Part 3
Signal Processing Algorithms

0368.3464

עיבוד ספרתי של אותות

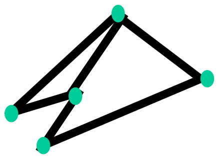
Digital Signal Processing for Computer Science

AKA

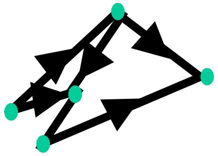
Digital Signal Processing – Algorithms and Applications

What is a graph?

A graph is a collection of
- *points* (AKA vertices, nodes)
- *lines* (AKA edges, links) between the points



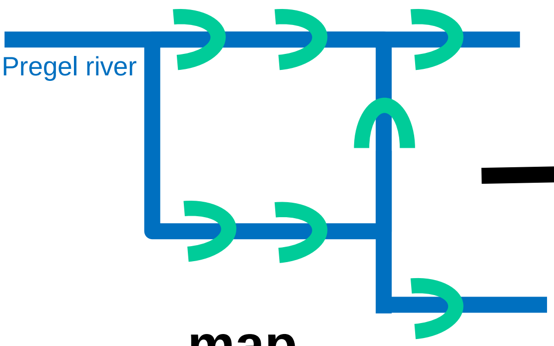
In DSP we will only use *digraphs* = directed graphs
where every line has a direction



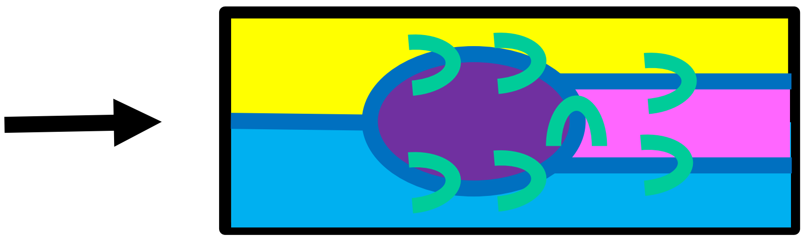
Graph theory was invented by Euler to solve
the puzzle of the Königsberg bridges

Is it possible to leave your home for a walk,
cross all the bridges exactly once,
and return home? (an *Euler cycle*)

But first he had to invent *topology*

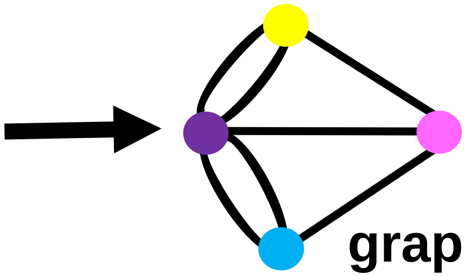


map



topologically equivalent map

There is an *Euler cycle*
iff every point has even *degree*

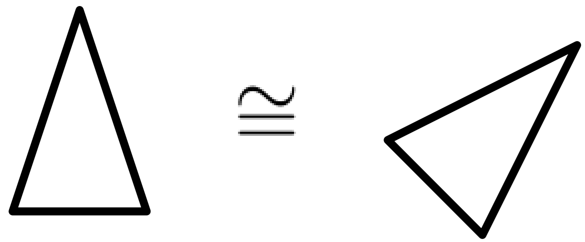


graph

Topology?

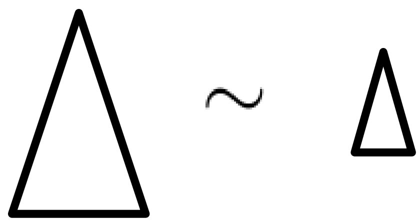
Topology is a generalization of *geometry*

- in geometry congruence allows *translation* and *rotation*



Side **S**ide **S**ide
Side **A**ngle **S**ide
Angle **S**ide **A**ngle
but not
Angle **A**ngle **A**ngle

- in *affine geometry* we also allow scale changes (zoom)



Side **S**ide **S**ide
Side **A**ngle **S**ide
Angle **S**ide **A**ngle
Angle **A**ngle **A**ngle

- in *projective geometry* we allow
  any transformation from lines to lines (maintains collinearity)
  here all triangles are equivalent, but squares are different
- in topology we allow any transformation
  that doesn't tear or glue together space
  (think of drawing on a rubber sheet)

Some more topology

Topology's equivalence relationship is called *homeomorphism*

A homeomorphism is a continuous function from *space* to *space*
with a continuous inverse function

In topology distance, angle, and linearity are meaningless

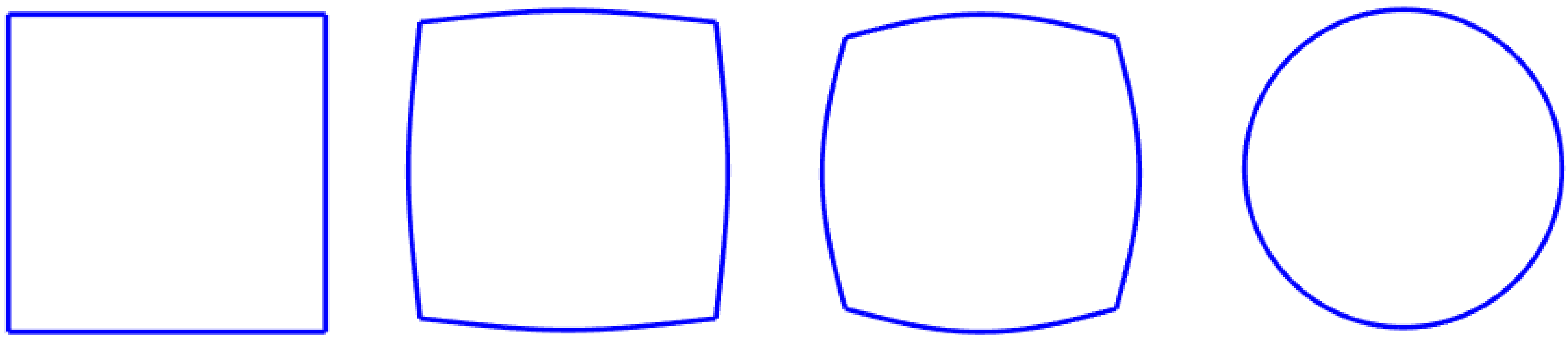
- triangle = square = polygon = circle
- all curves that don't cross themselves are equivalent
- a figure 8 is not the same as a circle (would require a *tear*)
  (the number of holes is preserved)
- in 3D topology a sphere is equivalent to a cube
  but not to a donut

What Euler realized is that the existence of a *Euler cycle*
is independent of the bridge location and orientation

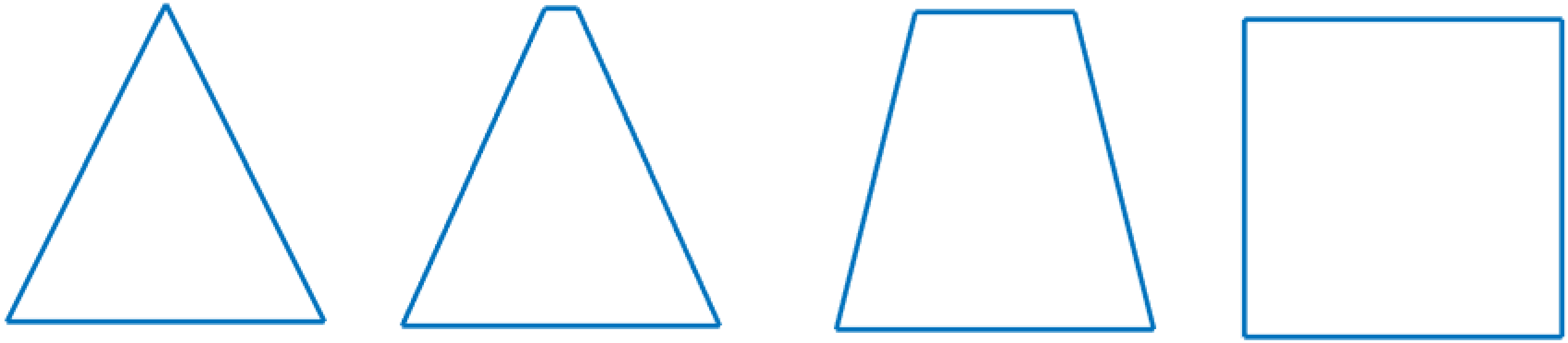
The bridge puzzle is truly a topological problem

Continuous transformations

Continuously morph a square into a circle (and back again)



Continuously triangle into a square (and back again)



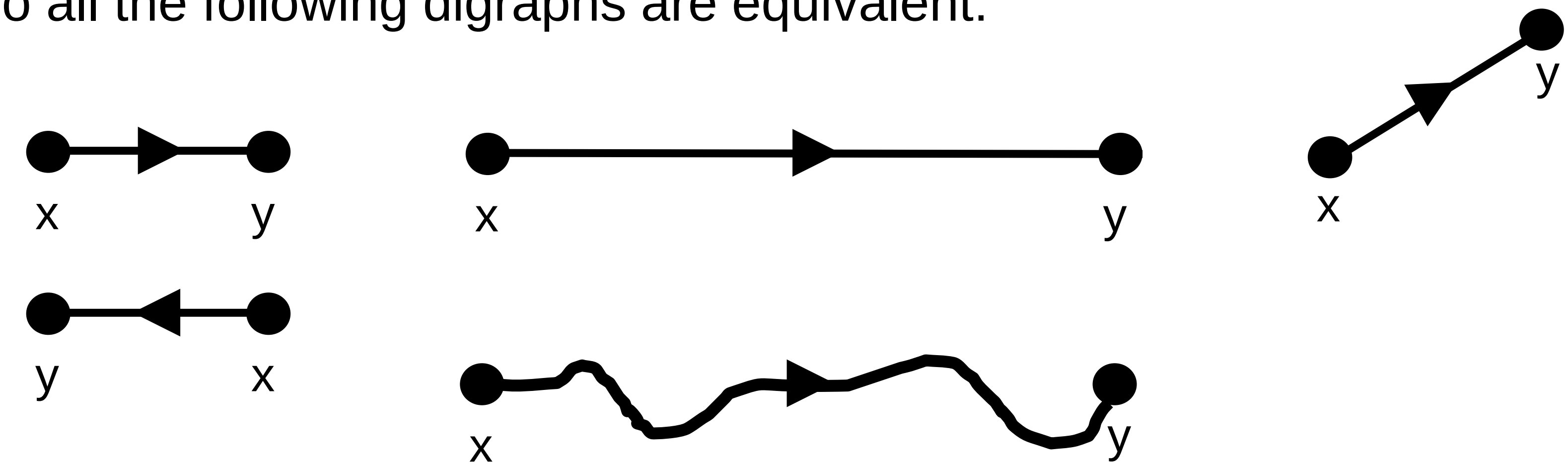
Topology and graph theory

In graph theory all we care about is connectivity
which point is connected to which point

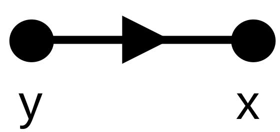
We don't care about the length or angle of the line
or even if it is a line

The meaning of a graph is purely topological

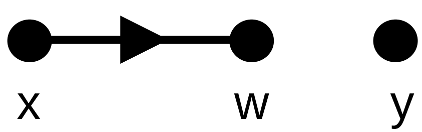
So all the following digraphs are equivalent:



But not



or

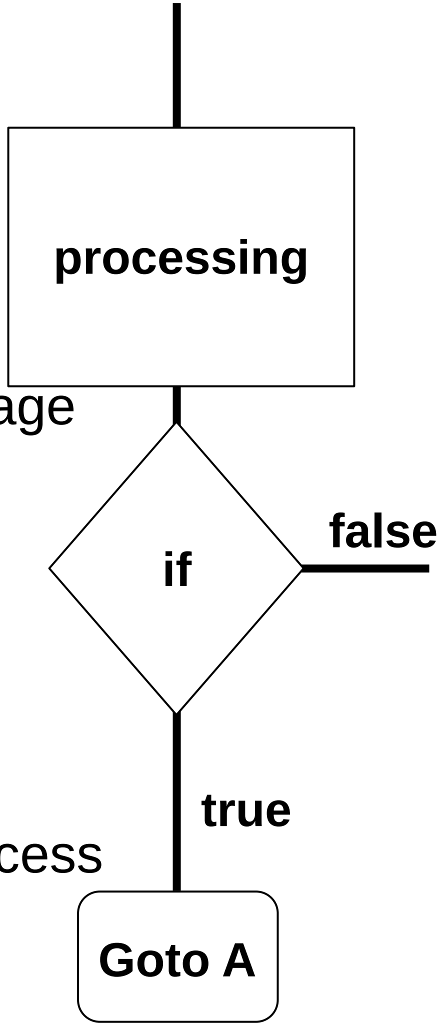


Graph theory in CS

In the early days of computer science
programs were represented by block diagrams
which are a kind of graph

This representation has been mostly abandoned
for several reasons:

- block diagrams are actually a programming language
  so using them in addition to code
  means maintaining 2 code different sets
- block diagrams are tightly coupled to
  imperative program with goto statements
  which has been disparaged
- block diagrams are purely *documentation*
  and add nothing positive to the programming process
- block diagrams only describe *algorithms*
  Wirth's Law
  programs = algorithms + data structures



But in DSP we still use signal flow graphs!

Signal flow graphs

Shannon introduced *signal flow graphs* in which

- the points represent signals
- the lines (and things on lines) represent signal processing functions

These graphs capture both

- algorithms and
- data structures

In addition to their purely documentary function

signal flow graphs are useful because of graphical mechanisms

for simplifying graphs

lowering computational power or memory requirements

The simplest graph

The simplest signal flow graph has 1 point and represents a signal

x ●

When we write a letter next to a point (below, left, right, above)

it represents the name of the signal (here: x !)

When interpreting signal flow graphs

it is often useful to ask – what is the value of the signals at time n ?

So we sometime draw $\overset{\bullet}{x_n}$

But don't be confused!

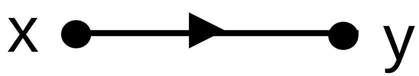
The point represents the signal x_n $\forall n = -\infty \ldots +\infty$

not a particular value

The next simplest graph

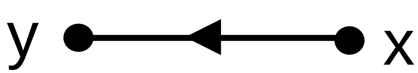
The simplest *nontrivial* signal flow graph has 2 points and one line
and it represents signal identity (y = x, i.e., $\forall n = -\infty \ldots +\infty \quad y_n = x_n$)

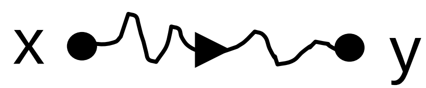
identity = assignment

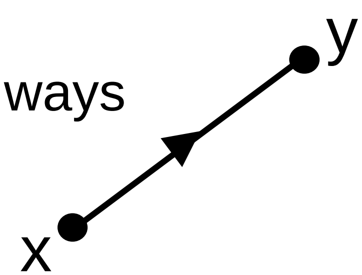


y = x

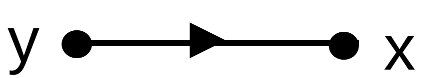
Of course, due to the topological nature of graphs
we could have drawn this graph in many other ways

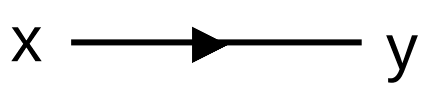






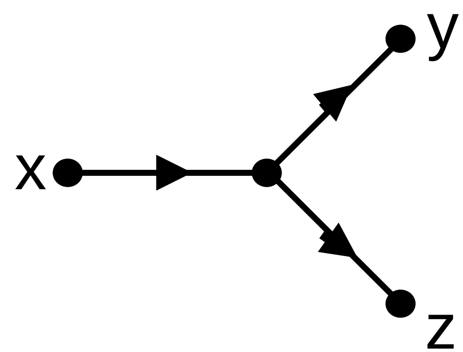
And in this case *only* we can also reverse the line direction

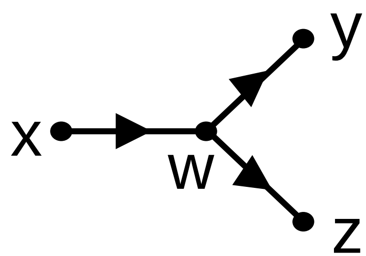




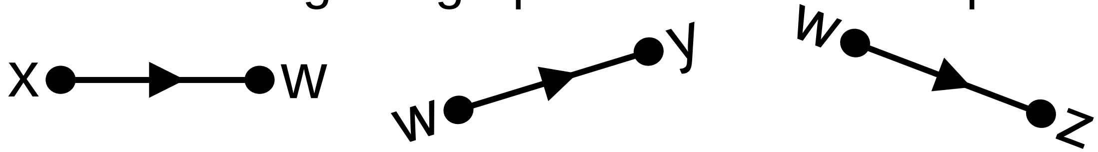
Note that we will often neglect to draw the point when it is obvious
(i.e., at the end of a line)
(we will later only draw points in specific places …)

What does this mean?





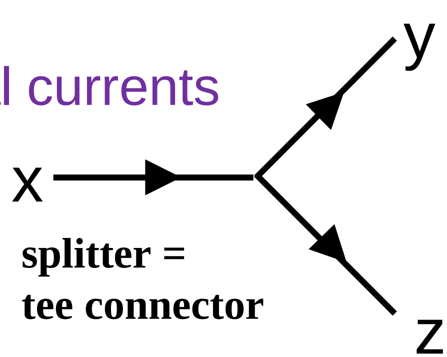
We can figure this out by naming the unlabeled point w
and breaking the graph down into three parts



So w = x and y = w = x and also z = w = x

WARNING! Do not think of this as electrical currents
in which case y=x/2 and z=x/2 !

This graph is called the splitter



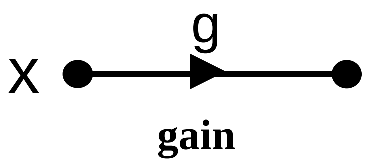
y = x
and
z = x

Note that the splitter always has
1 signal going in and 2 signals coming out

Gain

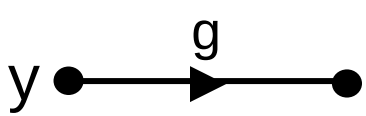
The simplest signal processing is the gain $y = g\,x$ ($\forall n = -\infty \ldots +\infty \quad y_n = g\,x_n$)

We draw this by putting the letter **g** next to the arrow



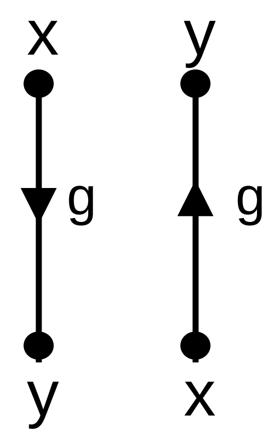
$y = g\,x$

Note that (for $g \neq 1$) this is very different from



$x = g\,y$

but the same as



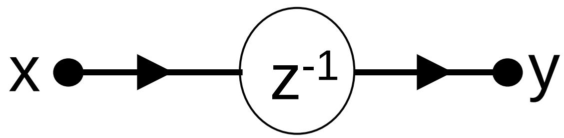
etc.

Be careful!

A letter near a point tells you the signal's name
but a letter near an arrow represents a gain

Delay

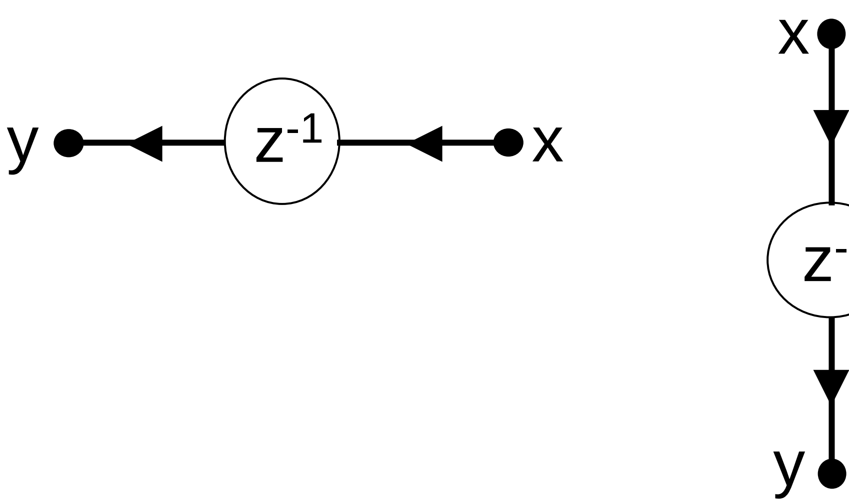
We have seen that the unit delay is very important in DSP
and so it deserves its own graphical symbol

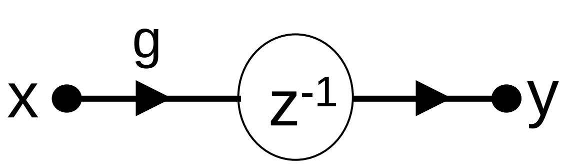


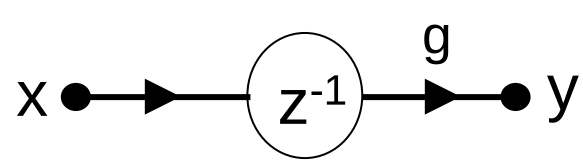
unit delay

$y = \hat{z}^{-1} x$ $\quad$ ($\forall n = -\infty \ldots +\infty \quad y_n = x_{n-1}$)

and as usual we can draw this in various orientations, such as



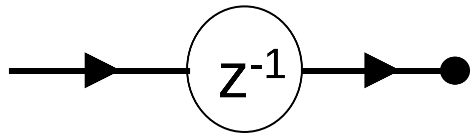
is 

the same as 

?

Drawing points

We will always explicitly draw the point after a delay element

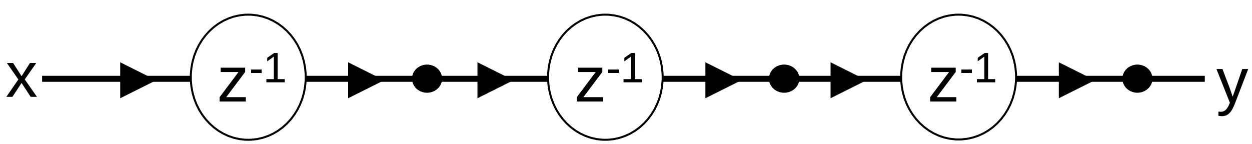


Since this point represents a signal value that must be remembered
that is, a memory location

For this reason we frequently use the term *memory point*

Marking memory points
help us count up how much memory is required

For example, we see that $y = \hat{z}^{-3}\,x$ requires 3 memory points



Note: We will sometimes temporarily draw and label points
just in order to understand the graph

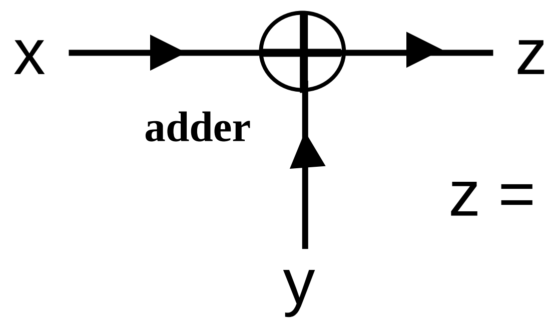
Adder

Signal addition is very important as well!

