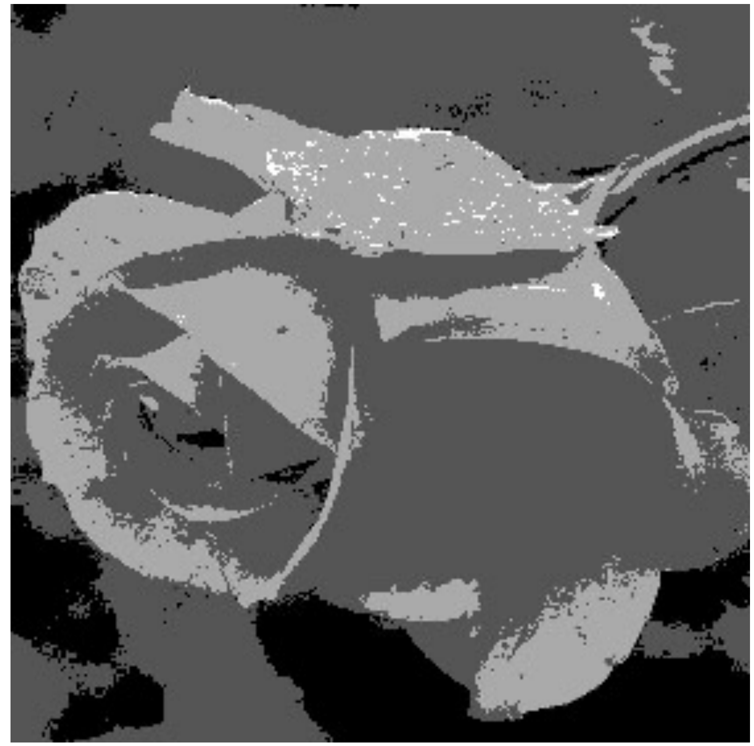
Quantizing Images

8 bit image

3 bit image

Quantizing Images

8 bit image



2 bit image

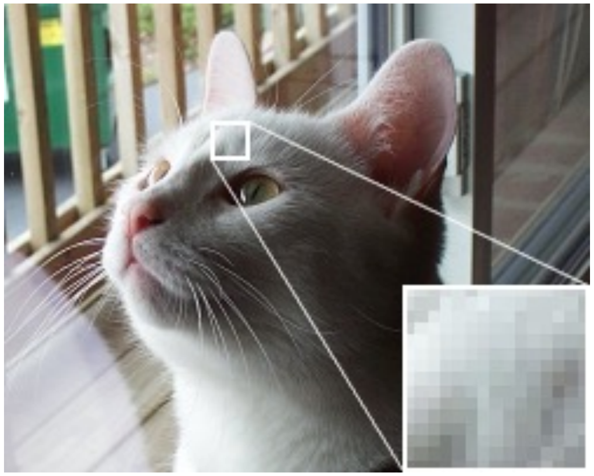
Quantizing Images

8 bit image

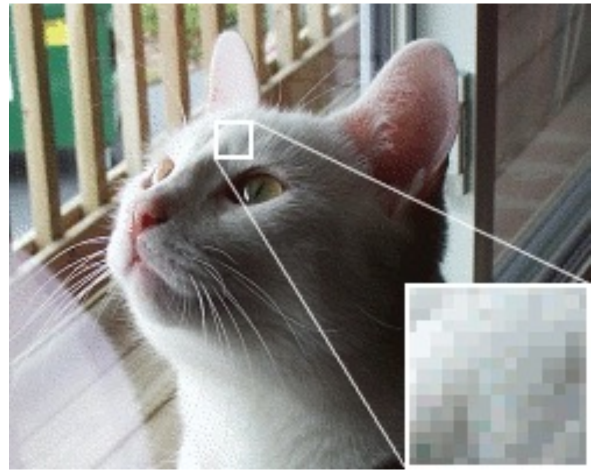
1 bit image

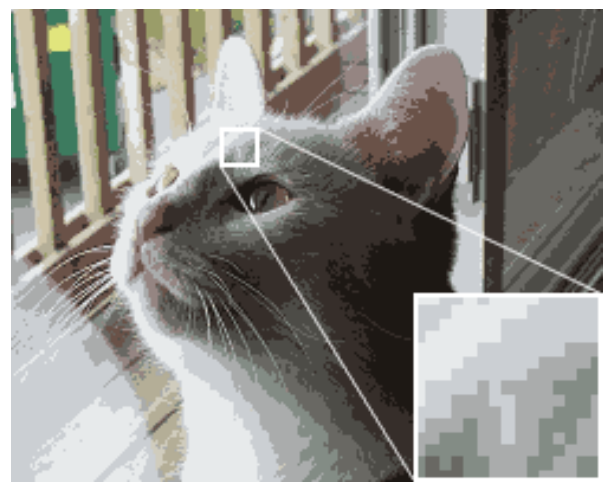
Quantizing Colors

256 (8bit) color kitteh



True color (24 bit) kitteh





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

16 color (4 bit) kitteh

Error Diffusion

- If you find yourself with an error signal that has structure* to it, there are ways to spread out the error.
- You'll never get rid of the error (which would involve making information from nothing), but you can "diffuse" it in the image in the frequency domain