We define the two-signal adder

which of course means

$$\forall n = -\infty \ldots +\infty \quad z_n = x_n + y_n$$

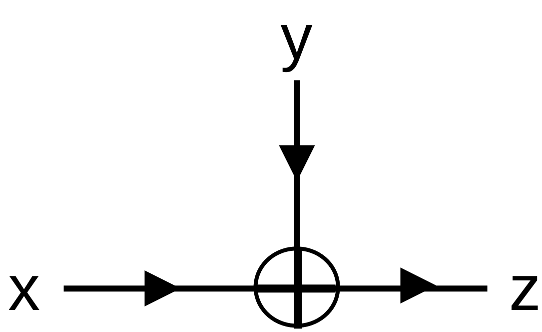


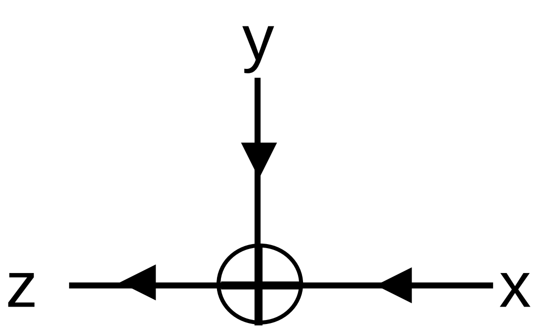
$$z = x + y$$

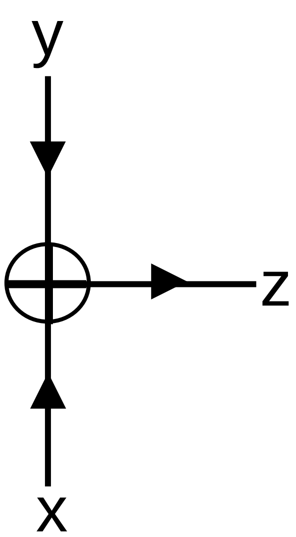
Note that the adder always has

2 signals going in and 1 signal coming out

As usual, we could draw the adder in many ways!

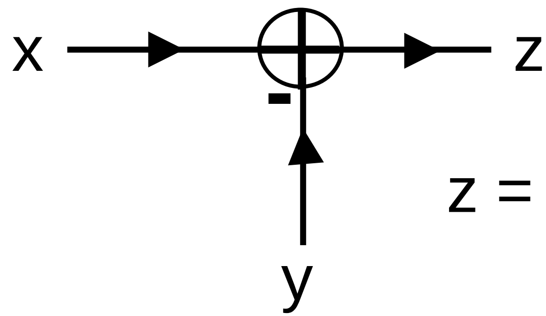






Subtractor

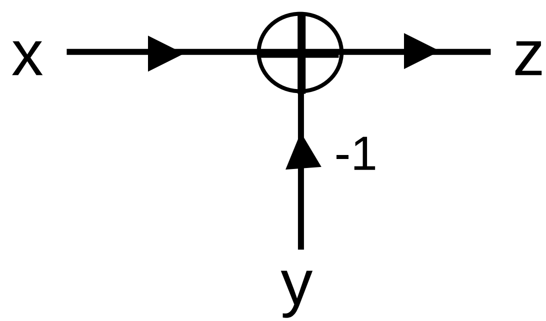
For convenience we also define the 2-signal subtractor



$z = x - y$

Note the position of the minus sign
It can be at either (or both) of the adder's inputs!

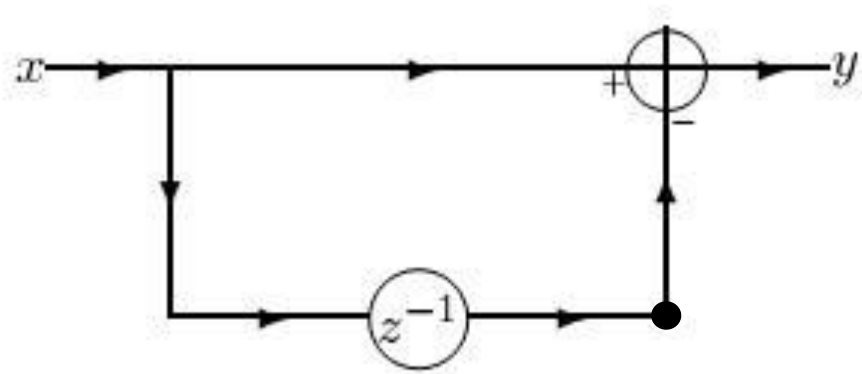
Although we could have used



instead

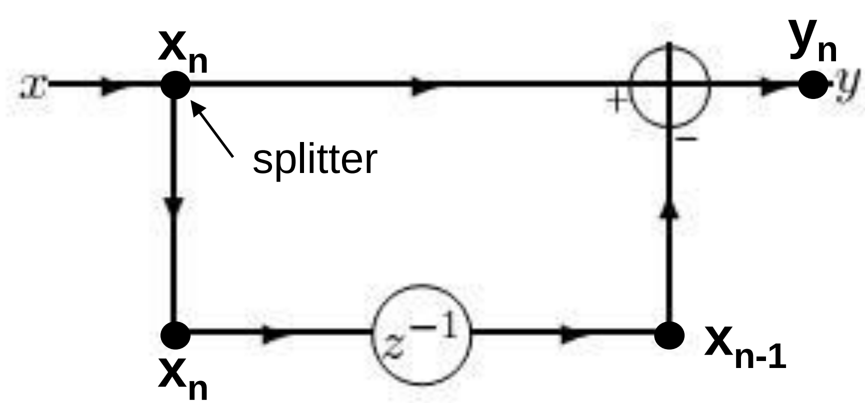
The finite difference

We can now use what we have learned so far to draw a useful graph the finite difference $y = \widehat{\Delta}\, x$ (i.e., $y_n = x_n - x_{n-1}$)



$$\mathbf{y_n = x_n - x_{n-1}}$$

To see that this is correct label needed points (not just the memory points) with their value at time n



The butterfly

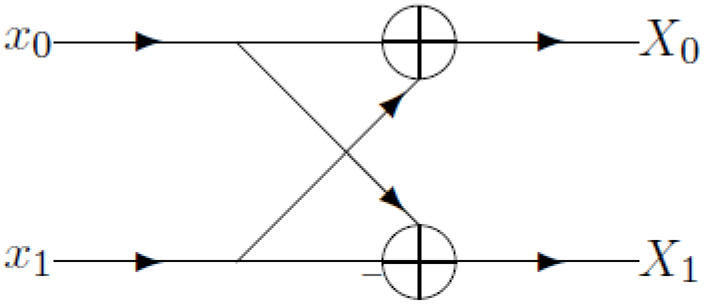
Remember the DFT for N=2 ?

$$X_0 = x_0 + x_1$$

$$X_1 = x_0 - x_1$$

We can draw this as a DSP graph

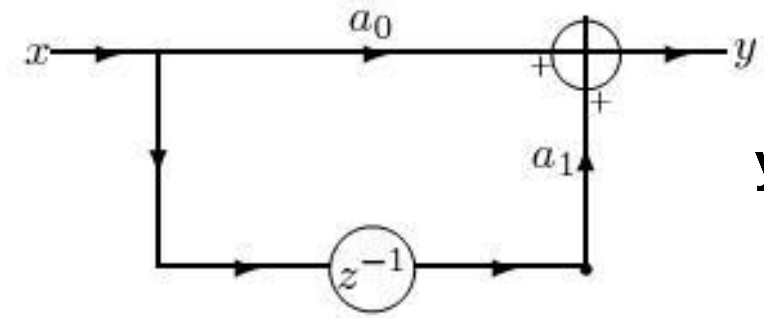
(it is not really a signal flow graph!)



Rotating this 90 degrees and using a lot of imagination
one can understand why this is called a butterfly

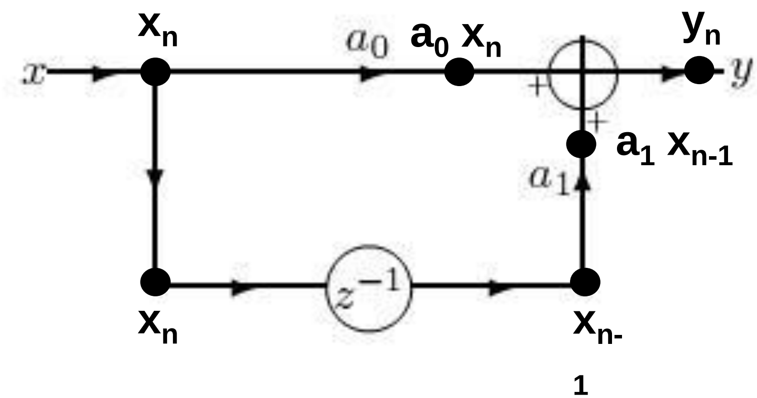
The basic MA filter

Let's draw something even more interesting



$$y_n = a_0\, x_n + a_1\, x_{n-1}$$

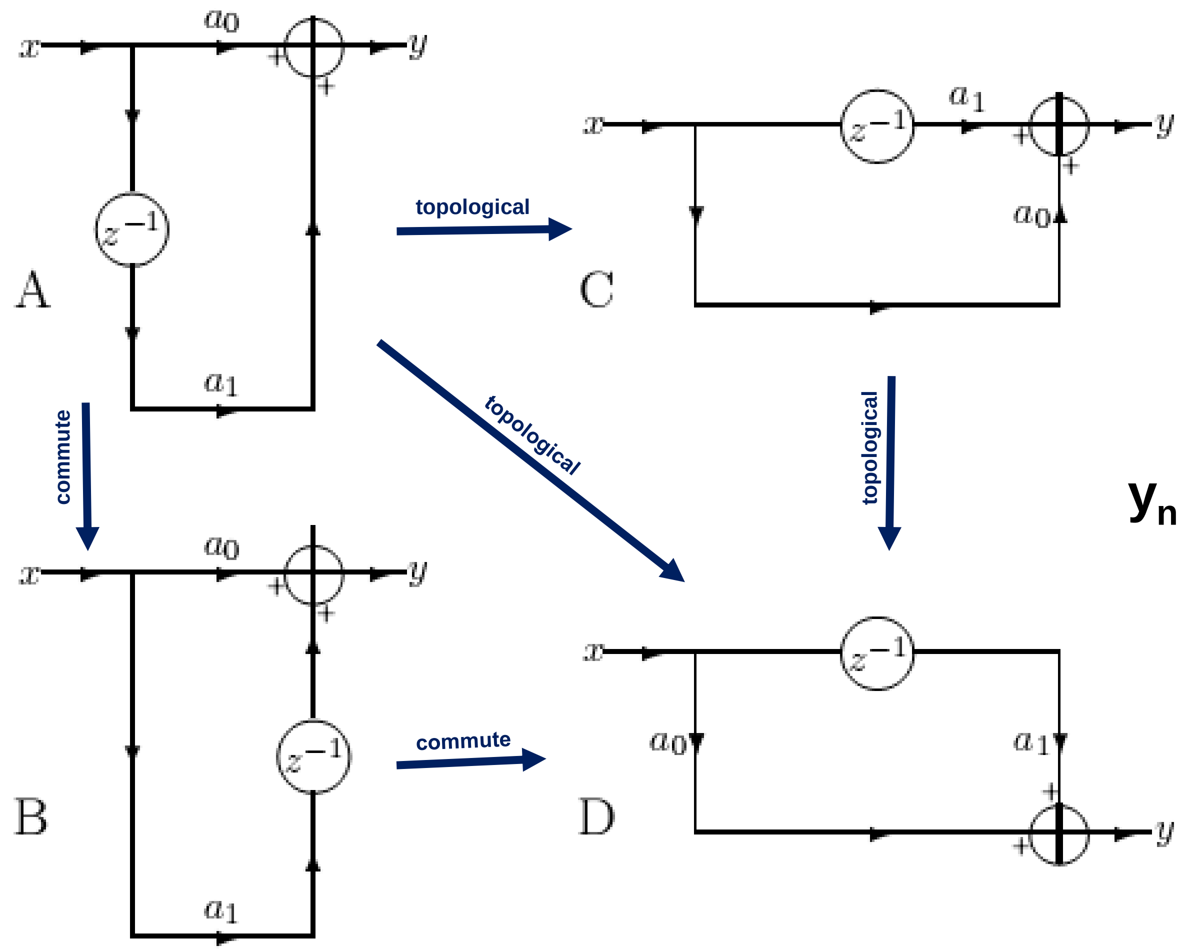
To see that this is indeed the MA filter label all these points



Basic MA blocks

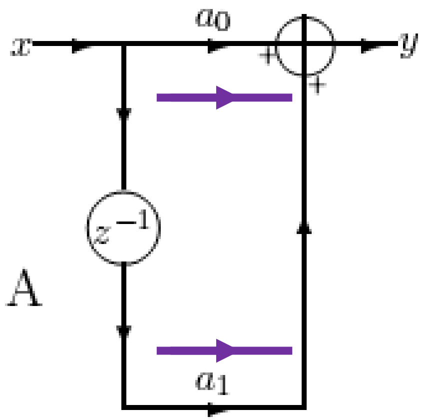
Here are 4 interesting ways to draw this same simple MA filter

What transformations brings us from one to the other?

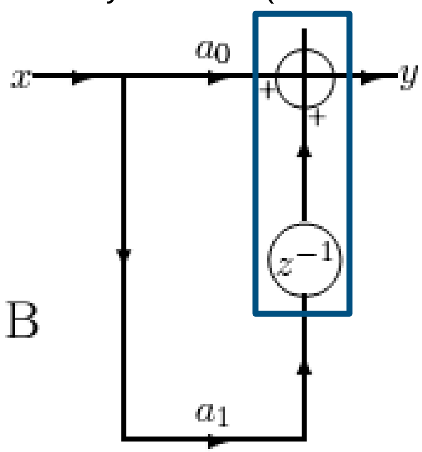


$$y_n = a_0\, x_n + a_1\, x_{n-1}$$

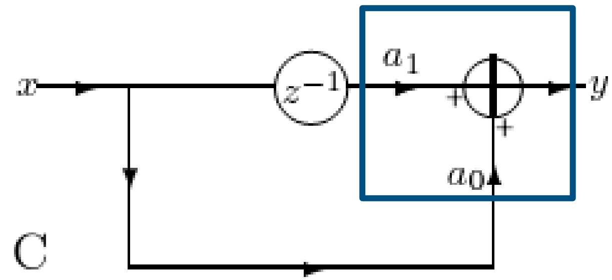
Why do we need 4 blocks?



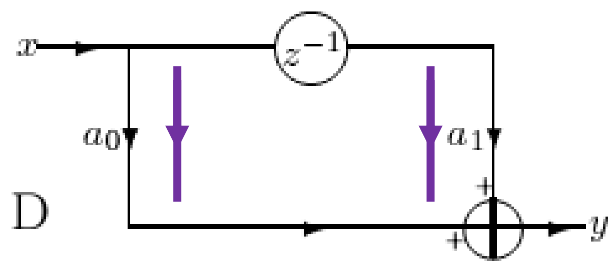
Products all parallel –
easy to iterate (we'll see later)



Delay element adjacent to adder



Products next to adder –
can make memoryless chip



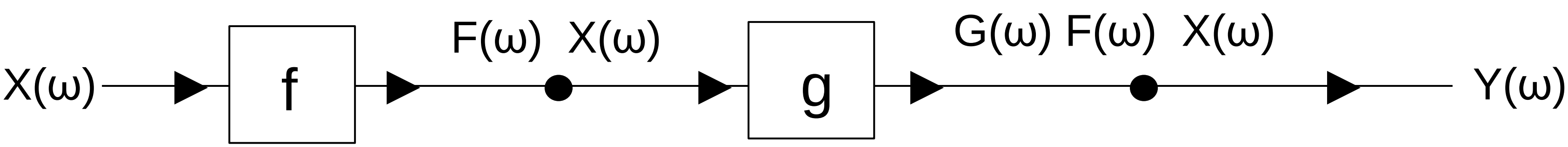
Products all parallel –
easy to iterate (we'll see later)
y line is different height from x line

Commutativity

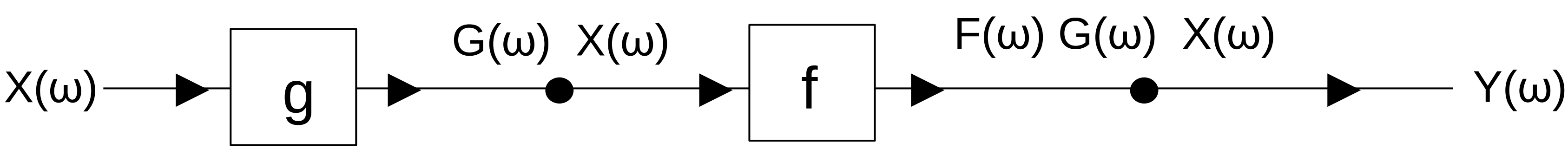
Note that it is obvious that the gain g and the delay $\hat{z}^{-1}$ commute
but this is true more generally for any two filters

While somewhat complicated to prove in the time domain
it is simple to see in the frequency (or z) domain

Since filters obey Y(ω) = H(ω) X(ω)
two filters – f and g – in series obey Y(ω) = G(ω) F(ω) X(ω)



while in the opposite order Y(ω) = F(ω) G(ω) X(ω)



which is the same thing since functions commute!

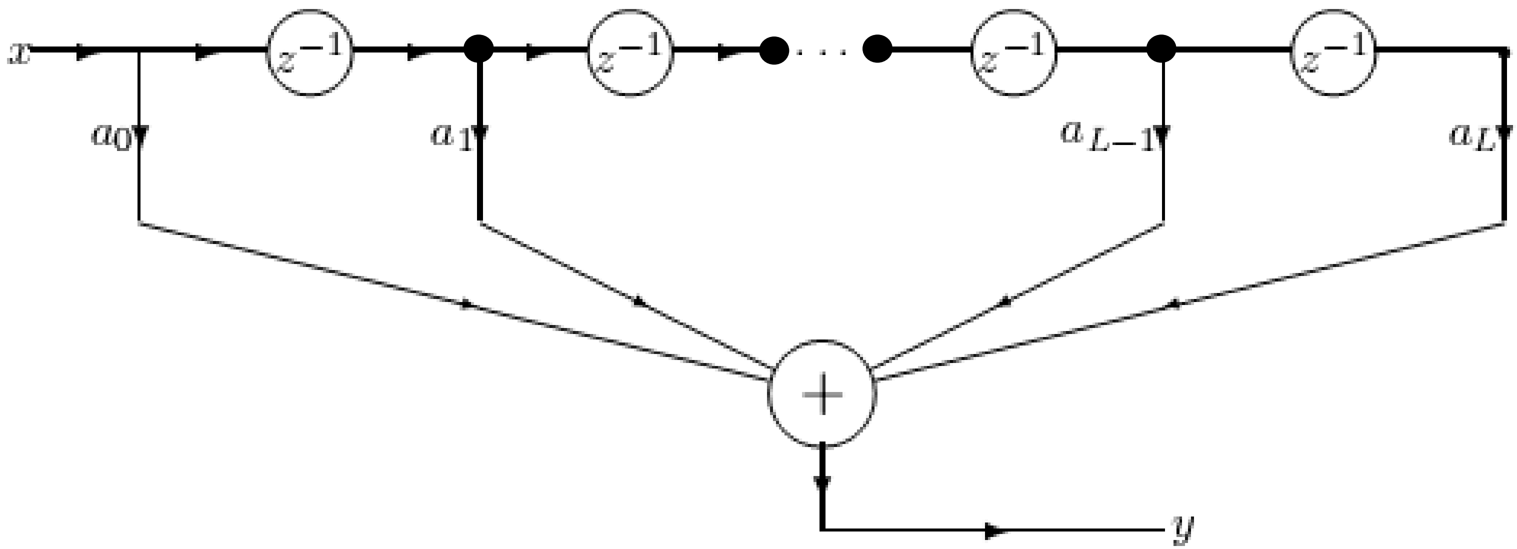
Show 2 systems that do not commute

General MA

Now we consider the general MA filter with L coefficients

$$y_n = \sum_{l=0}^{L} a_l x_{n-l}$$

We would like to draw



but we only defined 2-input adders !

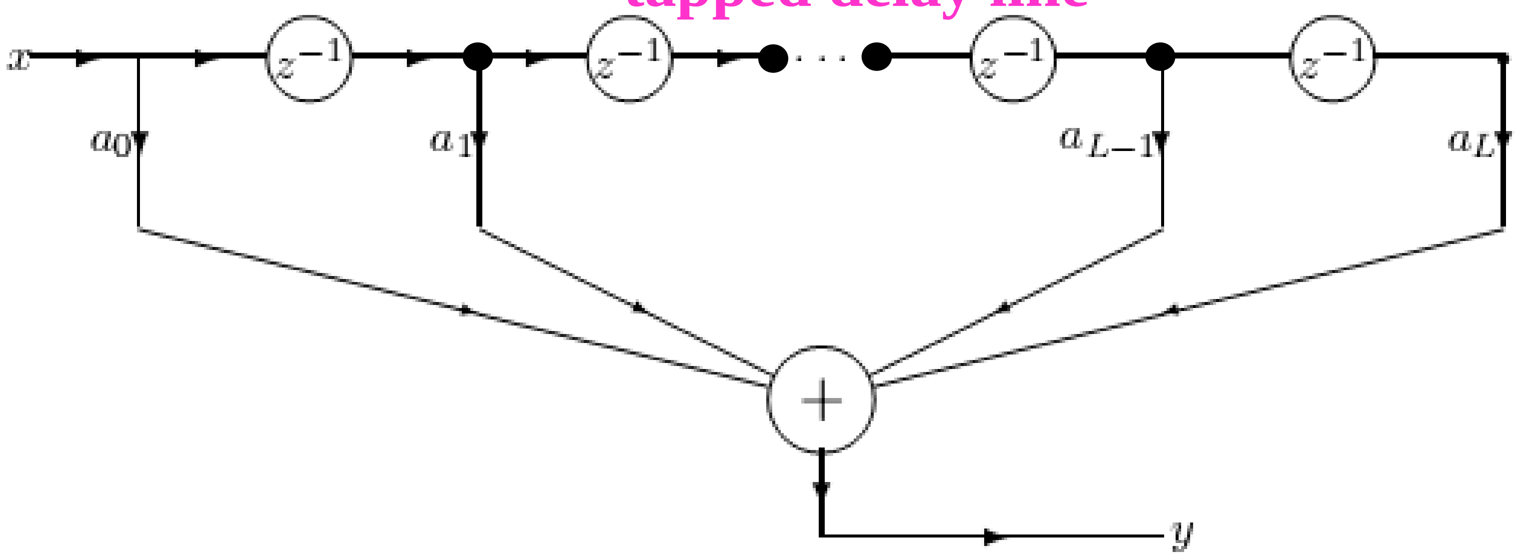
Tapped delay line

Before correcting this, note that top of this diagram
has an interesting analog interpretations

Engineers think of this as a *tapped delay line*
similar to a length of cable with finite transmission velocity

But since information travels in (copper or optical) cables
at 2/3 the speed of light (200 meters per μsec)
you need a long cable for significant delay !