- Consumers are often less sensitive to random noise than structured noise (eyes/ears tend to filter that out better)

*structure refers to non-uniform frequency composition…so like sharp frequency spikes

Dithering

- The solution is to add more noise ***when we quantize***, but do it so it spreads the frequency composition out to be more uniform

$$s(t) = s_o(t) + e_q(t, r)$$

Random variable

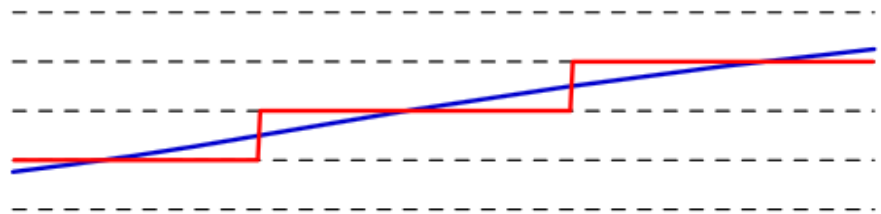
Quantization Error

Total Signal

Actual Signal
(never getting that back, sorry folks)

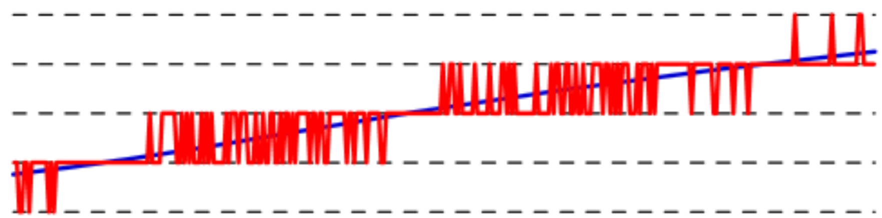
When quantizing in the first place and random noise in:

Quantization: $y = Q(x)$

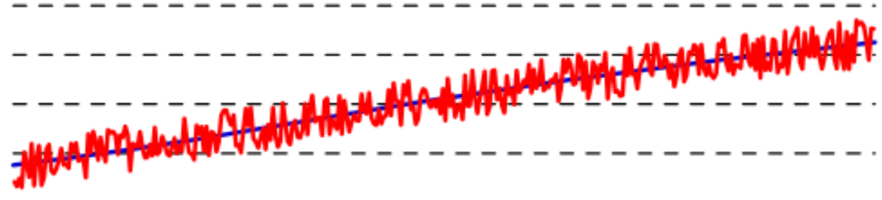


Quantization with dither: $y = Q(x + n)$

n = ±½ quantum



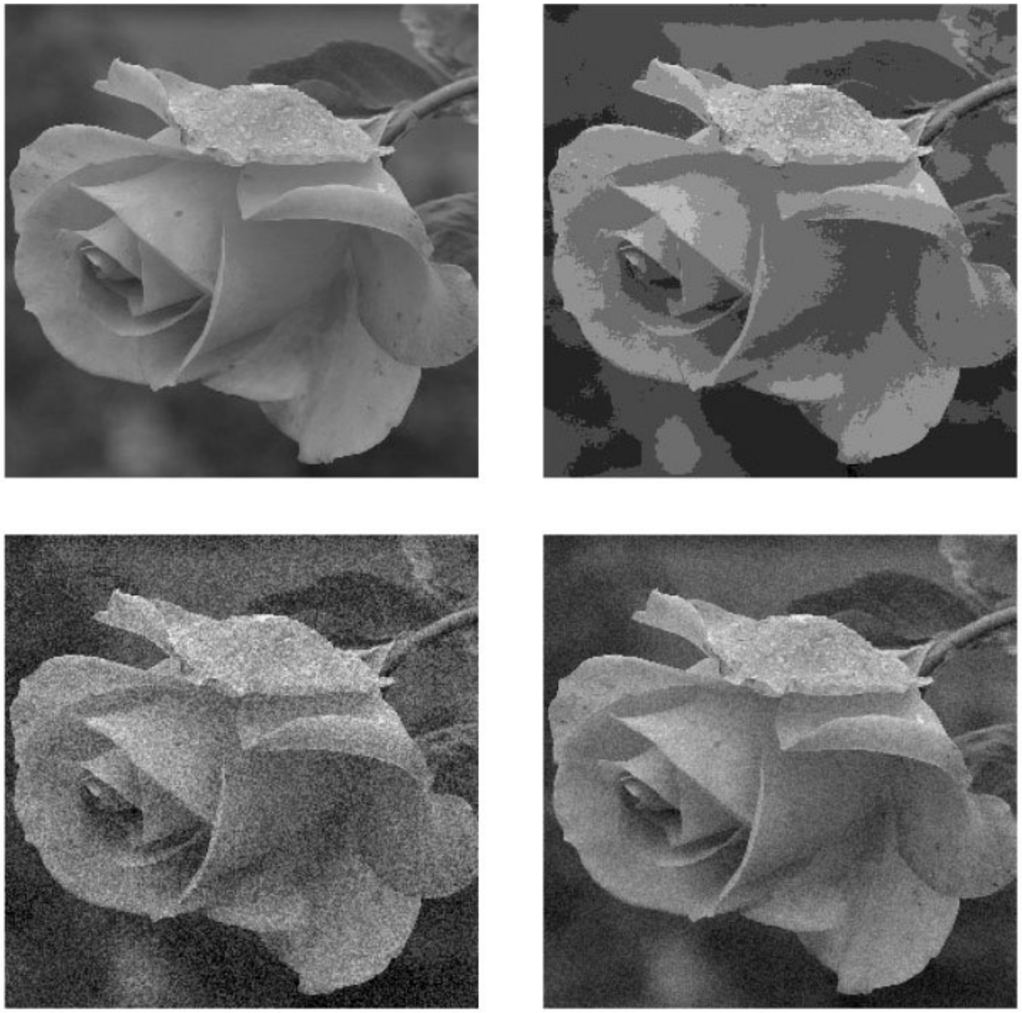
Quantization with Robert's technique: $y = Q(x + n) - n$