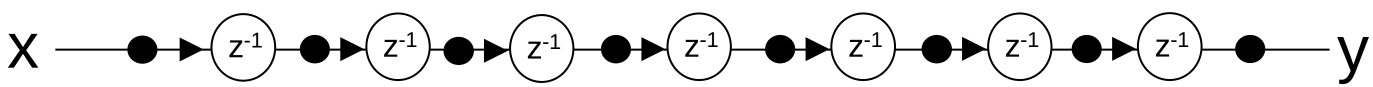
tapped delay line



A data structure!

We will think of this differently (and find a data structure in addition to the algorithm)

Considering the memory points from some time
we find a data structure (assume L=8)
with the following time varying contents



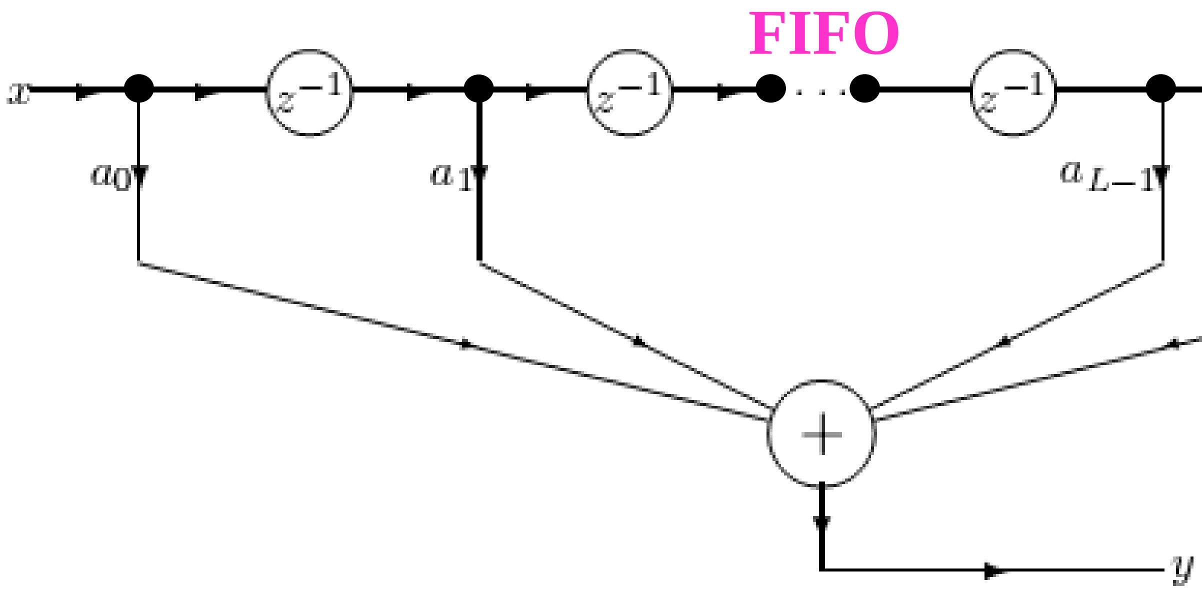
x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1
x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2
x_{10}	x_9	x_8	x_7	x_6	x_5	x_4	x_3
x_{11}	x_{10}	x_9	x_8	x_7	x_6	x_5	x_4

We see that values that enter first from the left
exit (are discarded) first to the right
so this is a **FIFO** buffer

How do we fix the adders

So, were we to use an N-input adder

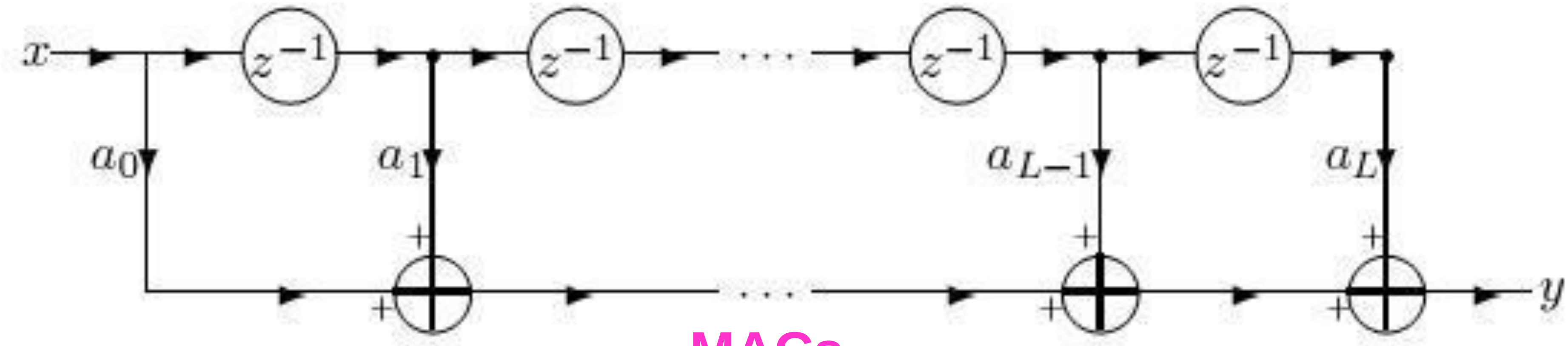
we would have a FIFO, multiplications, and an adder



Let's perform the additions one at a time!

General MA – 1st way

$$y_n = \sum_{l=0}^{L} a_l x_{n-l}$$



MACs

We still have the tapped delay line = FIFO as a data structure
but now we perform **M**ultiplication + **Ac**cumulations (MACs)

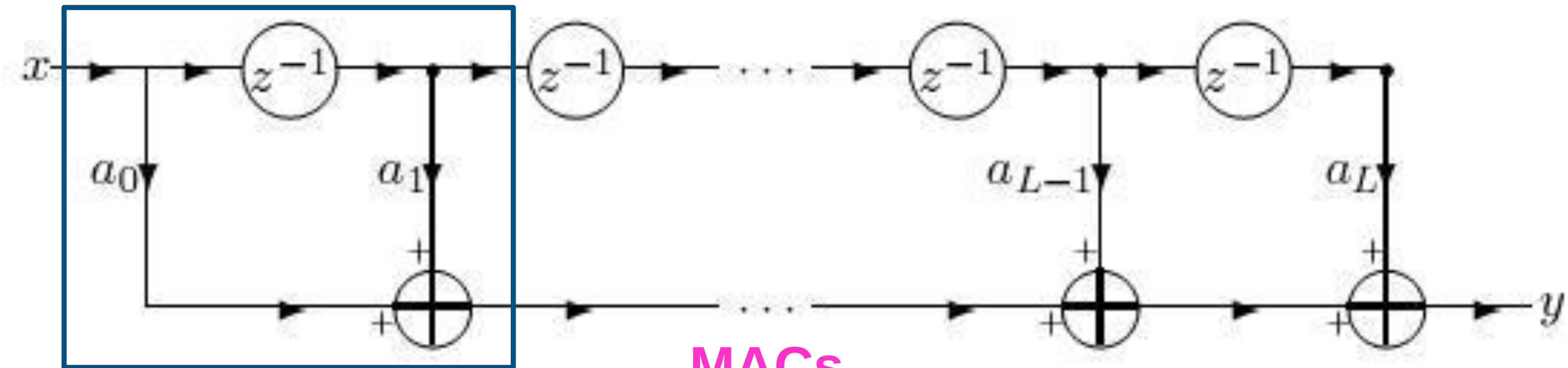
We have previously mentioned how important MACs are in DSP

Another way of looking at this is iteration on one of our basic MA blocks

Which one ?

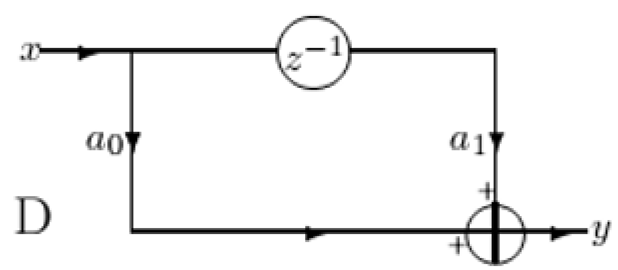
Iteration – 1st way

$$y_n = \sum_{l=0}^{L} a_l x_{n-l}$$



MACs

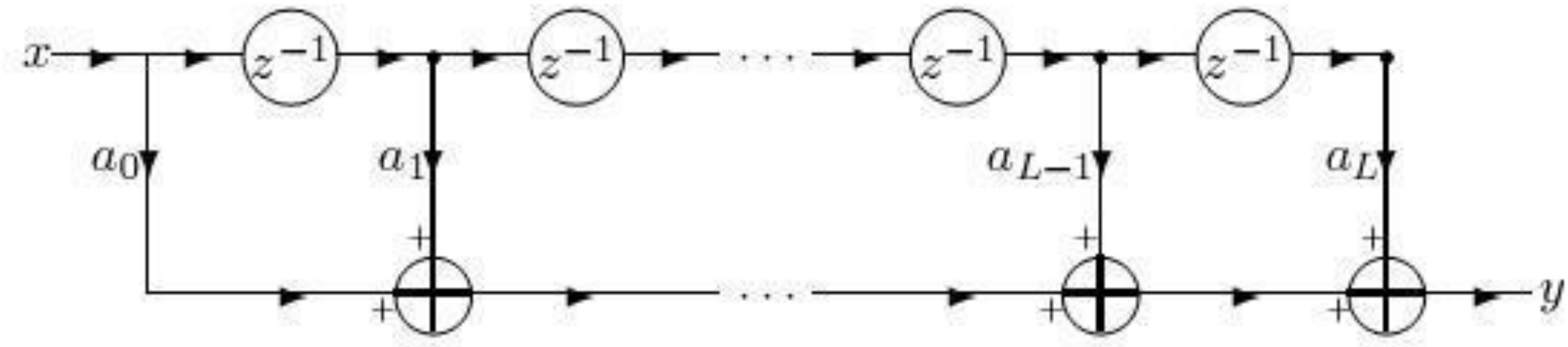
We iterate on basic block D



So this graph tells us

1. Data structure = FIFO
2. Algorithm = iteration over MA block D

The signal's point of view



We saw how to look at this from the processing point of view

Sometimes it is useful to look at graphs from the signal's point of view!

- the signal enters the filter and is split into 2 replicas : A and B
- gain is applied to replica A, replica B is delayed
- replica B is split in two : C and D
- gain is applied to replica C, which then is added to replica A
- etc.

General MA – 2nd way

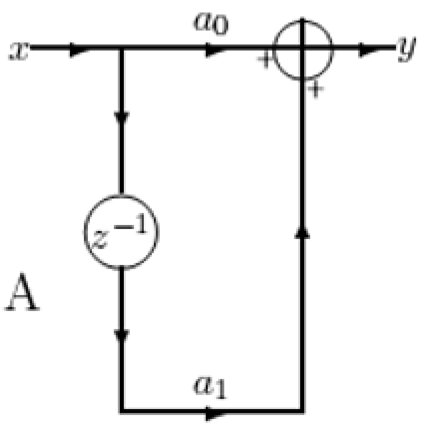
This isn't the only way to compute a general MA

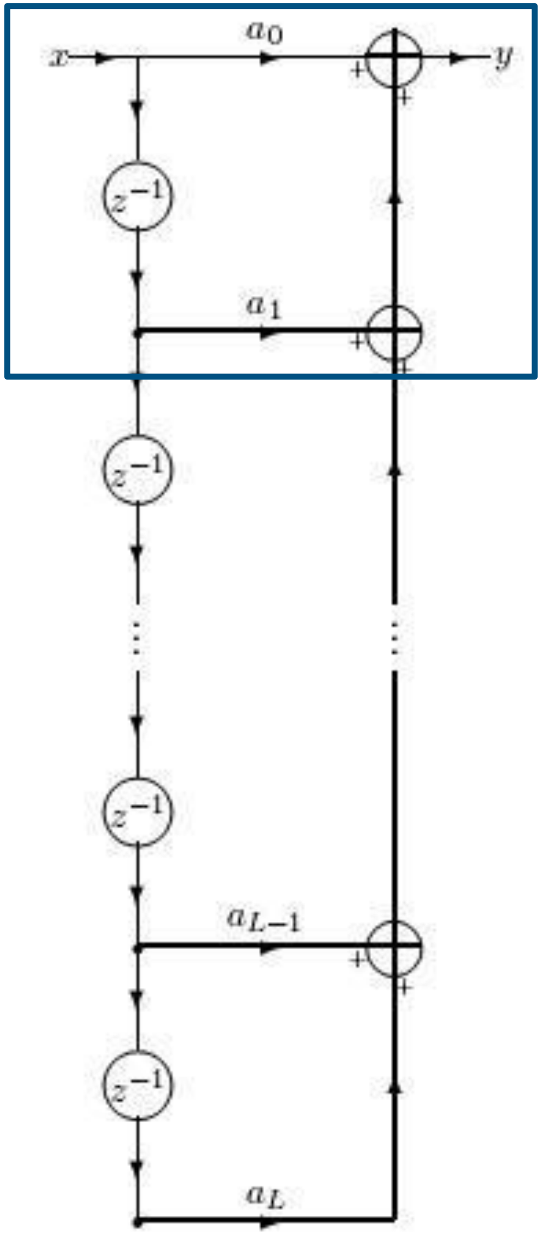
Here we see an alternative

It still uses a FIFO data structure

(which is now vertical – but who cares?)

Which basic MA block is used here?



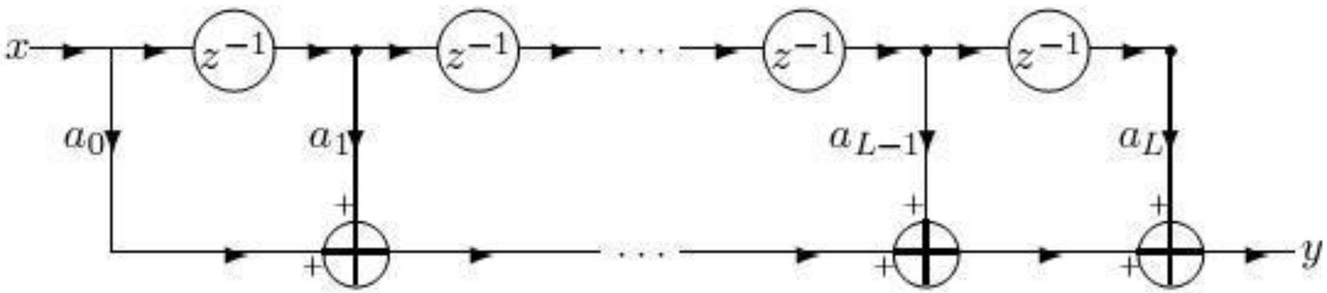


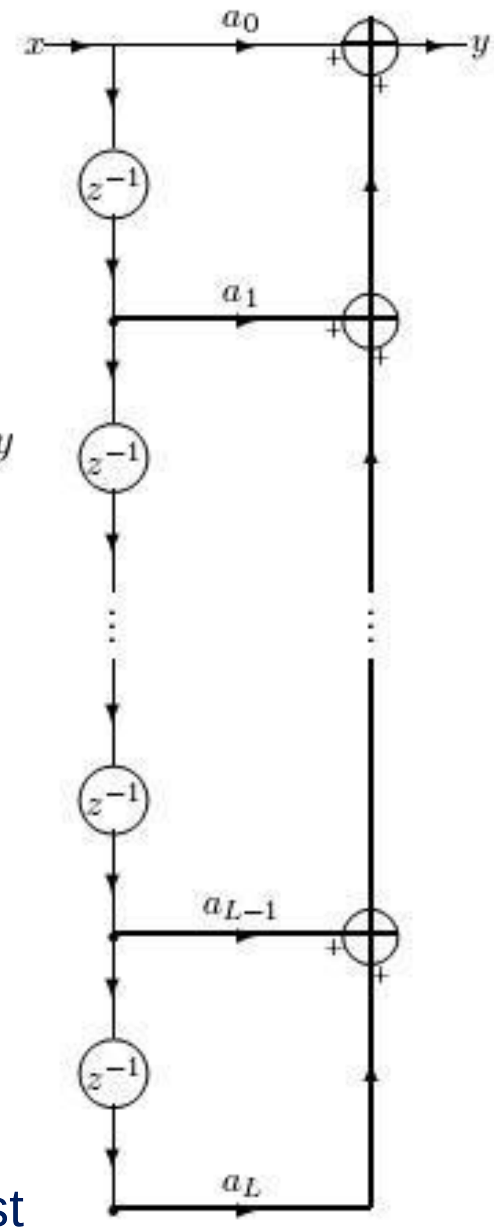
FIFO MACs

So this graph tells us

1. Data structure = FIFO
2. Algorithm = iteration over MA block A

Two ways to MA





Is there any difference between these two ways?

- the 1st way MACs from the new x backward
  $a_0\ x_n + a_1\ x_{n-1} + \ldots + a_L\ x_{n-L}$
- the 2nd way MACs from the earliest x forward
  $a_L\ x_{n-L} + \ldots + a_1\ x_{n-1} + a_0\ x_n$

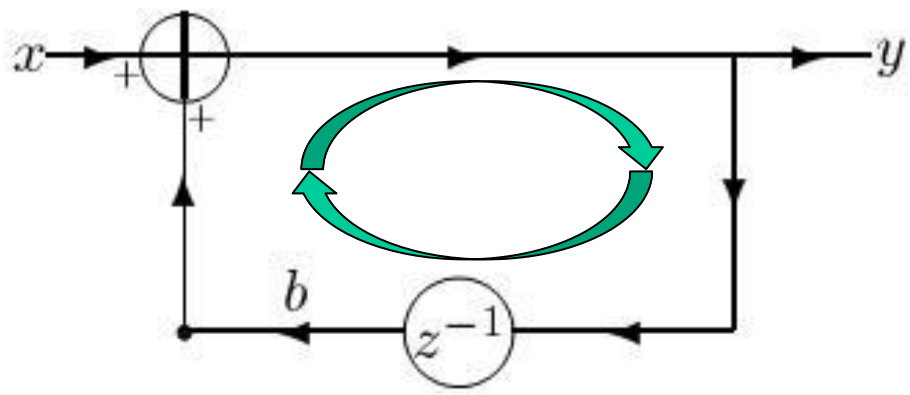
Theoretically there is no difference
(addition is commutative)
but in practice there may be

Given a list of numbers sorted from smallest to largest
which way is most accurate to sum?

Basic AR block

What is the graph for the basic AR filter $y_n = x_n + b y_{n-1}$?

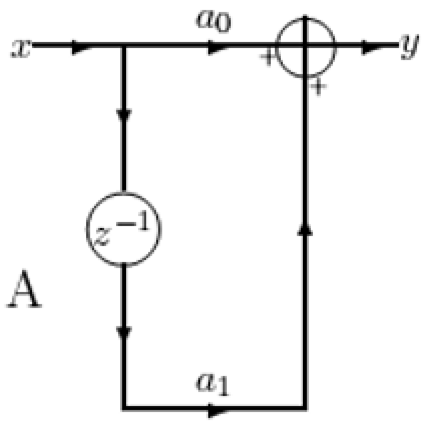
Here is one way:



Note that for the first time we see a *loop* in the graph
in none of the MA filters was there a loop!

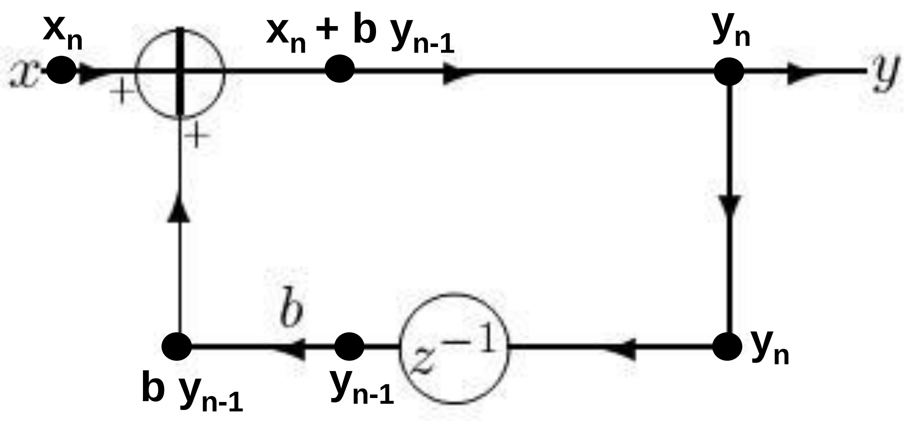
Whenever there is a loop, there is recursion (AR)

Put another way – loops correspond to poles



How does it work?

As usual – let's label points to see why this works



$$y_n = x_n + b y_{n-1}$$

We don't worry about signals from the past influencing the output now

but non-causal loops can be paradoxical (like time travel)

This is just one way the draw the simple AR

there are 4 basic blocks here too

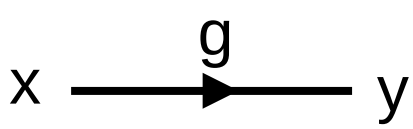
Can you find them?

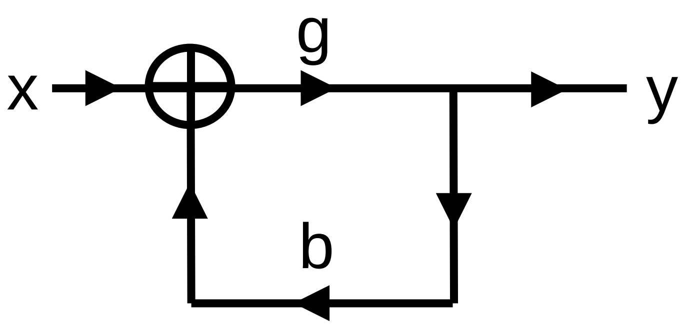
A loop with no delay

It can be useful (but dangerous) to make a loop with no delay

Consider an amplifier

which has some of the output fed back into the input





Then instead of y = g x we have y = g(x+by) or y – bgy = gx

and hence $y = \frac{g}{1-bg} x$

So the feedback increases the amplifier's gain when b < 1/g

but explodes as b → 1/g

We see here the connection between loops and poles!

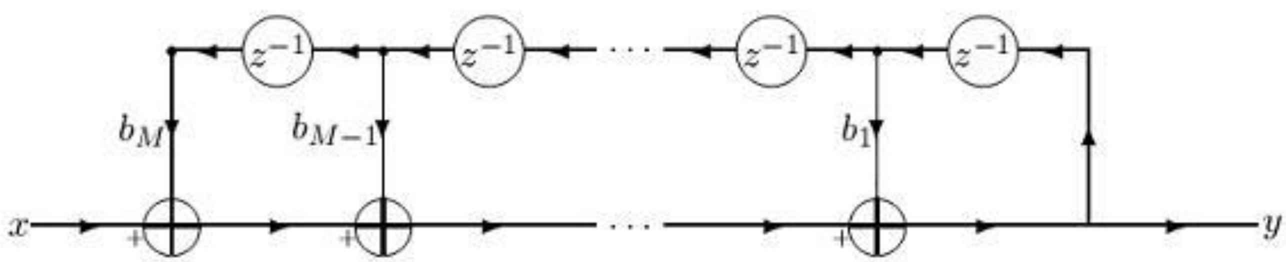
The same thing happens with delay

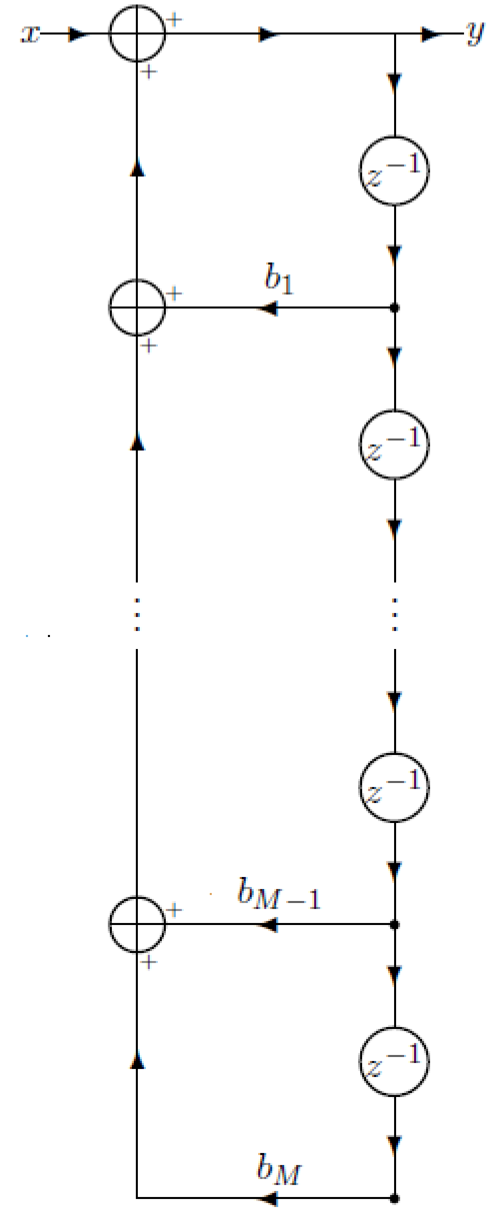
but only for certain frequencies!

General AR filters

Here are two ways to implement the general AR filter

$$y_n = x_n + \sum_{m=1}^{M} b_m y_{n-m}$$





Explain why these indeed implement the AR

Is there any difference between these two ways?

ARMA filters – stage 1

What do we do about ARMA filters?

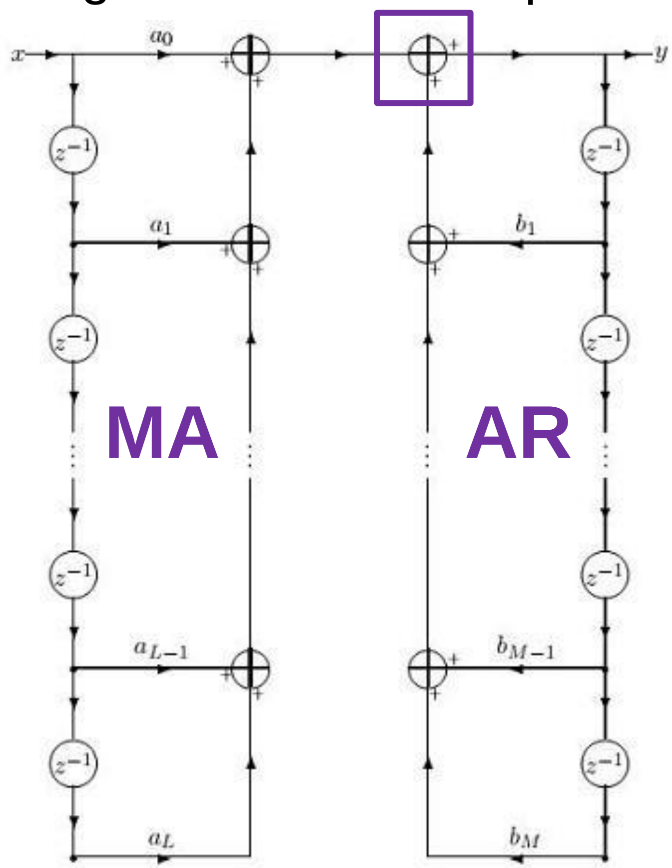
$$y_n = \sum_{l=0}^{L} a_l x_{n-l} + \sum_{m=1}^{M} b_m y_{n-m}$$

The straightforward implementation would be

- perform the MA portion using one of our MA implementations
- perform the AR portion using one of our AR implementations
- add the two together

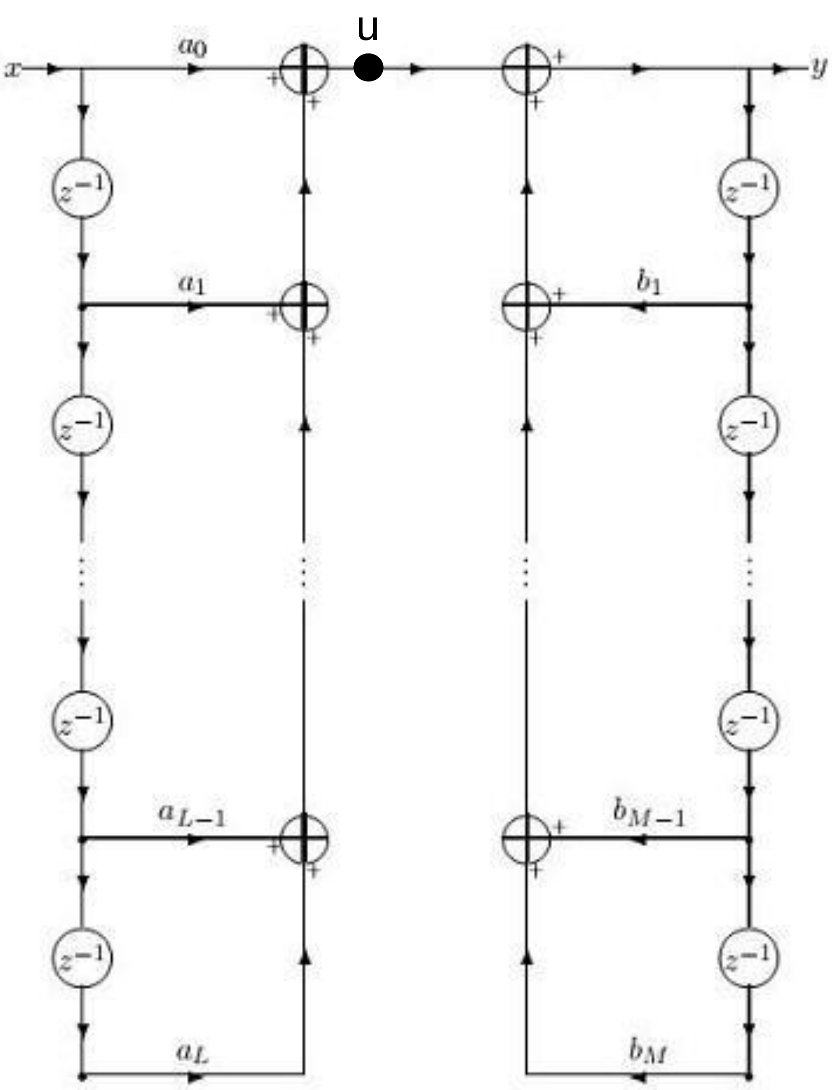
$$y_n = \sum_{l=0}^{L} a_l x_{n-l} + \sum_{m=1}^{M} b_m y_{n-m}$$

MA + AR



How much memory?

$$y_n = \sum_{l=0}^{L} a_l x_{n-l} + \sum_{m=1}^{M} b_m y_{n-m}$$



By observing the graph we see
that L+M memory points are used

Without limiting generality
we can say 2L memory points
and assume L=M

Why? Take max(L,M)
and pad the other with zeros

We will now use graph theory
to reduce the number
of needed memory points

ARMA filters – stage 2

$$y_n = \sum_{l=0}^{L} a_l x_{n-l} + \sum_{m=1}^{M} b_m y_{n-m}$$

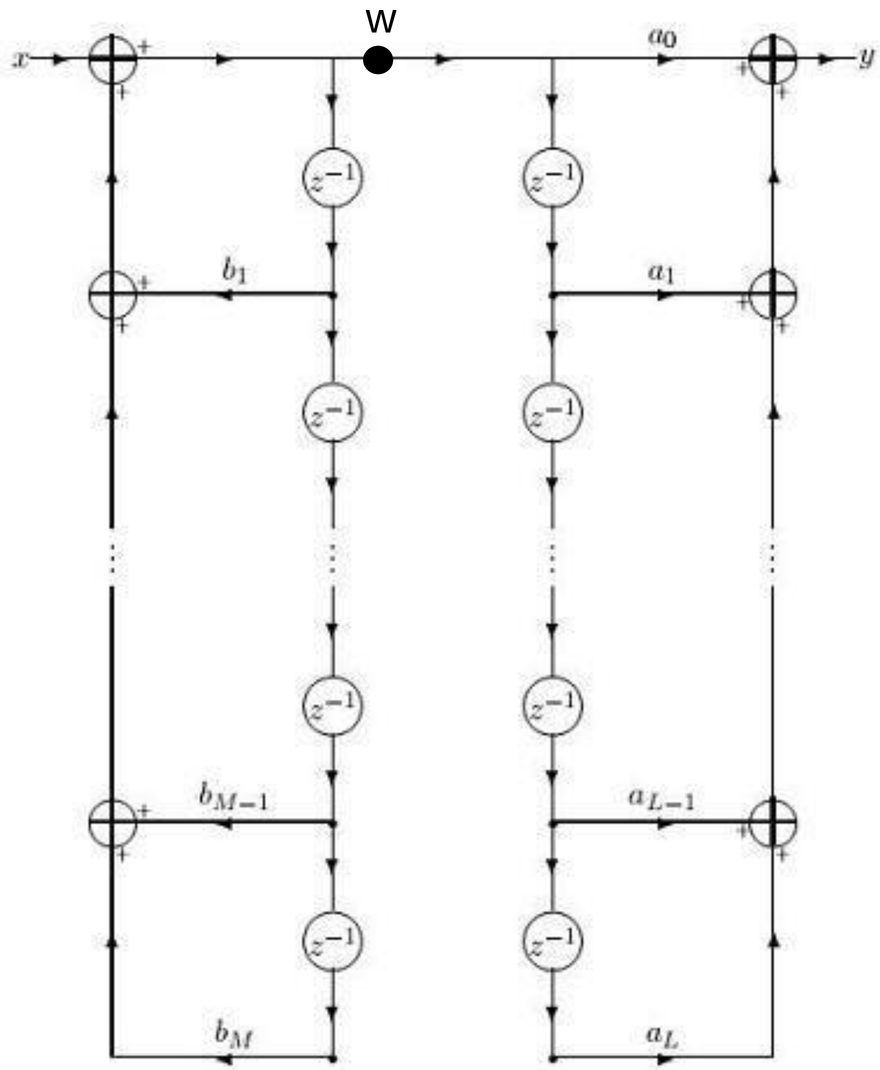
The graph has two filters in series

- 1 MA and 1 AR

Since any 2 filters commute
we can exchange their order

We obtain this new graph

Note that the signal w
between the 2 filters
is different from the signal u !



ARMA filters – stage 3

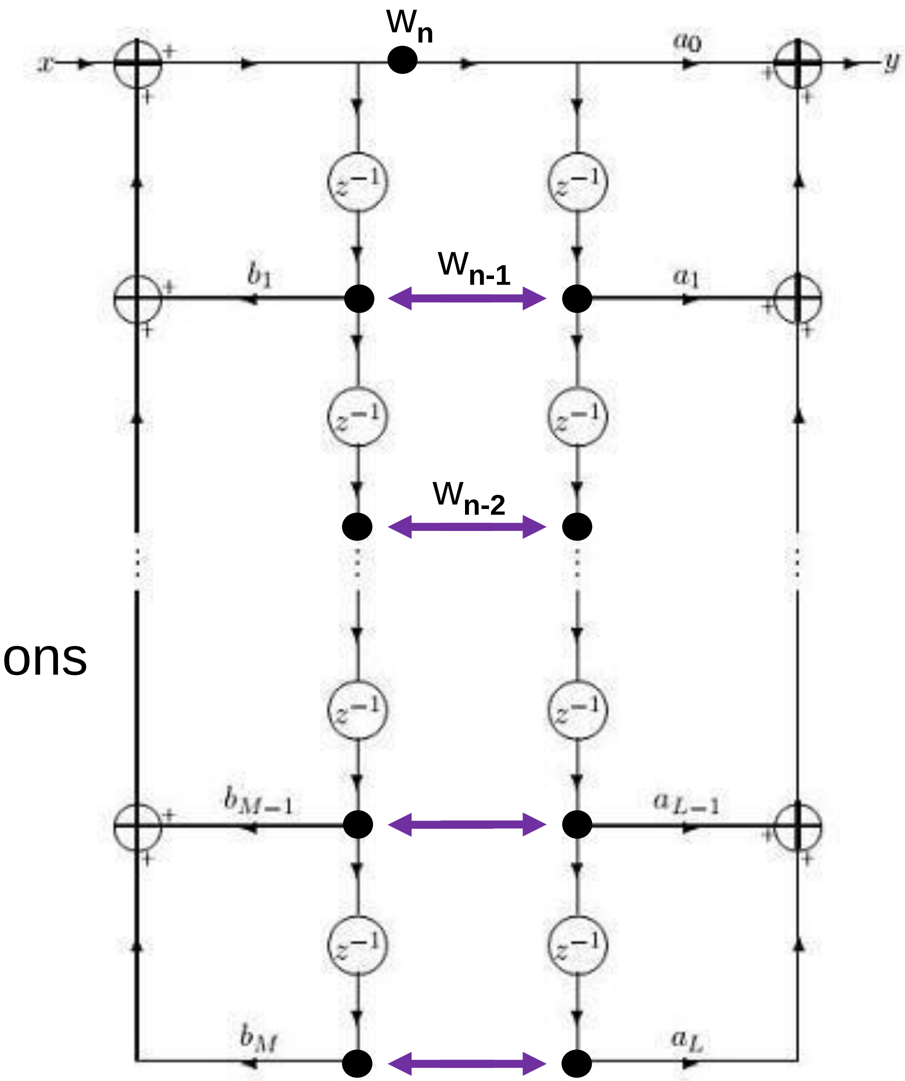
$$y_n = \sum_{l=0}^{L} a_l x_{n-l} + \sum_{m=1}^{M} b_m y_{n-m}$$

We see that there are points representing the same signal !

All of these are

So we can combine the memory locations and remove un-needed delays

This is a new graph transformation

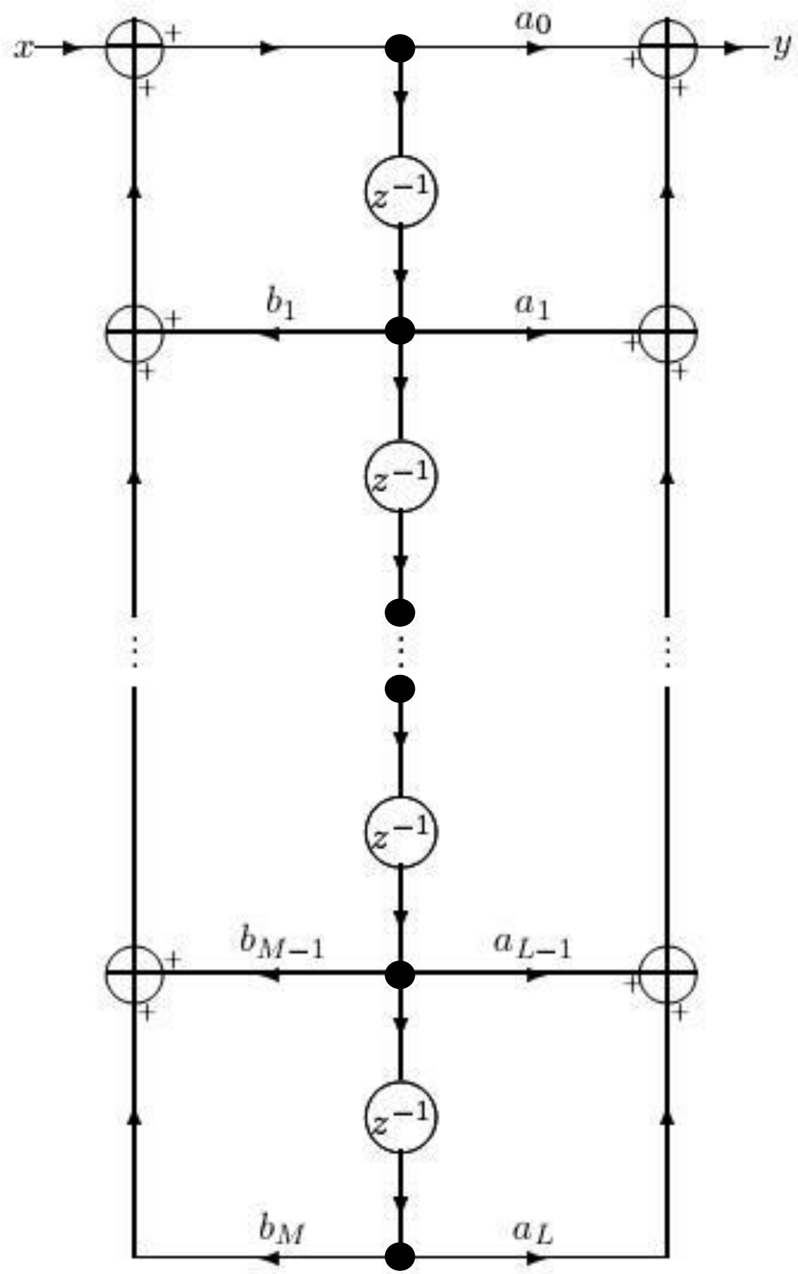


ARMA filters stage 3

$$y_n = \sum_{l=0}^{L} a_l x_{n-l} + \sum_{m=1}^{M} b_m y_{n-m}$$

We now require only L memory points
instead of 2L memory points

A reduction to 50% !



The transposition theorem

Another transformation that creates a new graph

that is equivalent in functionality to the original one

is given by the *Transposition theorem*

This transformation is more complex

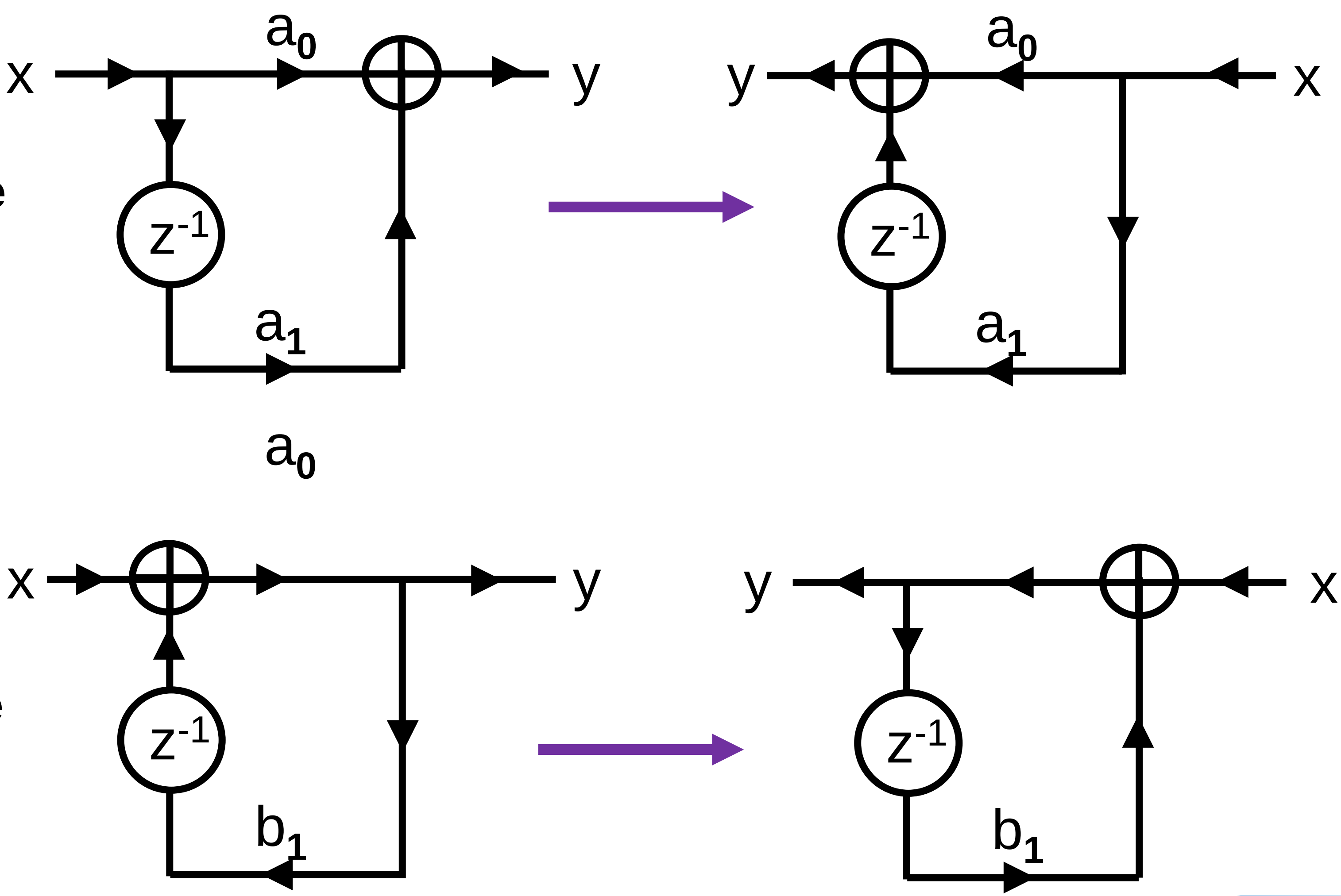
since multiple operations are carried out at the same time

- exchange input(s) and output(s)
- reverse direction of all arrows
- replace adders with splitters (since now 1 in - 2 out)
- replace splitters with adders (since now 2 in - 1 out)

2 simple cases

Simple MA

Simple AR



Summary – the 4 transformations

We have learned 4 basic transformations
that create equivalent signal flow graphs

1. transformations that do not change topology
2. changing order of filters
3. identification of identical signal points
   and removal of redundant branches
4. the transposition theorem

These transformations can be carried out mechanically
and are used to

- reduce the amount of memory needed (we saw such a case!)
- reduce the amount of computation needed (we'll see next time)

This is why graphs are still used in DSP !