3 Bits Quantization

8 bits

3 bits



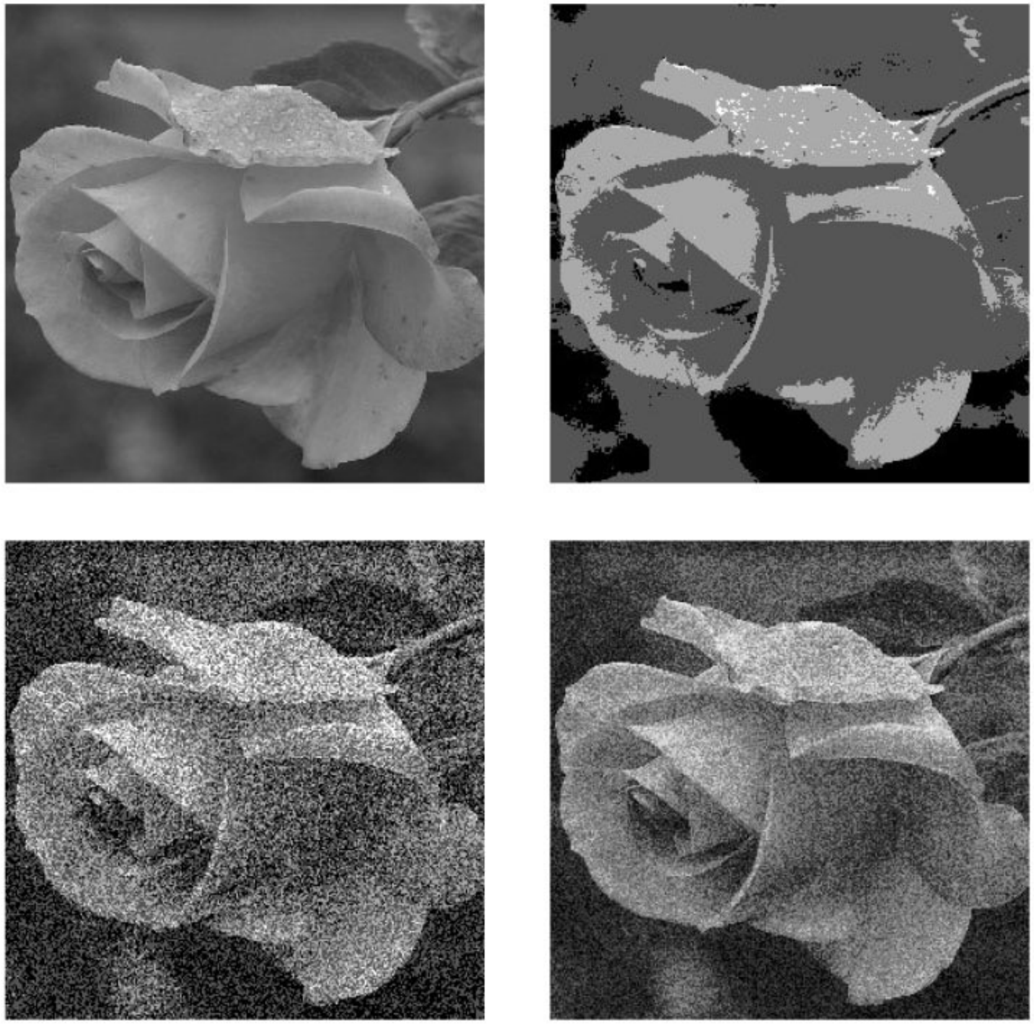
dither

Robert's

2 Bits Quantization + Noise

8 bits

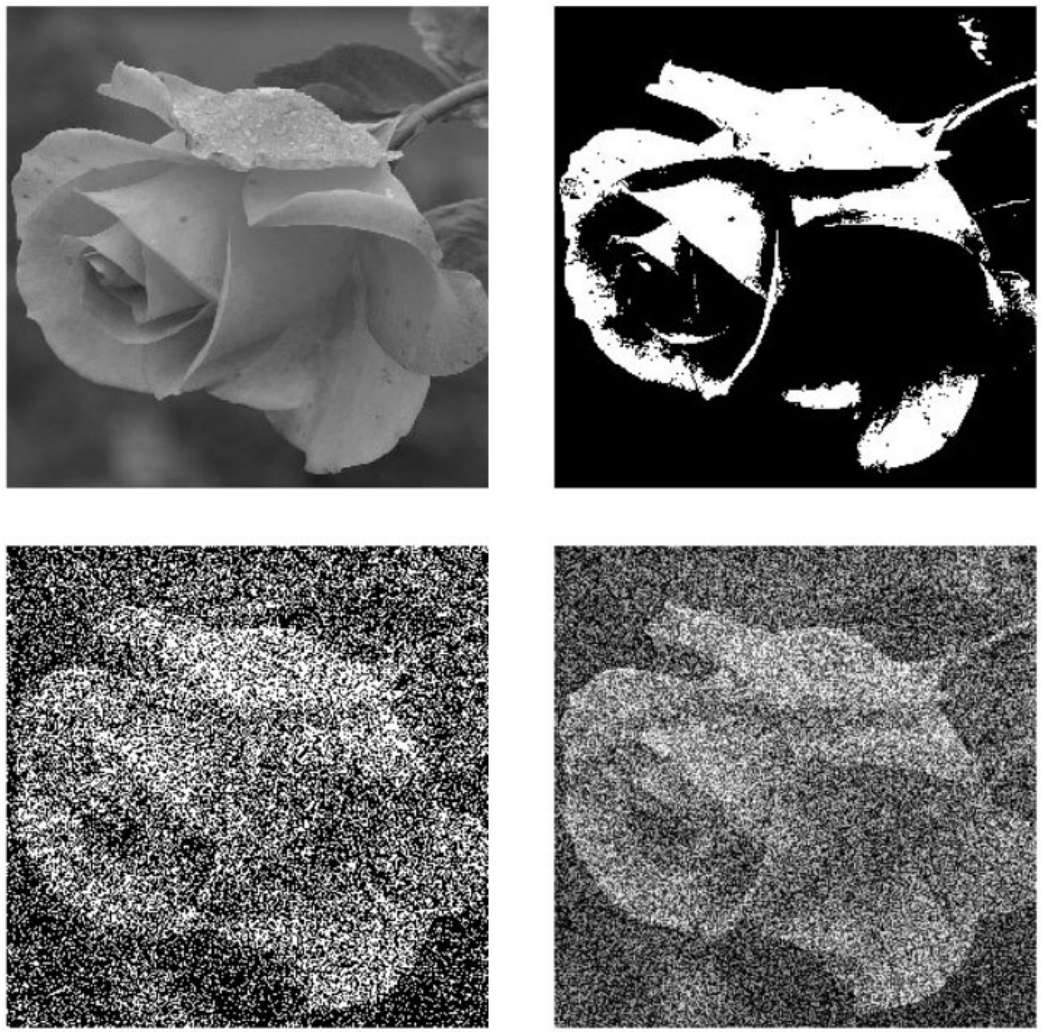
2 bits



dither

Robert's

1 Bit Quantization + Noise



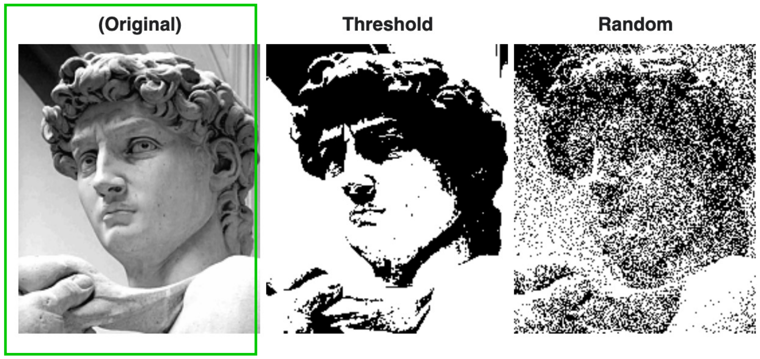
8 bits

1 bit

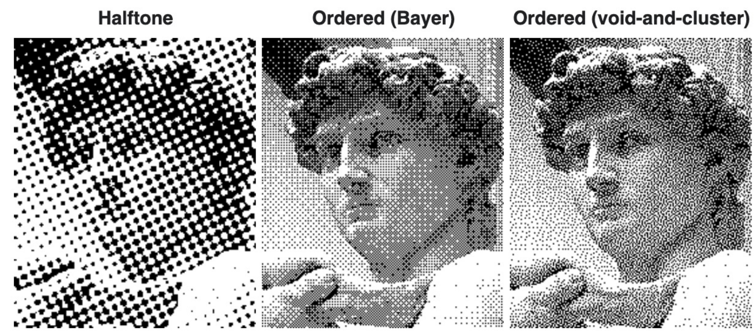
dither

Robert's

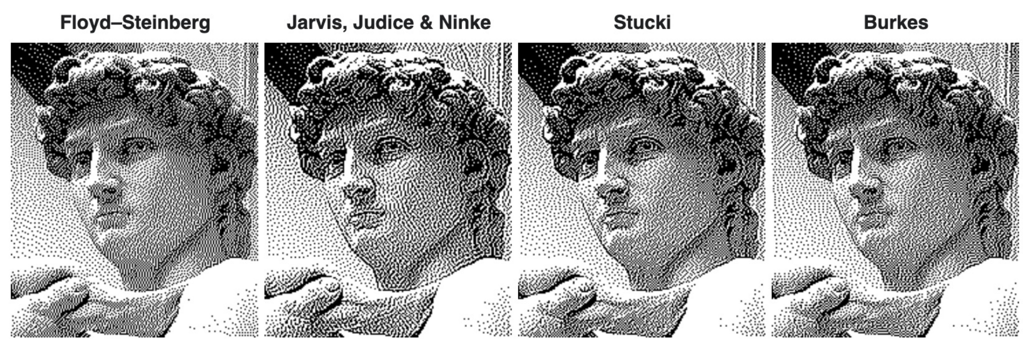
Dithering: Lots of Options/Algos

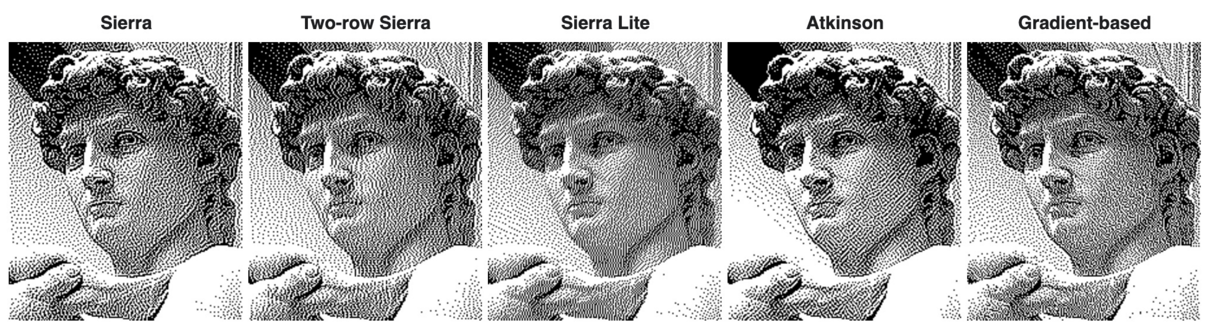