Real-time

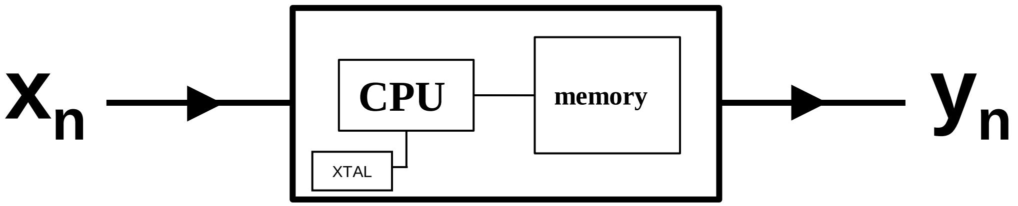
DSP processing is almost always **real-time**

Some exceptions:

- work on recordings
- systems with outputs that are not signals (e.g., detections)

What *is* **real-time** ?

For a signal processing system
    which inputs an input signal one value at a time



- **hard real-time**: ALWAYS finish computing output before next input
- **soft real-time**: finish computing output *on average* before next input
    store input points that arrive before output ready
    exploit some additional delay to output

Example

Assume samples arrive 1000 times per second f_s = 1000 Hz

then the time between samples is t_s = 1 millisecond

So, for hard real-time

all of the processing of a single input sample x_n

in order to produce the output sample y_n

must take place in less than 1 millisecond

(before the next sample x_{n+1} arrives)

For soft real-time

sometimes the processing of a single input sample x_n

can take longer than 1 millisecond

in which case we store the next sample x_{n+1}

until we output the output sample y_n

and then start processing x_{n+1}

DSP = Hard real-time

In DSP we will only deal with **hard** real-time
because we perform exactly the same computations each time
(there are no conditionals)

For example

- MA filters $y_n = \sum_{l=0}^{L} a_l x_{n-l}$
- AR filters $y_n = x_n + \sum_{m=1}^{M} b_m y_{n-m}$ (or ARMA)
- DFT $X_k = \sum_{n=0}^{N-1} x_n \; W_N^{nk}$

So, if we miss a deadline once
it doesn't help to store the next input
since the next input will also take too much time
and the situation will only get worse and worse

Real-time for multi-input

What about the DFT? We can't perform a DFT on one sample x_n
it only makes sense to perform on N samples $x_{0,}\ x_{1,}\ x_{2,}\ \dots\ x_{N-1}$!

For a system which performs calculation on N inputs
hard real-time means that
we must finish processing all N samples $x_{0,}\ x_{1,}\ x_{2,}\ \dots\ x_{N-1}$
before the next N samples $x_{N,}\ x_{N+1,}\ x_{N+2,}\ \dots\ x_{2N-1}$ arrive!

The requirement is the same as before, but *on average*

That is, finishing processing of N old samples
during the time N new samples appear
means on average processing a sample in a sampling time

However, in general the processing of N samples
need not reduce to N processing stages each on 1 sample!

Wrong way to process N samples

You might think that we do the following:

- time 0: input x_0, store in buffer, but don't perform any processing
- time 1: input x_1, store in buffer, but don't perform any processing
- …
- time N-1: input x_{N-1}, store in buffer
  and perform all processing of N samples
  before the next sample x_N arrives
- time N: input x_N but don't perform any processing
- etc.

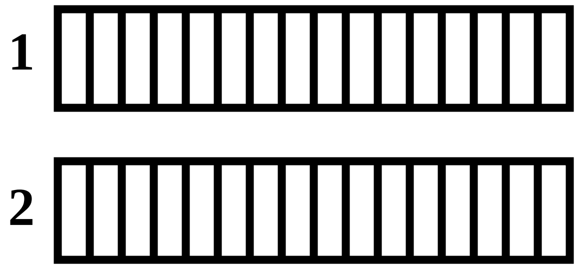
but that would be really hard!

We would need to process N samples in 1 sampling time
although on average we need to process 1 sample per sample time

So, what do we do instead?

Double buffering

double buffer

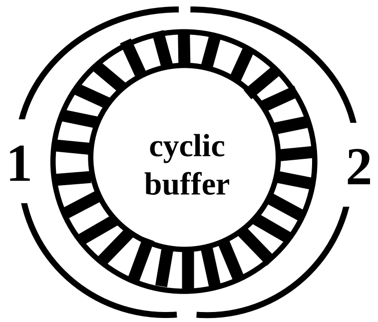


What we do is the following:

- time 0: input x_0 into buffer 1
- time 1: input x_1 into buffer 1
- …
- time N-1: input x_{N-1} into buffer 1 (filling buffer)
  and start performing all processing of N samples in buffer 1
- time N: input x_N into buffer 2 and continue processing buffer 1
- time N+1: input x_{N+1} into buffer 2 and continue processing buffer 1)
- some time before time 2N-1: finish processing buffer 1 and output
- time 2N-1: input x_{2N-1} into buffer 1
  and start performing all processing of N samples in buffer 2

Trick:

Instead of having to swap write pointers
between buffer 1 and buffer 2
we can use a *cyclic buffer*



Theorem for real-time

The computational complexity of a real-time system
that performs calculation on N inputs
must not exceed O(N)

In particular the DFT can not be performed in real-time
since $X_k = \sum_{n=0}^{N-1} x_n \, W_N^{nk}$
requires computing N values X_0, X_1, …, X_{N-1}
each of which requires N multiplications (n = 0 … N-1)
and is thus $O(N^2)$

What does this theorem mean?

Why can't we find a fast enough processor
to perform anything we want in real-time?

The meaning of the theorem

Imagine that you need to program some $O(N^2)$ process
and as before samples arrive every millisecond

Let's assume that you are told that N=1024
and that you manage to program you CPU
to finish the processing in less than 1024 milliseconds

But then it turns out that N=2048 is really needed
You now have twice the time to perform the computation – 2048 ms
but because of $O(N^2)$ you require 4 times the time – 4096 ms!

So you buy a faster processor and manage to run in real-time
but then if it turns out that N=4096 is needed
no strong enough CPU is available!

But if the complexity is $O(N)$
then when N is increased from 1024 to 2048
you have twice the time to perform the computation
but only need twice the time!

and the solution is …

DFT and iDFT are so critical in DSP
that without a real-time implementation DSP won't work!

The Fast Fourier Transform
reduces the $O(N^2)$ complexity of the straightforward DFT
to $O(N \log N)$

Note we don't need to specify the base of the log
since changing base only inserts a multiplicative constant
but we will always assume $\log_2$

But $O(N \log N)$ is higher than $O(N)$
and so violates the theorem that real-time requires $O(N)$!

$O(N \log N)$ is not low enough to guarantee real-time for all N
but is sufficiently low to enable even extremely large Ns

DSP processors are rated by
how large an FFT they can perform in real-time!

Warm-up problem #1

Find minimum and maximum of N numbers $x_0\ x_1\ x_2\ x_3 \quad \ldots \quad x_{N-2}\ x_{N-1}$

- minimum alone takes N comparisons *prove* this
- maximum alone takes N comparisons *prove* this

So we can certainly find both with 2N comparisons

But there is a way to find both in 1½ N comparisons

$$\underbrace{x_0\ \ x_1}\ \underbrace{x_2\ \ x_3}\ \ \ldots\ \ \underbrace{x_{N-2}\ \ x_{N-1}}$$

smaller	x_0	x_3	…	x_{N-1}
larger	x_1	x_2	…	x_{N-2}

for example

- run over at pairs, separating into *larger* and *smaller*
  – this takes ½ N comparisons
- the minimum *must* be in the *smaller* list (why?)
  – find it in ½ N comparisons
- the maximum must be in the larger list
  – find it in ½ N comparisons
- altogether 3/2 N comparisons – 25% savings

Can we improve this by further decimation? Why not?

2 remarks

- this method uses ***decimation***
  that is separating a sequence of N elements
  into two subsequences of N/2 elements
  based on even and odd elements

- although we reserved 2 buffers

smaller	x_0	x_3	...	x_{N-1}
larger	x_1	x_2	...	x_{N-2}

the calculation can be performed *in-place*
that is, without additional memory !

$x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad \dots \quad x_{N-2} \quad x_{N-1}$

$x_0 \quad x_1 \quad x_3 \quad x_2 \quad x_5 \quad x_4 \quad \dots \quad x_{N-2} \quad x_{N-1}$

But to swap two values x_0 x_1 we *do* need an additional memory

$y \leftarrow x_0 \ , \ x_0 \leftarrow x_1 \ , \ x_1 \leftarrow y$

why don't we count this?

Warm-up problem #2

Multiply two N digit numbers A and B (w.o.l.g. N binary digits)

we saw that long multiplication is a convolution

and thus takes $2N^2$ 1-digit multiplications

But there is a faster way!

Partition the binary representation of A and B into 2 parts

A7 A6 A5 A4 A3 A2 A1 A0 = A_L A_R

B7 B6 B5 A4 B3 B2 B1 B0 = B_L B_R

Now

$$\begin{aligned} A &= A_L 2^{\frac{N}{2}} + A_R \\ B &= B_L 2^{\frac{N}{2}} + B_R \\ C &= A_L B_L 2^N + (A_L B_R + A_R B_L) 2^{\frac{N}{2}} + A_R B_R \\ &= A_L B_L (2^N + 2^{\frac{N}{2}}) + (A_L - A_R)(B_R - B_L) 2^{\frac{N}{2}} + A_R B_R (2^{\frac{N}{2}} + 1) \end{aligned}$$

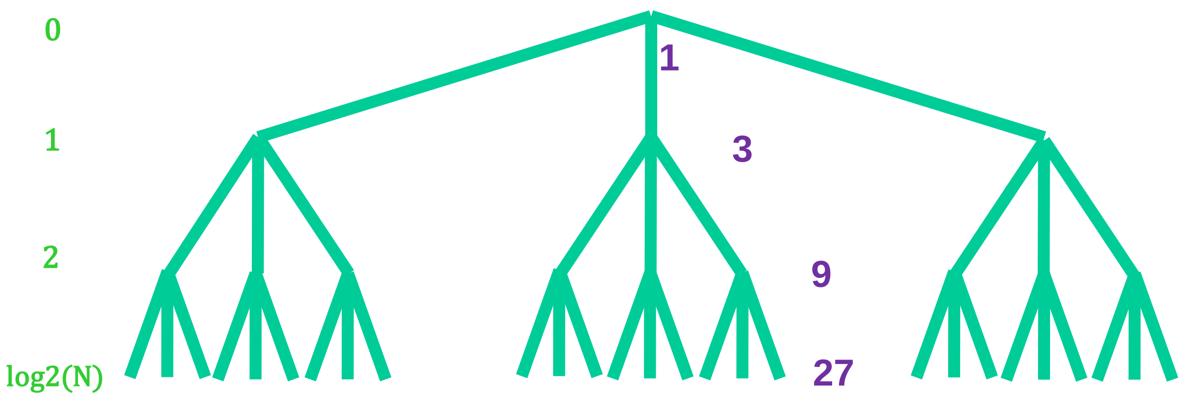
So partitioning factors reduces to $3/4\ N^2$ saving 25% !

There's a small problem here – the subtractions might add a bit!

But $O(3/4\ N^2) = O(N^2)$ so we haven't change the O complexity!

Continued …

But this time we *can* continue



3 multiplications, each N/2 bits

3^2 multiplications, each N/4 bits

$3^{\log_2(N)}$ multiplications, each 1 bit

Now, $3^{\log_2(N)} = N^{\log_2 3}$ Why is $a^{\log_k(b)} = b^{\log_k a}$?

So, the complexity is $O(N^{\log_2 3}) \approx O(N^{1.585})$

and $O(N^{1.585}) < O(N^2)$ -- the O complexity has been reduced!

This is the **Toom-Cook (Karatsuba)** algorithm

which was thought to be the fastest way to multiply

until the FFT way was discovered

Toom-Cook example

Let's multiply A=83 times B=122 using Toom Cook

(the answer is 10,126)

A = 0 1 0 1 0 0 1 1 B = 0 1 1 1 1 0 1 0
A_L = 0 1 0 1 = 5 B_L = 0 1 1 1 = 7
A_R = 0 0 1 1 = 3 B_R = 1 0 1 0 = 10

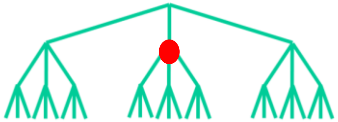
AR = $A_L B_L(2^N + 2^{\frac{N}{2}}) + (A_L - A_R)(B_R - B_L)2^{\frac{N}{2}} + A_R B_R(2^{\frac{N}{2}} + 1)$
= **5*7** *(256+16) + **(5-3)** ***** **(10-7)** *16 + **3*10***(16+1)
= **35** * 272 + **6** * 16 + **30** * 17
= 35 *shl 8* + (35 + 6 + 30) *shl 4* + 30 [3 N/2-bit products + 2 shifts + 4 adds]

Now we repeat the process for $A_L B_L$
A_L = 0 1 0 1 B_L = 0 1 1 1
A_{LL} = 0 1 = 1 B_{LL} = 0 1 = 1
A_{LR} = 0 1 = 1 B_{LR} = 1 1 = 3

$A_L B_L$ = 1*1*(16+4) + (1-1)*(3-1)*4 + 1*3*(4+1) **= 20 + 0 + 15 = 35**

and the same for the other 2 multiplications

Toom-Cook example (cont.)



$(A_L - A_R)(B_R - B_L) = 2 * 3 = 0010 * 0011 =$

$0*0*(16+4) + (0-2)*(3-0)*4 + 2*3*(4+1)$ **= 0 + -24 + 30 = 6**

$A_R B_R = 3 * 10 = 0011 * 1010 =$

$0*2*(16+4) + (0-3)*(2-2)*4 + 3*2*(4+1)$ **= 0 + 0 + 30 = 30**

Finally, we go one step further to 9 individual bit multiplications, e.g.,

$A_{LR} * B_{LR} = 1 * 3 = 3$

$A_{LRL} = 0 \quad A_{LRR} = 1 \qquad B_{LRL} = 1 \quad B_{LRR} = 1$

$A_{LR} B_{LR} = A_{LRL} B_{LRL} (4+2) + (A_{LRL} - A_{LRR})(B_{LRR} - B_{LRL}) + A_{LRR} B_{LRR} (2+1)$

$= 0 * 1 * 6 + -1 * 0 + 1 * 1 * 3$

and similarly for all the others

Decimation and Partition

The two warm-up problems had a strategy in common

- If the complexity is $C=cN^2$
  then it is worthwhile to divide the input sequence into 2 subsequences
- Since performing the operation on each part costs $c(N/2)^2 = C/4$
  so the two together cost $C/2$
- If we can *glue* the two parts back together in less than $C/2$
  then we have a more efficient algorithm!

But the two problems used two different methods of dividing the sequence

$$x_0\ x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7$$

Decimation

$x_0\ x_2\ x_4\ x_6$ EVEN

$x_1\ x_3\ x_5\ x_7$ ODD

LSB sort

Partition

$x_0\ x_1\ x_2\ x_3$ LEFT

$x_4\ x_5\ x_6\ x_7$ RIGHT

MSB sort

Radix 2

In both warm-up problems, and in the FFT algorithm we will derive
we divide up the sequence into 2 sub-sequences of length N/2

In fact we will require that N be a power of 2
so that we can continue to divide by 2 until we get to units

Such algorithms are called **radix-2** FFT algorithms

We could also chose to divide it up into 3 subsequences
or 4 or 5 or any other integer into which N factors

There are special FFT algorithms for powers of other primes
and for semi-primes like N=15=5*3

In fact, only for prime N is no possibility of reducing complexity

Shmuel Winograd discovered FFTs with few multiplications
for various values of $N = N_1 * N_2$ where N_1 and N_2 are coprime

Decimation in Time $\Leftrightarrow$ Partition in Frequency

What does decimating a signal in the time domain
do to the frequency domain representation ?

Assume that the original signal $x_0\ x_1\ x_2 \ldots x_{N-1}$
was sampled at $f_\mathbf{s}$
and thus by the sampling theorem
have maximum frequency $f_\mathbf{N} = f_\mathbf{s}/2$

Then the decimated signals, $x_0\ x_2\ x_4 \ldots$ and $x_1\ x_3\ x_5 \ldots$
are sampled at $f_\mathbf{s}/2$
and thus have maximum frequency $f_\mathbf{s}/4$

So we obtain only the lower ½ of the original spectral width
in other words the LEFT partition of the spectrum

Thus DIT = PIF

We'll see later the exact relationship between
the lower and upper partitions of the spectrum

Partition in Time ⇔ Decimation in Frequency

What does partitioning a signal in the time domain
do to the frequency domain representation ?

Assume that the original signal $x_0\ x_1\ x_2\ \ldots\ x_{N-1}$
was sampled at f_s and thus has duration $T = N\ t_s = N / f_s$

Then the partitioned signals, $x_0\ x_1\ \ldots\ x_{N/2-1}$ and $x_{N/2}\ x_{N/2+2}\ \ldots\ x_{N-1}$
have duration $N/2\ t_s = T/2$

According to the uncertainty principle
if the time duration Δt is reduced by ½
then the frequency uncertainty $\Delta\omega$ is increased by 2
(the frequency resolution is *blurred*)

So, we can effectively observe only every other spectral line!

Thus PIT = DIF

FFT history

The FFT has been discovered many times

perhaps as early as unpublished 1805 work by Gauss

which predates Fourier!

In 1903 Runge discovered an FFT for N a power of 2

and in 1942 Danielson and Lanczos discovered a O(N log N) DFT

However, credit is now usually given to

- John Wilder Tukey – American mathematician/statistician (Princeton)
  - who coined the words *bit* = binary digit and *software*
- James William Cooley – American mathematician / programmer (IBM)

who published in 1965 (in order to avoid patenting)

The Cooley-Tukey algorithm is **D**ecimation **I**n **T**ime

that is, it decimates the signal in the time domain

performing DFTs separately of the evens and the odds

The Sande-Tukey algorithm is Decimation in Frequency

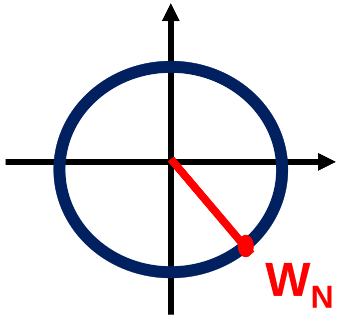
that is, it partitions the signal in the time domain

performing DFTs separately of the 1^{st} half and 2^{nd} half

Before starting

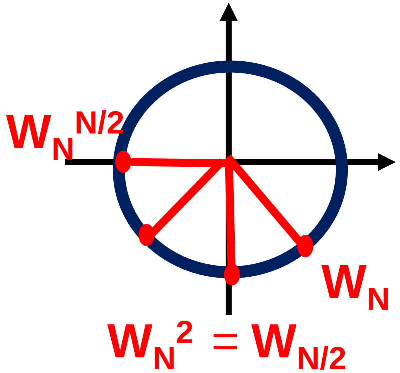
Recall that the DFT is $X_k = \sum_{n=0}^{N-1} x_n \; W_N^{nk}$

where W_N is the N^{th} root of unity $W_N = e^{-i\frac{2\pi}{N}}$



We will need three *trigonometric identities*

1. $W_N^N = 1$ (that's the definition!)
2. $W_N^{N/2} = -1$ $(e^{-i\frac{2\pi N}{N\ 2}} = e^{-i\pi} = -1$ or $)$
3. $W_N^2 = W_{N/2}$ $(e^{-i\frac{2\pi}{N}2} = e^{-i\frac{2\pi}{N/2}}$ or $)$



They are *trigonometric* identities

since $W_N = \cos(\frac{2\pi}{N}) - i \sin(\frac{2\pi}{N})$

DIT (Cooley-Tukey) FFT

Let's derive the **radix-2 DIT** FFT algorithm!

We start by decimating the formula for the DFT

that is, we separate the even terms 2n from the odd terms 2n+1

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} = \sum_{n=0}^{\frac{N}{2}-1} \left(x_{2n} W_N^{2nk} + x_{2n+1} W_N^{(2n+1)k} \right)$$

3rd identity

Now, $W_N^{2nk} = W_{N/2}^{nk}$ and $W_N^{(2n+1)k} = W_N^{k}\ W_N^{2nk}$ and so we can rewrite:

$$X_k = \underbrace{\sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_{\frac{N}{2}}^{nk}}_{\text{DFT of evens}} + W_N^k \underbrace{\sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_{\frac{N}{2}}^{nk}}_{\text{DFT of odds}}$$

DIT – the first step

So we have found

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} \overset{\text{Even}}{x_n^E} W_{\frac{N}{2}}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} \overset{\text{Odd}}{x_n^O} W_{\frac{N}{2}}^{nk}$$

which shows that the DFT indeed divides up into 2 half-sized DFTs
and an additional N multiplications by W_N^k (for k=0…N-1)

This is encouraging, since the *glue* is O(N) !

The glue factor is usually called the *twiddle factor*
and it is the entire difference between the contribution
of the two decimations to the original DFT

Note that we precompute and store the N twiddle factors W_N^k (k=0 … N-1)
and don't have to compute them over and over again!

PIF

The next step is to exploit the relationship between

Decimation In Time and Partition In Frequency

What is the connection between

Xk in the left partition : $0 \le k \le N/2 - 1$

and the corresponding component in the right partition

Xk in the right partition : $N/2 \le k \le N - 1$

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk}$$

$$X_{k+\frac{N}{2}} = \sum_{n=0}^{N-1} x_n W_N^{nk} W_N^{\frac{Nn}{2}}$$

$$= \sum_{n=0}^{N-1} x_n W_N^{nk} (-1)^n$$

2nd identity
$W_N^{N/2} = -1$

Note that we compute exactly the same products

but add them with different signs $+ - + - + - + -$

DIT is PIF

So, we have already reduced the number of multiplications by ½

Now, the products for which $(-1)^n$ is negative are odd n

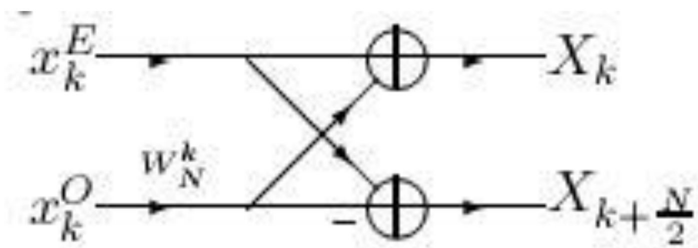
i.e., exactly those terms in the odd decimation!

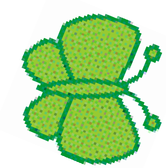
So

$$X_{k+\frac{N}{2}} = \sum_{n=0}^{\frac{N}{2}-1} x_n^E W_{\frac{N}{2}}^{nk} - W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_n^O W_{\frac{N}{2}}^{nk}$$

We can draw this as a DSP diagram in a nice *in-place* way !

x_k^E is the DFT sum
don't confuse it with x_n^E !
(it should have an intermediate sized x …)





This reminds us of the N=2 butterfly

but has a twiddle factor before the butterfly

The DIF algorithm/s butterfly

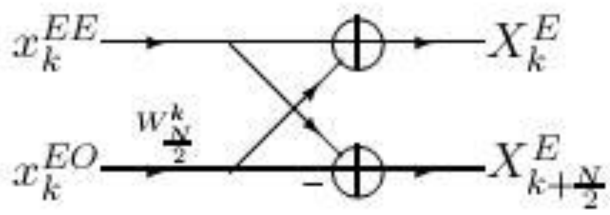
has a twiddle factor after the butterfly

DIT all the way

We have already saved a factor of 2 in the multiplications

but we needn't stop after splitting the original sequence in two !

Each half-length sub-sequence can be decimated again

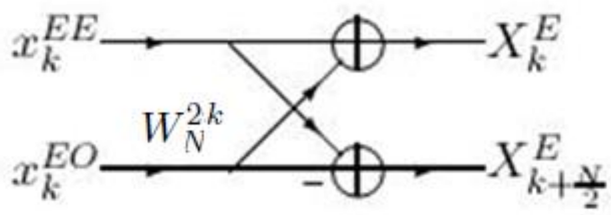


note that this is in-place!

Assuming that N is a power of 2

we continue decimating until we get to the basic N=2 butterfly

Note that since $W_{N/2} = W_N^2$ we can draw this



and we only have to keep one table of W_N^k

Let's continue!

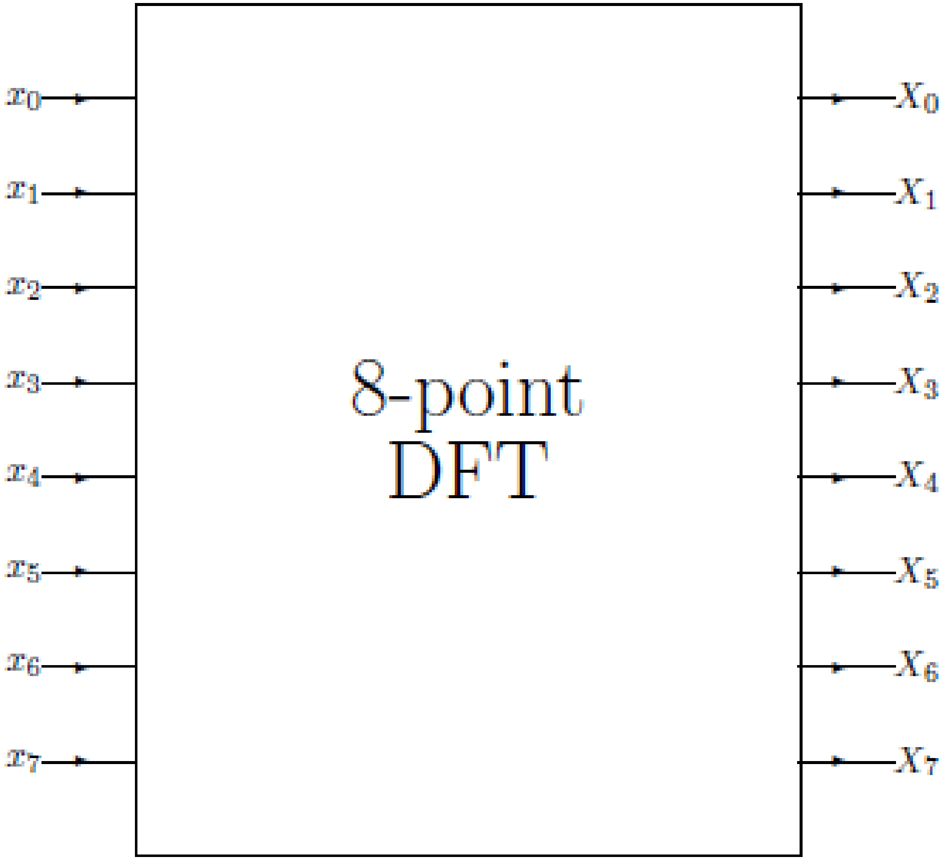
Instead of explicitly writing equations for the next step
it is easier to do everything graphically

In order to make things simple, we'll assume N=8
and explicitly draw out all the steps

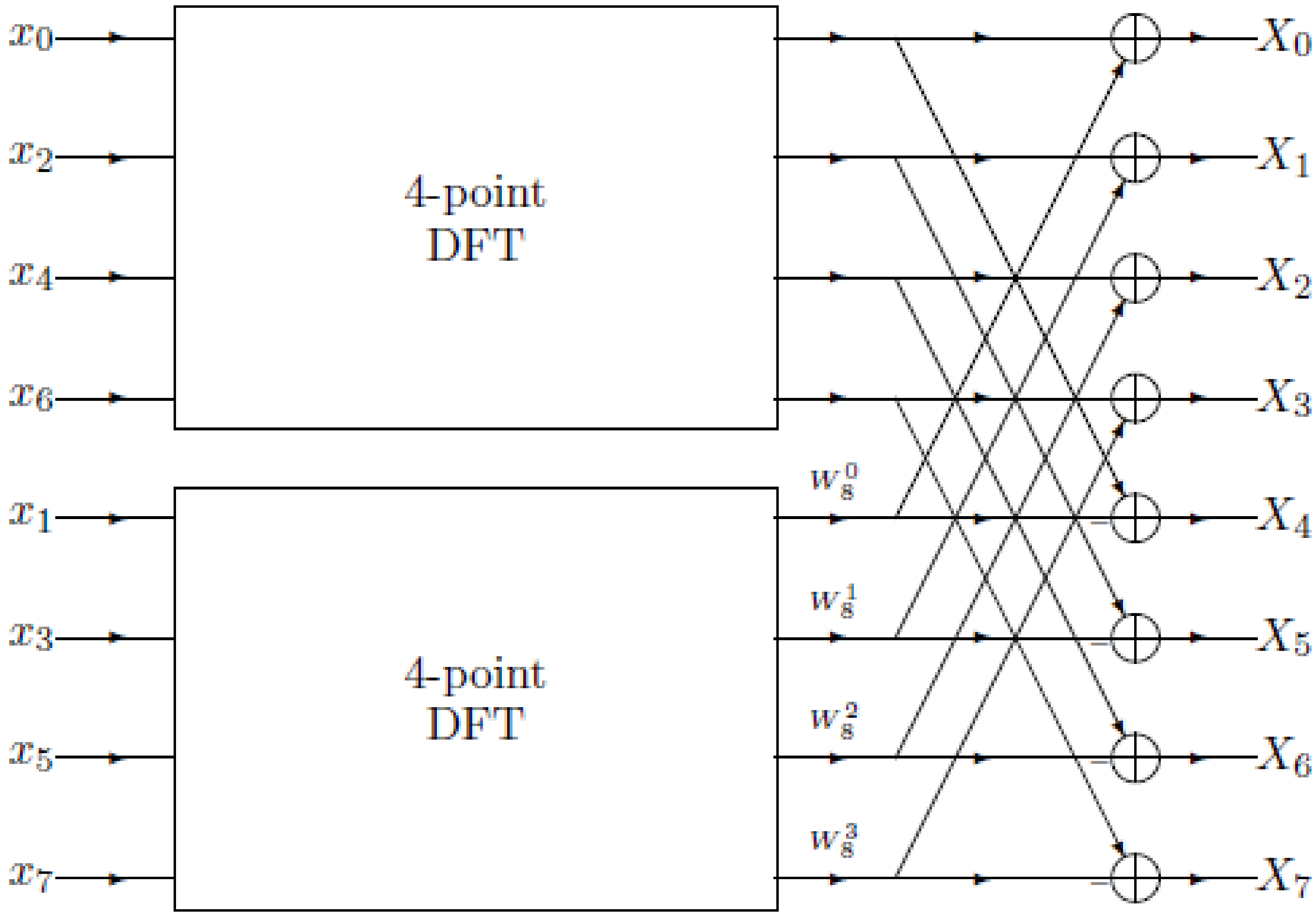
- decimate the N=8 sequence into two subsequences of length 4
- decimate each of the sub-sequences of length 4
  into two sub-sub-sequences of length 2 (4 altogether)
- perform four basic N=2 butterflies

Let's see this happen!

DIT N=8 - step 0

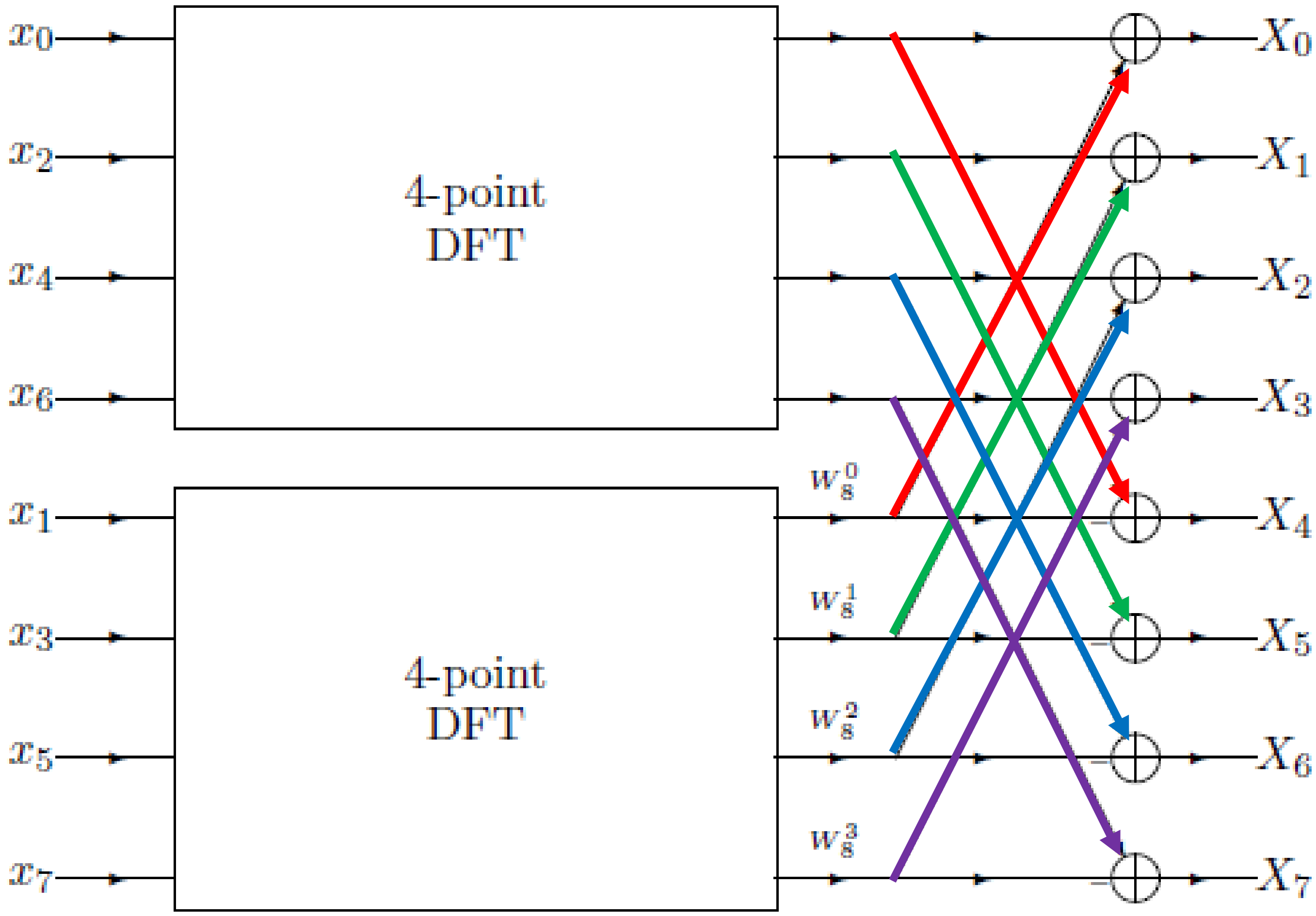


DIT N=8 - step 1



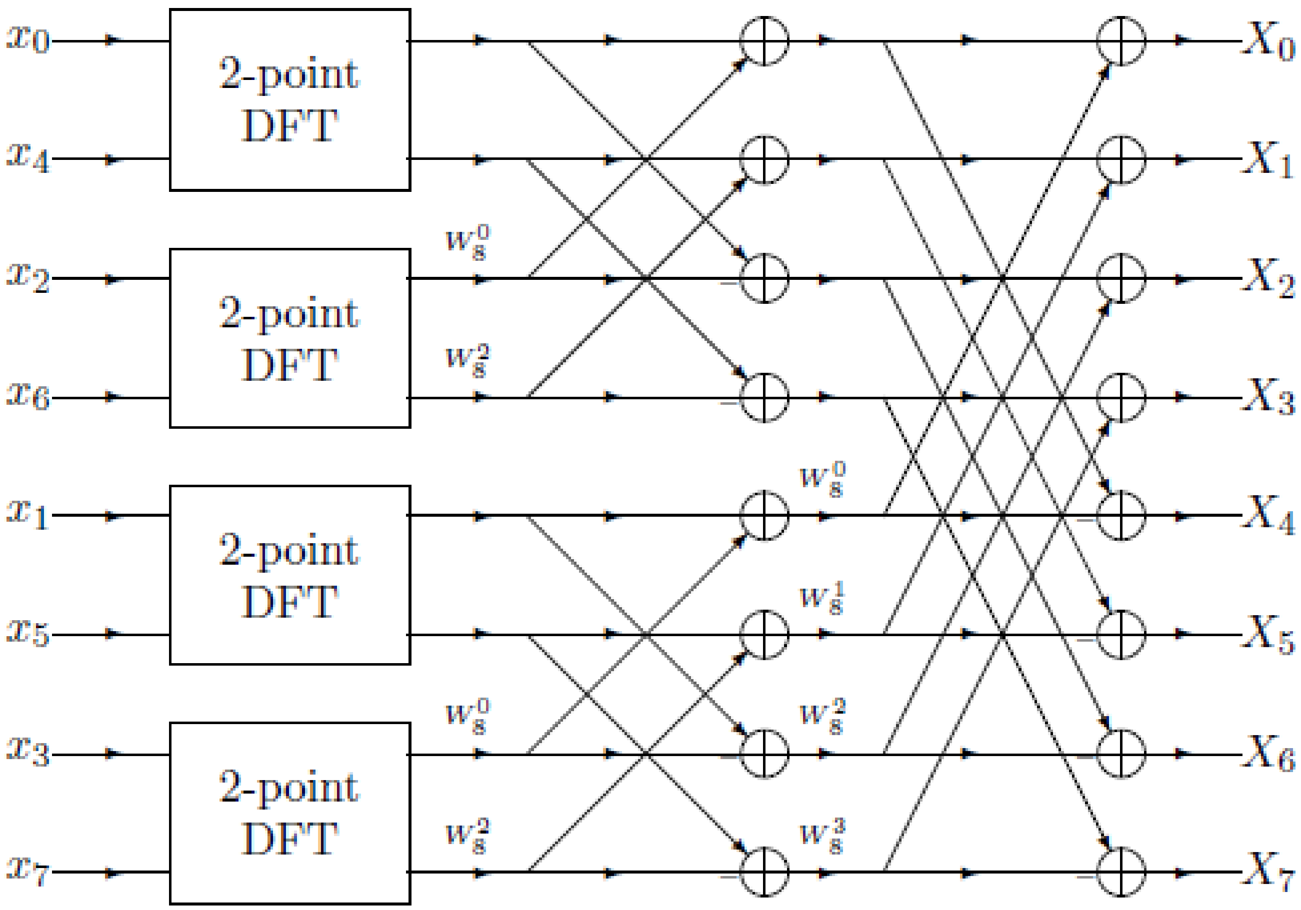
in-place!

DIT N=8 - step 1 : 4 butterflies



The butterflies are all entangled – do you see them?

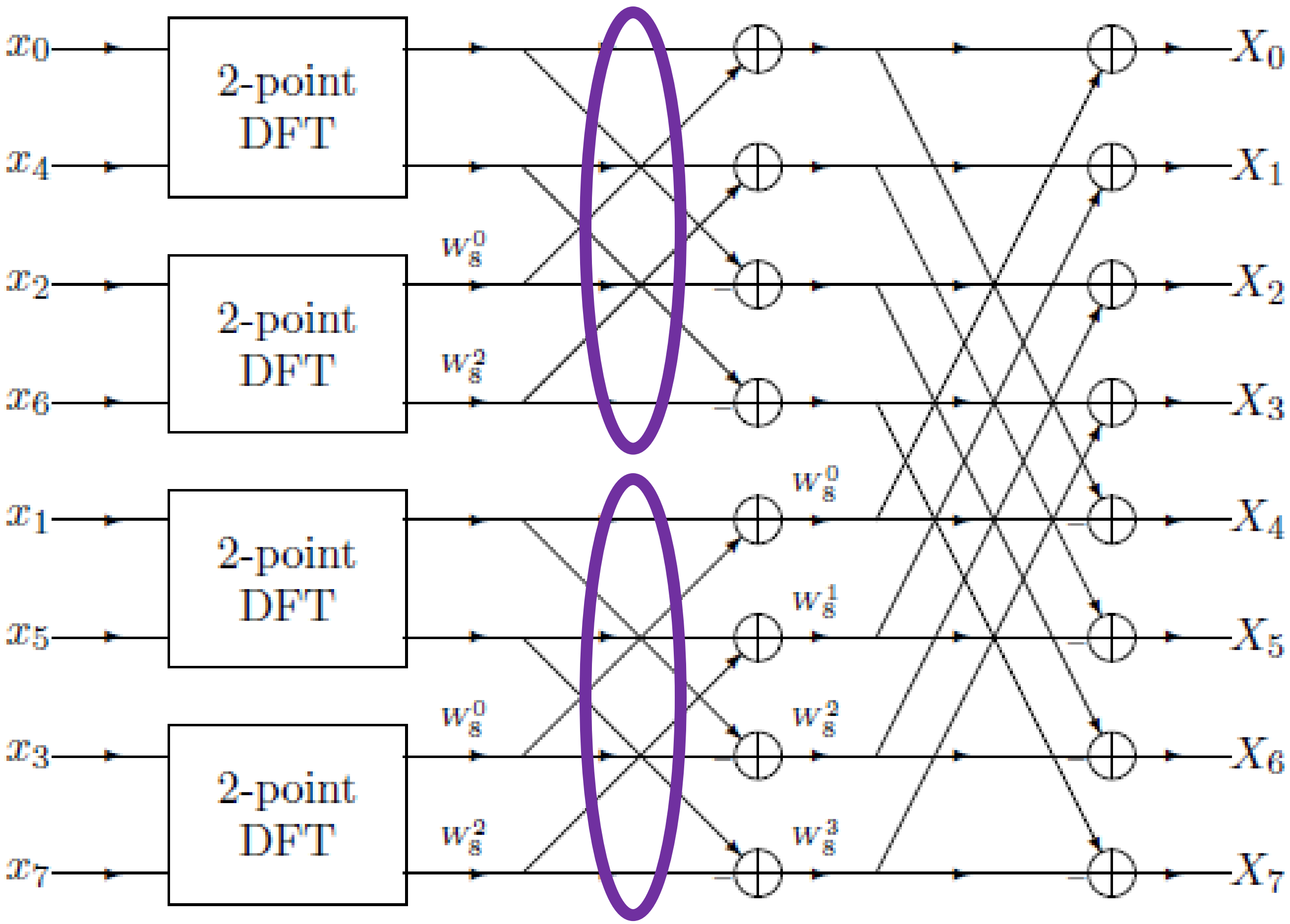
DIT N=8 - step 2



$W_4^0 = W_8^0$

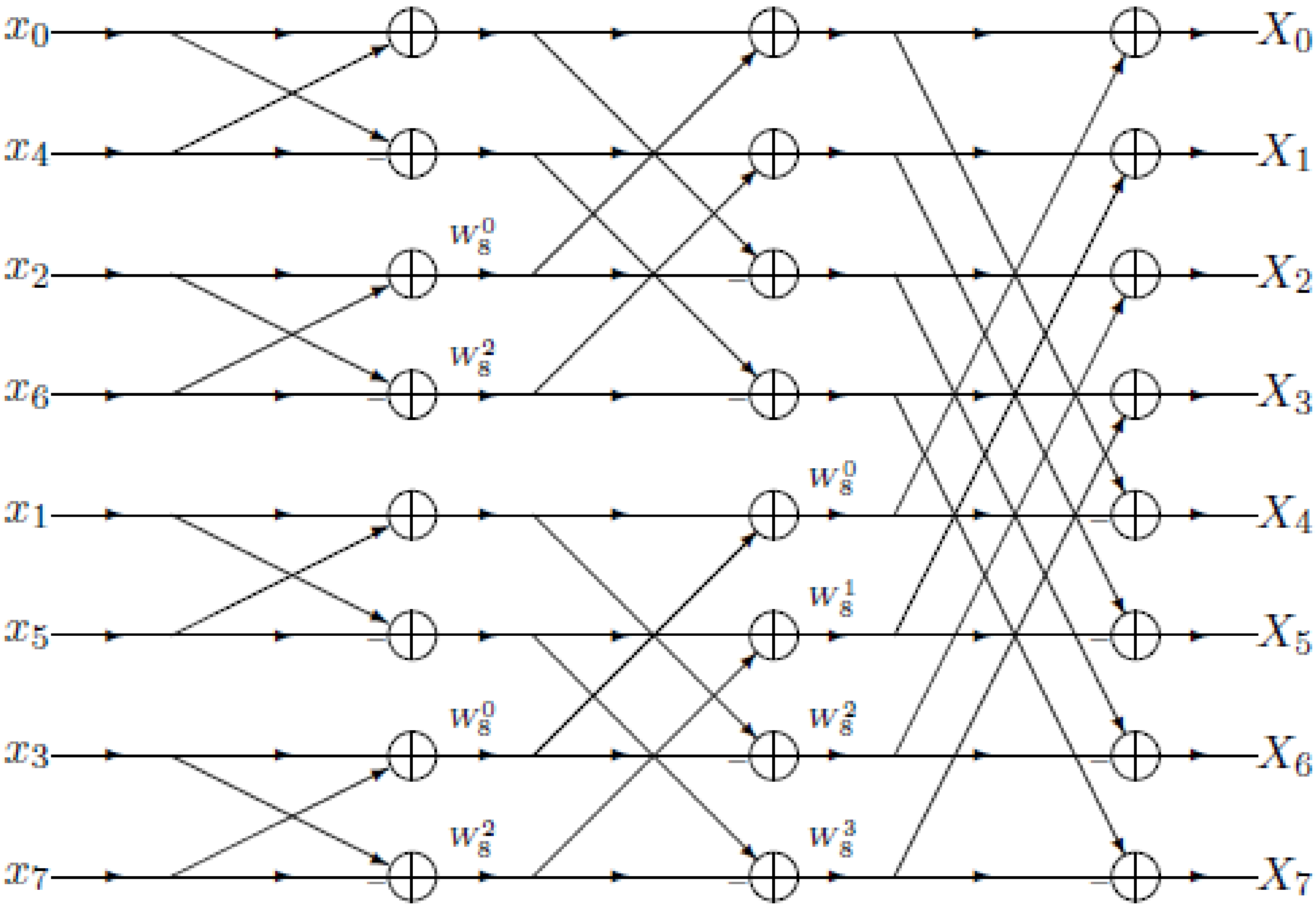
$W_4^1 = W_8^2$

DIT N=8 - step 2



Note that the second stage butterflies are less entangled!