ORIGINAL
8bit Greyscale



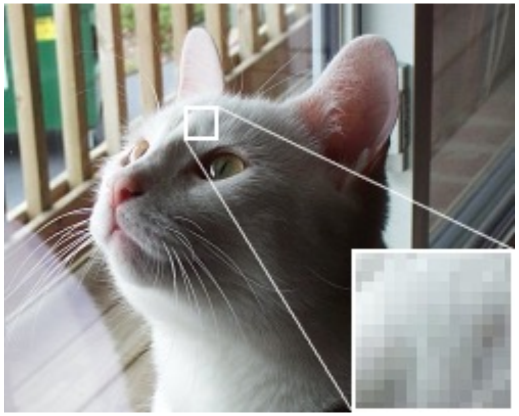
Every other example on page…1 bit quantization



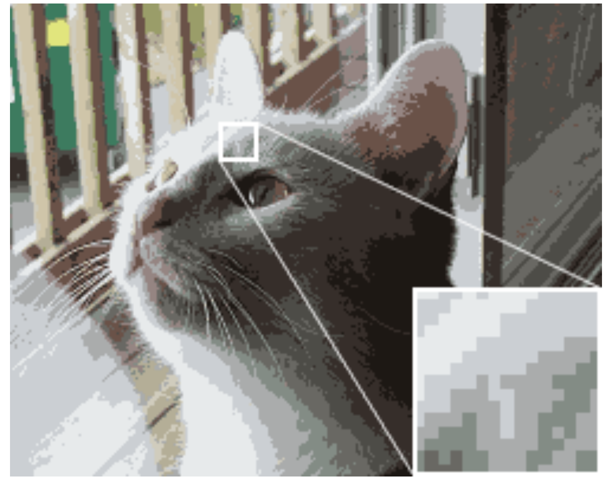


https://en.wikipedia.org/wiki/Dither

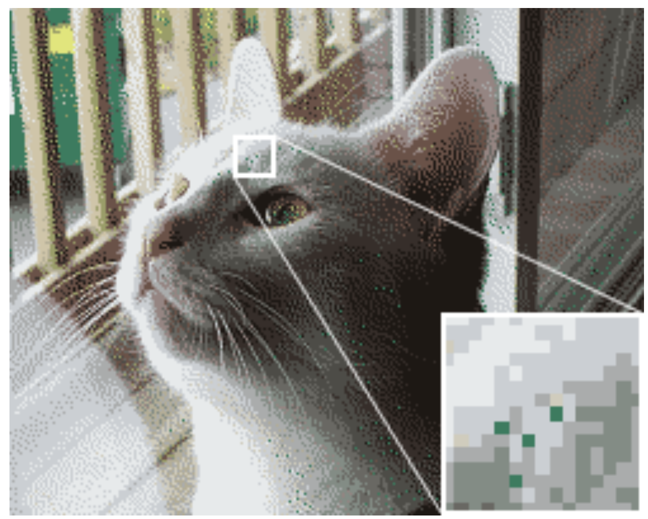
Color Dithering



True color (24 bit) kitteh



16 color (4 bit) kitteh



16 color (4 bit) dithered kitteh (Floyd-Steinberg)

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

Cool Student Project from Last year

Dithering

- In early computer/video games, space was at a premium, so if you could store your graphics at low (i.e one bit), then great!
- Lucas Pope (of *Papers Please!* fame) more recently created game *Return of the Obra Dinn* recreates the graphics of early games

Fantastic Discussion on Dithering:

https://forums.tigsource.com/index.php?topic=40832.msg1363742#msg1363742

Dithering in Audio

https://www.youtube.com/watch?v=h59LwyJbfzs&ab_channel=loopitstreamed