DIT N=8 - step 3



x_0
x_4
x_2
x_6
x_1
x_5
x_3
x_7
W_8^0
W_8^2
W_8^0
W_8^2
W_8^0
W_8^1
W_8^2
W_8^3
X_0
X_1
X_2
X_3
X_4
X_5
X_6
X_7

Complexity

An FFT of length N has

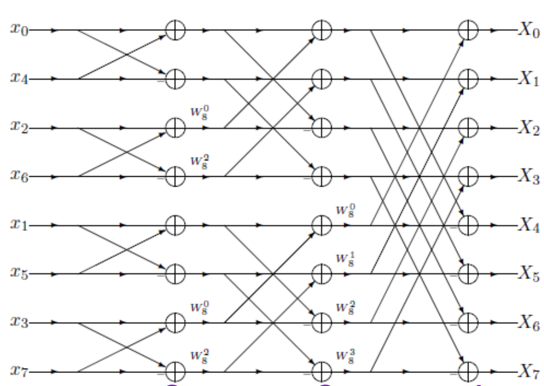
- $\log_2(N)$ **stages** of butterflies
- there are ½N butterflies in each stage, each with
  - 1 complex multiply
  - 2 complex adds (1 add and 1 subtract)

So there are :

- ½ N $\log_2(N)$ complex multiplications
- N $\log_2(N)$ complex additions

Which is why we say that the complexity is O(N log N)

for N=8 there are 3 stages
Stage 1:   4 butterflies
Stage 2: 2*2 butterflies
Stage 3: 4*1 butterflies



stage 3   stage 2   stage 1

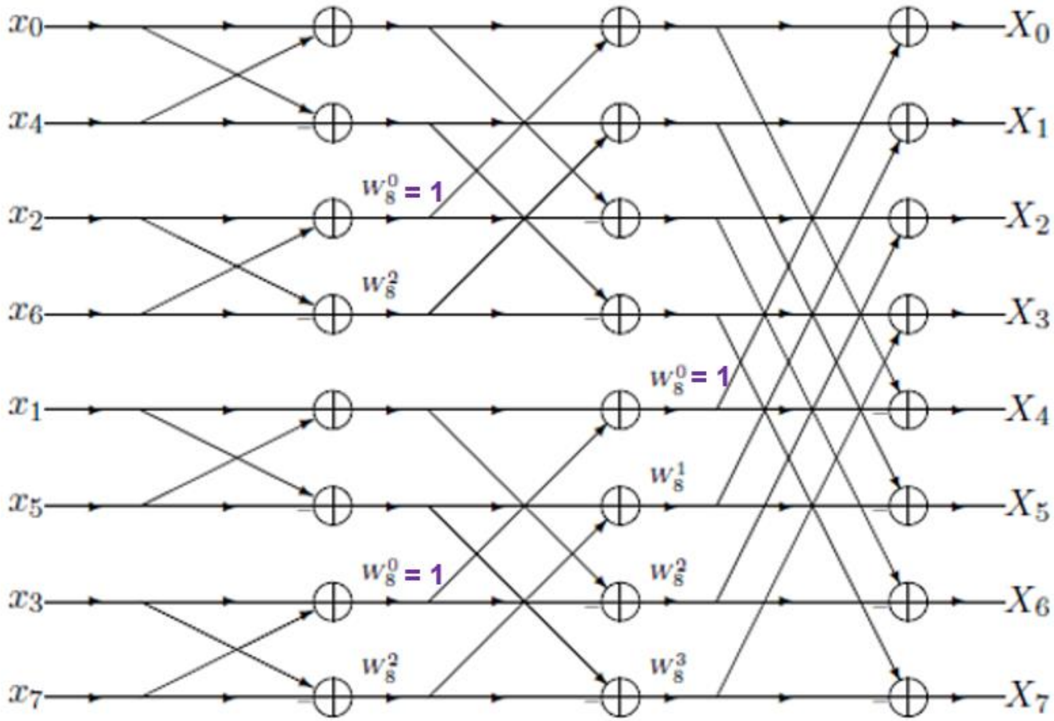
Well, its even a bit less

Actually, some of the multiplications are trivial!

- the first stage has one trivial multiplication ($W_N^0=1$)
- the 2nd stage has 2 trivial multiplications
- …
- the last stage has no true multiplications (it has N=2 butterflies!)

So for N=8 there are really only 5 multiplications instead of $8\log_2(8)$ = 24 !

This is mostly important for small N!



Real complexity

So far we have counted complex multiplications and additions

Each complex add entails 2 real adds

Each complex multiply is either:

- 4 real multiplies and 2 real adds
  $(a + i\, b)\,(c + i\, d) = (a*c - b*d) + i\,(a*d + b*c)$
- or 3 real multiplies and 5 real adds
  $M1 = a*c \quad M2 = b*d \quad M3 = (a+b)*(c+d)$
  $(a + i\, b)\,(c + i\, d) = (M1 - M2) + i\,(M3 - M2 - M1)$

So

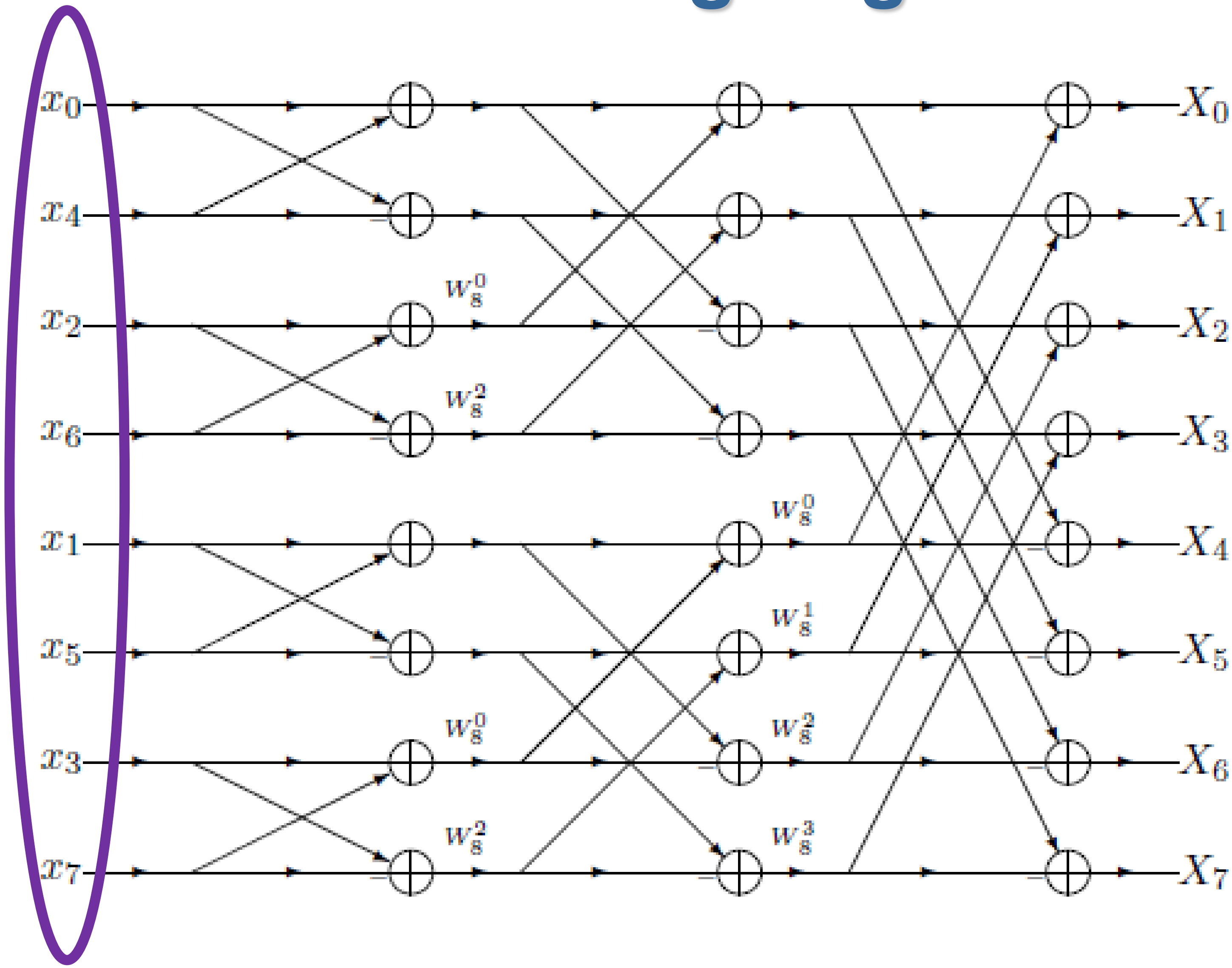
- $N \log_2(N)$ complex additions = $2N \log_2(N)$ real additions
- ½ $N \log_2(N)$ complex multiplications =
  - $2N \log_2(N)$ real multiplications
    and another $N \log_2(N)$ real additions

    or

  - $3/2\, N \log_2(N)$ real multiplications
    and another $5/2\, N \log_2(N)$ real additions

What's going on?

The order of the input values is very strange!



Our time domain signal is not arranged x_0, x_1, x_2, … !

Bit reversal

Let’s see if we can figure it out! Here for **N=16** IN-PLACE!

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	2	4	6	8	10	12	14	1	3	5	7	9	11	13	15
0000	0010	0100	0110	1000	1010	1100	1110	0001	0011	0101	0111	1001	1011	1101	1111
0	4	8	12	2	6	10	14	1	5	9	13	3	7	11	15
0000	0100	1000	1100	0010	0110	1010	1110	0001	0101	1001	1101	0011	0111	1011	1111
0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
0000	1000	0100	1100	0010	1010	0110	1110	0001	1001	0101	1101	0011	1011	0111	1111

1st transition is cyclic left shift

2nd transition freezes the LSB and cyclic left shifts the rest

3rd transition freezes the 2 LSBs and cyclic left shifts (swaps) the rest

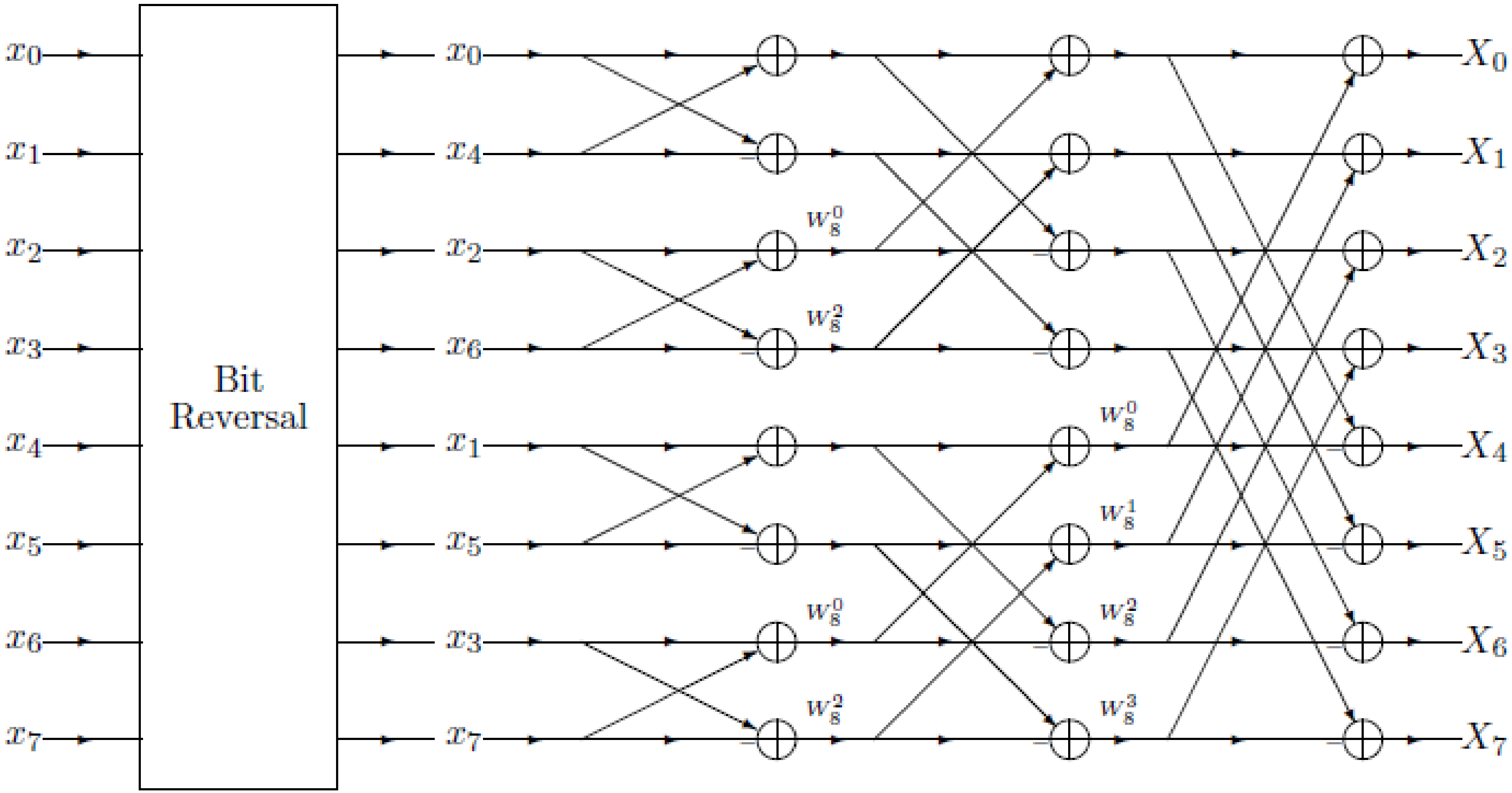
Altogether we find **abcd** $\rightarrow$ **bcda** $\rightarrow$ **cdba** $\rightarrow$ **dcba**

The bits of the index have been reversed !

This is called *bit-reversal*

and DSP processors have a special addressing mode for it

DIT N=8 with bit reversal



The matrix interpretation

The FFT can be understood as a *matrix decomposition*

that reduces the number of operations to multiply by it

For example, when N=4

$$\underline{\underline{W_4}} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -i \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & i \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The right matrix is a permutation matrix

which carries out the bit reversal

The middle matrix comprises the butterflies

(note the block matrix form)

The left matrix is the twiddle factors

What about the DIF algorithm?

The other radix-2 FFT algorithm could be called Partition in Time but is always called **D**ecimation **I**n **F**requency

To derive it algebraically we need to return to the DFT formula and partition the sum into high and low halves

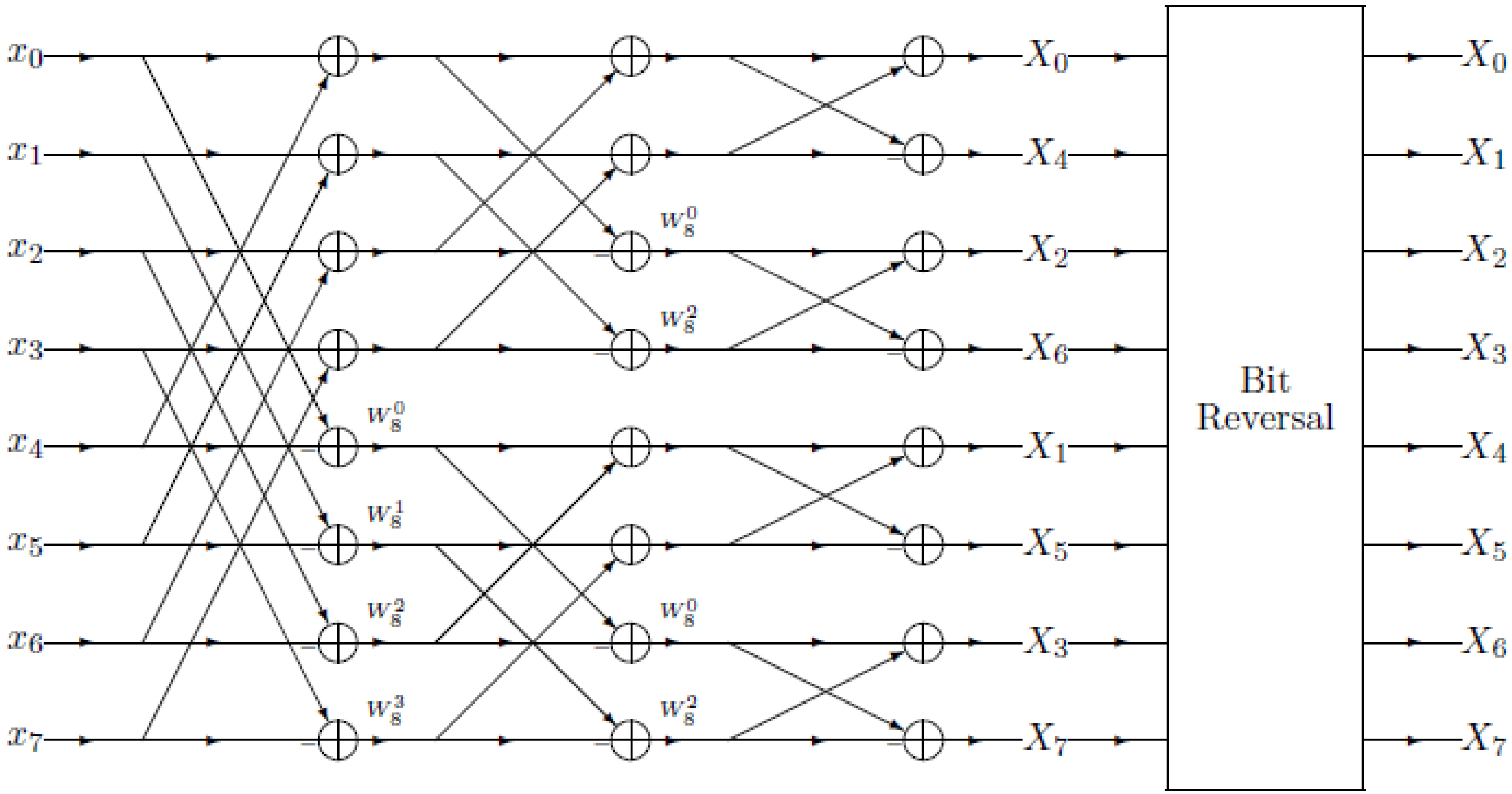
$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n W_N^{nk} = \sum_{n=0}^{\frac{N}{2}-1} x_n W_N^{nk} + \sum_{n=\frac{N}{2}}^{N-1} x_n W_N^{nk} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x_n^L W_N^{nk} + \sum_{n=0}^{\frac{N}{2}-1} x_n^R W_N^{nk} W_N^{\frac{Nk}{2}} \end{aligned}$$

We then exploit that DIF to relate X_k (even k) with X_{k+1} resulting in butterflies

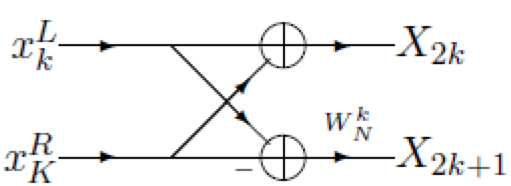
But instead of working hard we'll use a trick!

Performing the transposition theorem on the N=8 DIT (and a mirror reflection) gives us the n=8 DIF!

DIF N=8



DIF butterfly



FIFO FFT

There are many other Fast Fourier Transform algorithms!

What if we need to update the DFT every sample?

In other words, $[x_{0,} x_{1,} x_{2,} \ldots x_{N-1}], [x_{1,} x_{2,} x_{3,} \ldots x_N], [x_{2,} x_{3,} x_{4,} \ldots x_{N+1}], \ldots$

You might already know the trick
on how to update a simple moving average

$$A_{\mathbf{1}} = x_0 + x_1 + x_2 + \ldots + x_{N-1}$$
$$A_{\mathbf{2}} = x_1 + x_2 + x_3 + \ldots + x_N \quad = A_{\mathbf{1}} - x_0 + x_N$$
$$A_{\mathbf{3}} = x_2 + x_3 + x_4 + \ldots + x_{N+1} \quad = A_{\mathbf{2}} - x_1 + x_{N+1}$$

This is implemented by maintaining a FIFO of length N
adding the new input and subtracting the one to be discarded

A similar trick works for weighted MA
if the weights form a geometric progression

$$A_{\mathbf{1}} = x_0 + q\, x_1 + q^2\, x_2 + \ldots + q^{N-1}\, x_{N-1}$$
$$A_{\mathbf{2}} = x_1 + q\, x_2 + q^2\, x_3 + \ldots + q^{N-1}\, x_N \quad = (A_{\mathbf{1}} - x_0)/q + q^{N-1}\, x_N$$
$$A_{\mathbf{3}} = x_2 + q\, x_3 + q^2\, x_4 + \ldots + q^{N-1}\, x_{N+1} \quad = (A_{\mathbf{2}} - x_1)/q + q^{N-1}\, x_{N+1}$$

FIFO FFT (cont)

The DFT is just such a weighted moving average, with q = W_N^k

but when moving from time to time we shouldn't *reset the clock*!

So

$$
\begin{aligned}
X_{k_m} &= \sum_{n=0}^{N-1} x_{m+n} W_N^{(m+n)k} \\
X_{k_{m+1}} &= \sum_{n=0}^{N-1} x_{m+1+n} W_N^{(m+1+n)k} \\
&= \sum_{n=1}^{N} x_{m+n} W_N^{(m+n)k} \\
&= \sum_{n=0}^{N-1} x_{m+n} W_N^{(m+n)k} - x_m W_N^{mk} + x_{m+N} W_N^{(m+N)k}
\end{aligned}
$$

and hence $X_{k_{m+1}} = X_{k_m} + (x_{m+N} - x_m) W_N^{mk}$

requiring only 2 complex additions and one multiplication per k

or altogether N multiplications and 2N additions!

Goertzel's algorithm

Sometimes we are only interested in
the *energy* $|X_k|^2$ of a few of the frequencies k
and computing all N spectral values would be wasteful

For example, when looking for energy at a few discrete frequencies
as in a DTMF detector

For such cases there is an algorithm due to Goertzel (Herzl in Russian)
which is less expensive that running many bandpass filters

The idea is to compute only the X_k needed
by using Horner's rule for evaluating polynomials (simplify W_N^k to W)

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} = x_0 + x_1 W + x_2 W^2 + \ldots x_{N-1} W^{N-1}$$

$$= \Big(\Big(\cdots (x_{N-1} W + x_{N-2}) W + \ldots + x_2 \Big) W + x_1 \Big) W + x_0$$

This can be further simplified to get a noncomplex recursion

Goertzel 1

To make the recursion look like a convolution we use V = W^{-1}

$$X_k = \sum_{n=0}^{N-1} x_n V^{N-n} = x_0 V^N + x_1 V^{N-1} + \ldots + x_{N-2} V^2 + x_{N-1} V$$

Changing the overall phase doesn't change the power spectrum

$$X'_k = x_0 V^{N-1} + x_1 V^{N-2} + \ldots + x_{N-2} V + x_{N-1}$$

which using Horner's rule is coded like this :

$$\begin{array}{l} P_0 \leftarrow x_0 \\ \texttt{for } n \leftarrow 1 \texttt{ to } N-1 \\ \qquad P_n \leftarrow P_{n-1} V + x_n \\ X'_k \leftarrow P_{N-1} \end{array}$$

Since all the x_n are real, at each step $P_n - P_{n-1}V$ is real

So we implicitly define a new real sequence Q_n by $P_n = Q_n - Q_{n-1}W$

Goertzel 2

After a little algebra we find the following recursion:

$$
\begin{aligned}
&\texttt{Given: } x_n \texttt{ for } n = 0 \ldots N-1 \\
&Q_{-2} \leftarrow 0, \qquad Q_{-1} \leftarrow 0 \\
&Q_0 \leftarrow x_0 \\
&\texttt{for } n \leftarrow 1 \texttt{ to } N-1 \\
&\qquad Q_n \leftarrow x_n \quad + \quad A Q_{n-1} - Q_{n-2} \\
&X'_k \leftarrow Q_{N-1} - W Q_{N-2}
\end{aligned}
$$

And the desired energy is given by

$$|X_k|^2 = Q_{N-1}^2 + Q_{N-2}^2 - A Q_{N-1} Q_{N-2}$$

where $A \equiv V + W = 2\cos(\frac{2\pi k}{N})$.

Using Goertzel

To use Goertzel first decide on how many points N you want to use

Since Goertzel's algorithm only works for integer digital frequencies
(that is, for analog frequencies $f = k/N \, fs$)
larger N allows finer resolution and narrower bandwidth
but also longer computation time and delay

For each frequency that is needed

- compute W and A
- initialize
- iterate N-1 times using A
- compute X using W
- compute the desired squared power using A

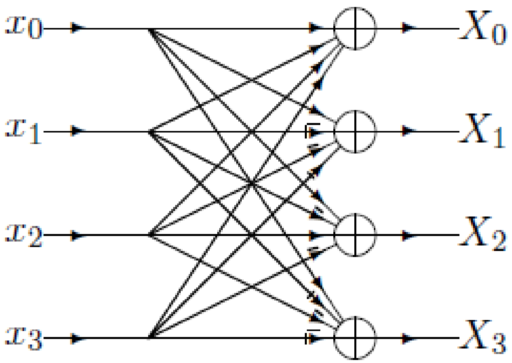
Other radixes

While radix-2 is popular, sometimes other radixes are better

The radix 4 DFT is

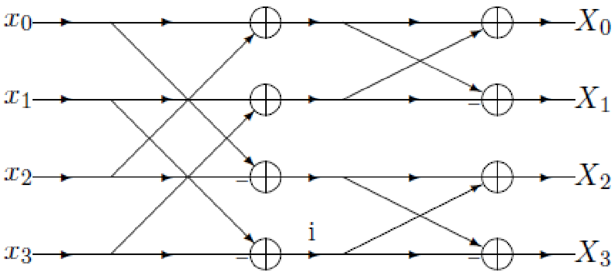
$$\begin{array}{ccccccccc} X_0 & = & x_0 & + & x_1 & + & x_2 & + & x_3 \\ X_1 & = & x_0 & - & \mathrm{i}x_1 & - & x_2 & + & \mathrm{i}x_3 \\ X_2 & = & x_0 & - & x_1 & + & x_2 & - & x_3 \\ X_3 & = & x_0 & + & \mathrm{i}x_1 & - & x_2 & - & \mathrm{i}x_3 \end{array}$$

which corresponds to radix-4 butterflies



12 complex additions
0 true multiplications

which is more expensive than radix-2



8 complex additions
0 true multiplications

FFT842

But this is only the case for N=4 itself

For powers of 4 there are only $\log_4 N = 1/2 \log_2 N$ stages of butterflies
and each has ¾ N complex multiplications
and so only $3/8 \log_2 N$ multiplications altogether
which is *slightly less* than ½ $\log_2 N$!

But only half of the powers of 2 are also powers of 4
so the algorithm is less applicable …

Similarly, for $N=8^m$ there are even fewer stages
but only a quarter of the powers of 2 are powers of 8

So, the FFT842 algorithm performs as many radix-8 stages that it can
it then performs either a radix-4 or a radix-2 stage as needed

It beats out pure radix-2 algorithms on general purpose CPUs
but highly optimized radix-2 are preferable on DSPs

Multiplication by FFT

When learning the Toom-Cook algorithm
we said that for large N the FFT will multiply even faster

That is because $O(N \log N) < O(N^{\log_2 3})$

We saw that long multiplication c = a*b is actually 2N convolutions

Hence convolution in the time domain takes $O(N^2)$ multiplications
but in the frequency domain it only takes O(N)

So the strategy is instead of convolution c = a * b

- use the FFT to convert from the time to the frequency domain
  a→A and b→B [O(N log N)]
- multiply point by point in the frequency domain C=AB [O(N)]
- convert back from the frequency to the frequency domain
  C → c [O(N log N)]

Altogether O(N log N) !

Example multiplication (1)

Let's see how this works for N=4 !

We want to multiply $a = a_3\, a_2\, a_1\, a_0$ by $b = b_3\, b_2\, b_1\, b_0$

We convert the numbers into time domain signals

$a_0\, a_1\, a_2\, a_3$ and $b_0\, b_1\, b_2\, b_3$

a	bit representation	time representation
0	0000	(0, 0, 0, 0)
1	0001	(1, 0, 0, 0)
2	0010	(0, 1, 0, 0)
3	0011	(1, 1, 0, 0)
4	0100	(0, 0, 1, 0)
5	0101	(1, 0, 1, 0)
6	0110	(0, 1, 1, 0)
7	0111	(1, 1, 1, 0)
8	1000	(0, 0, 0, 1)
9	1001	(1, 0, 0, 1)
10	1010	(0, 1, 0, 1)
11	1011	(1, 1, 0, 1)
12	1100	(0, 0, 1, 1)
13	1101	(1, 0, 1, 1)
14	1110	(0, 1, 1, 1)
15	1111	(1, 1, 1, 1)

Example multiplication (2)

For this simple case we can simply convert all 16 signals into the frequency domain

To do this we multiply by the DFT matrix and we find:

$$\begin{pmatrix} W_4^0 & W_4^0 & W_4^0 & W_4^0 \\ W_4^0 & W_4^1 & W_4^2 & W_4^3 \\ W_4^0 & W_4^2 & W_4^4 & W_4^6 \\ W_4^0 & W_4^3 & W_4^6 & W_4^9 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} \\ 1 & -1 & 1 & -1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} \end{pmatrix}$$

a	time representation	frequency representation
0	(0, 0, 0, 0)	(0, 0, 0, 0)
1	(1, 0, 0, 0)	(1, 1, 1, 1)
2	(0, 1, 0, 0)	(1, -i, -1, +i)
3	(1, 1, 0, 0)	(2, 1-i, 0, 1+i)
4	(0, 0, 1, 0)	(1, -1, 1, -1)
5	(1, 0, 1, 0)	(2, 0, 2, 0)
6	(0, 1, 1, 0)	(2, -1-i, 0, -1+i)
7	(1, 1, 1, 0)	(3, -i, 1, +i)
8	(0, 0, 0, 1)	(1, +i, -1, -i)
9	(1, 0, 0, 1)	(2, 1+i, 0, 1-i)
10	(0, 1, 0, 1)	(2, 0, -2, 0)
11	(1, 1, 0, 1)	(3, 1, -1, 1)
12	(0, 0, 1, 1)	(2, -1+i, 0, -1-i)
13	(1, 0, 1, 1)	(3, +i, 1, -i)
14	(0, 1, 1, 1)	(3, -1, -1, -1)
15	(1, 1, 1, 1)	(4, 0, 0, 0)

Of course, using the W matrix for conversion is $O(N^2)$

But we would get the same answers with the FFT

Example multiplication (3)

For example, let's multiply 2*3

$$
\begin{aligned}
S_0^{[2*3]} &= S_0^{[2]} S_0^{[3]} &&= 1 * 2 = 2 \\
S_1^{[2*3]} &= S_1^{[2]} S_1^{[3]} &&= -\mathrm{i} * (1 - \mathrm{i}) = -1 - \mathrm{i} \\
S_2^{[2*3]} &= S_2^{[2]} S_2^{[3]} &&= -1 * 0 = 0 \\
S_3^{[2*3]} &= S_3^{[2]} S_3^{[3]} &&= \mathrm{i} * (1 + \mathrm{i}) = -1 + \mathrm{i}
\end{aligned}
$$

Looking this up we find $S^{[2*3]} = (2, -1 - \mathrm{i}, 0, -1 + \mathrm{i})$ is indeed $S^{[6]}$

And similarly for almost all other multiplications that fit into 4 bits

- 0 * s = 0
- 1 * s = s
- 2 * 4 = 8
- 2 * 5 = 10
- 2 * 6 = 12
- 2 * 7 = 14
- 3 * 4 = 12
- 3 * 5 = 15

Example multiplication (4)

All products that fit into 4 bits work correctly - except 3*3

$$
\begin{array}{llll}
S_0^{[9]} & = 2 & \text{but} & S_0^{[3]} S_0^{[3]} = 4 \\
S_1^{[9]} & = 1 - \mathrm{i} & \text{but} & S_1^{[3]} S_1^{[3]} = 2\mathrm{i} \\
S_2^{[9]} & = 0 & \text{and} & S_2^{[3]} S_2^{[3]} = 0 \\
S_3^{[9]} & = 1 + \mathrm{i} & \text{but} & S_3^{[3]} S_3^{[3]} = -2\mathrm{i}
\end{array}
$$

What's going on?

Converting back using the iDFT we find (1, 2, 1, 0)

which has a meaningless 2 bit !!!

So, we convert back into binary digits 0121

and perform the *carries* to get 1001 which is indeed 9

For all products that exceed 4 bits

we can use 8 bits

i.e., signals with 8 time values

Spectral Estimation

Sometimes we only need to know which frequencies are in a signal

For this task the FFT is almost always ***not*** the best solution

- for unknown frequencies you need to compute the entire spectrum
- it does not give accurate frequencies - only bins (depending on N)

There are better ways, for example:

- If you know that the signal is a single sinusoid in white noise
  or N sinusoids in white noise
  then use the **P**isarenko **H**armonic **D**istribution
- If the signal can be assumed to be generated by an AR filter
  solve the Yule-Walker equations
  and the pole angles give the frequencies!

Why do we need DSPs?

In this part of the course DSP = Digital Signal Process**or**

A DSP is a CPU that is used in signal processing applications

Why do we need a DSP? Why not use a *regular* CPU?

DSPs are optimized for DSP, and thus :

- DSPs are physically small
  several millimeters as compared to several centimeters
- DSPs are much more energy efficient
  a DSP may consumes milliwatts
  as compared to standard CPUs tens of watts or more
- DSPs are less expensive
  a DSP may cost several dollars or less
  as compared to a CPUs 10s – 100s of dollars or more

Other special processors

DSPs are not the only species of special CPUs

- Array Processors specialize in matrix multiplication
- FFT chips compute FFT even faster than DSPs
  by parallelizing the butterflies (up to a given size)
- Systolic Arrays have arrays of simple processors to perform
  - matrix operations
  - convolutions
  - image processing
- Graphics Processing Units were designed for graphics displays
  but are now used for many parallelizable tasks
  such as deep learning
- AI processors, to accelerate neutral network training
- Network processors are optimal for packet forwarding

DSP Processors

We have seen that the Multiply and Accumulate (MAC) operation
is very prevalent in DSP computation

- computation of energy
- MA filters
- AR filters
- correlation of two signals
- DFT

A Digital Signal Processor (DSP) is a CPU
that can compute MACs very efficiently

In fact, a DSP computes each individual MAC in **1 *CPU clock cycle***

Thus an L coefficient MA takes (about) L clock cycles in a DSP
and to perform it in real-time
L must be less than the sample interval (time between 2 inputs)

CPU architecture

The term architecture in CS originated
when IBM designed a series of computers
and desired to use the same (assembly) code on all of them

Like in buildings, architecture means the overall design
without quantitative details

A DSP is a CPU with a specific architecture
designed to be efficient in computation of MACs

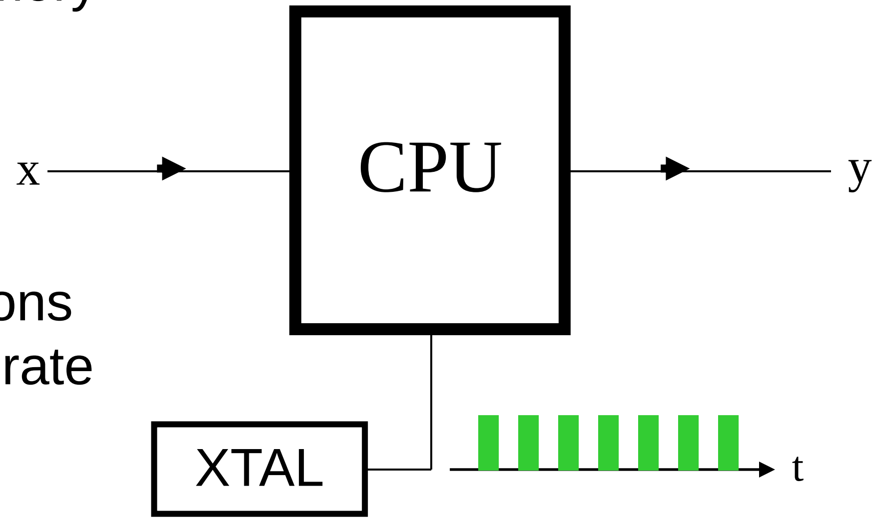
The idea is to remove all architectural elements not needed for MACs
(e.g., cache memory) in order to keep size and power minimal
and add new architectural elements that support MACs

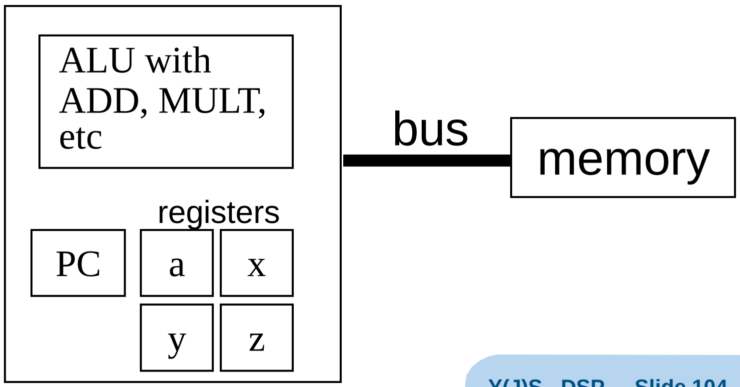
We will start with a simple generic CPU architecture
and see what elements we need to add

A simple CPU

We will assume a simplistic model of CPU architecture

- the CPU is driven by a crystal (clock)
  - faster CPUs can use higher frequency clocks
- the CPU connects to external memory
  over a bus
- the CPU has an ALU with
  the usual arithmetic operations
- the CPU has registers
  which are internal memory locations
  upon which the ALU can operate





What is the XTAL for?

All CPUs are driven by an oscillator (usually a piezoelectric crystal)
that supplies periodic pulses (we often say *clocks* or *cycles* or *ticks*)
We quantify efficiency of an operation by the number of ticks it requires

CPUs are rated according to the maximum frequency of the crystal
So, a 3 GHz CPU can compute 3 times as fast as a 1 GHZ CPU
if it is fed by a 3 GHZ crystal (but will be the same if fed by 1 GHz xtal!)
To increase yield, fabricated CPUs dies are tested for speed
and the CPUs rated according to the speed attained

Modern CPUs use microcode
their op-codes do not directly translate into hardware operations
but are actually subroutines in a lower level language

Each individual microcode instruction takes place in on pulse time

Most op-codes require multiple microcode instructions
(e.g., the multiplication op-code might be microcoded Toom-Cook)

Why registers?

CPUs are classified based on the number of addresses in an op-code

- 3 address CPUs: A1 = A2 **op** A3
- 2 address CPUs: A1 = A1 **op** A2
- 0 address CPUs (stack machines): **op**

Early computers allowed arithmetic operations on memory locations
but this severely limits memory space

So a full 3-address architecture
needs an opcode that contains 3 addresses in memory
For example, a computer with 1 MB of memory
requires 3*20bits = 60 bits just to specify memory
and more bits to describe the operation

The alternative is to enable arithmetic only on registers
which are special memory locations internal to the CPU
So, if we have 16 registers
a full 3-address architecture only requires 3*4=12 bits + operation

The cost is the need to **load** from and **store** to external memory

Special registers

Not all registers are created equal!

In addition to general purpose registers all CPUs have special ones

There is one special register called the **P**rogram **C**ounter
that always holds the address of the next op-code to be performed

It is auto-incremented each operation
but can be overwritten by *goto* and *conditional branch* op-codes

In DSPs some registers are *accumulators*
Accumulators hold larger numbers than regular registers
(e.g., a regular register may be 16 bits in length
and an accumulator 24 bits – 8 *guard bits*)
Accumulators are used for accumulating
and need the longer length in order not to overflow!

Many CPUs have other special registers
such as stack pointers, loop counters, pointer registers, etc.

High-level MAC loop

The basic *MAC loop* in high level languages is
(assuming that a and x are in static buffers)

loop over all times n
  initialize $y_n \leftarrow 0$
  loop over i from 1 to number of coefficients (L)
    $y_n \leftarrow y_n + a_i * x_j$ (j *somehow* related to i)
  output y_n

For *energy* and *correlation* i and j increase together
For *convolution* i increases and j decreases

Efficient low level programming always uses (read) pointers
since array indexing requires wasteful offset calculations

```
ADDR(a[i]) = ADDR(a[0]) + i * word-length
```

To explicitly increment the pointers

```
ADDR(a[i+1]) = ADDR(a[i]) + word-length
```

Intermediate level MAC loop

So, in some imaginary assembly level language
our MAC loop looks like this:

```
loop over all times n
   clear y
   set number-of-iterations to L
   loop
      decrement number-of-iterations
      if number-of-iterations = 0 then terminate loop
      update a pointer
      update x pointer
      multiply z ← a * x        (3-address addressing)
      increment y ← y + z       (2-address addressing)
   output y
```

Low level MAC loop

Now let's use registers! (remember we have a, x, and y registers)

```
loop over all times n
    clear y register
    set number-of-iterations to L
    loop
        decrement number-of-iterations
        if number-of-iterations = 0 then terminate loop
        update a pointer
        load contents of memory addressed by a into register a
        update x pointer
        load contents of memory addressed by x into register x
        multiply z ← a * x   (register operation!)
        increment y ← y + z  (register operation!)
    store y
```

Zero-overhead loops

DSPs, like many CPUs, have a **zero-overhead loop**

This means that we can configure a special *loop counter* register that auto-decrements and is tested implicitly

```
loop over all times n
    clear y register
    loop number-of-iterations times (zero overhead loop)
        update a pointer
        load contents of memory addressed by a into register a
        update x pointer
        load contents of memory addressed by x into register x
        multiply z ← a * x    (register operation!)
        increment y ← y + z   (register operation!)
    store y
```

Why do we no longer care about the decrement and testing?

Since additional hardware (*silicon*) takes care of this task *in parallel to* other operations!

Cycle counting

We still can't count clock ticks
since really low level (hardware) operations
need to take the op-code fetch and decode into account

So the clocks operations *inside* the outer loop look something like this:

1. Update pointer to a_i
2. Update pointer to x_j
3. LOAD contents of a_i into register a
4. LOAD contents of x_j into register x
5. Fetch operation (MULT)
6. Decode operation (MULT)
7. MULT a*x with result in register z (MULT really takes >1 clock!)
8. Fetch operation (INC)
9. Decode operation (INC)
10. INC register y by contents of register z

So, it takes at least 10 cycles to perform each MAC using a ***regular*** CPU

Our mission (and we have decided to accept it!)
is to reduce this to 1 clock cycle by adding new silicon

This really isn't right!

We ridiculously assumed each operation takes only 1 cycle

- we know multiplication takes many more
- addition frequently takes a few cycles
- even fetch really requires at least 2 cycles
  - 1 to send an address to external memory
  - 1 to retrieve the value from the memory

So we are radically underestimating
the number of cycles a regular CPU needs

But we don't care since this will happen in any CPU
even a DSP!

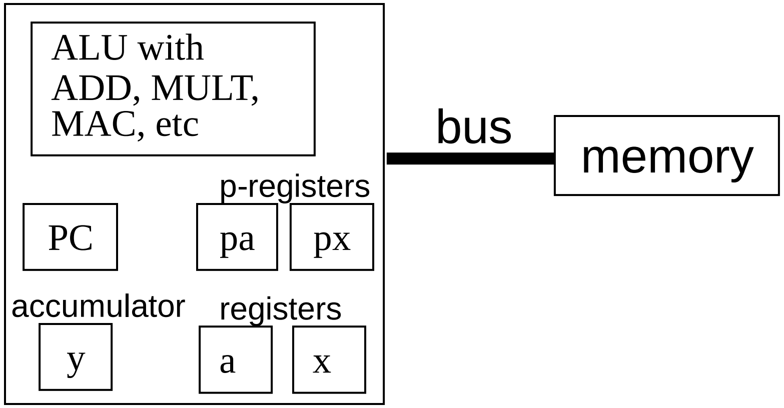
Step 1 - new opcode

To build a DSP (a 1-cycle MAC CPU)

we need to enhance the basic CPU with new hardware (silicon)

The easiest step is to define a new opcode called MAC

which is what Intel did in the *MMX extensions*



The upgraded code now looks like this:

1. Update pointer to a_i
2. Update pointer to x_j
3. LOAD contents of a_i into register a
4. LOAD contents of x_j into register x
5. Fetch operation (MAC)
6. Decode operation (MAC)
7. MAC a*x with incremented to accumulator y

However 7 > 1, so this is still NOT a DSP !

Step 2 - register arithmetic

The two operations

- `Update pointer to` a_i
- `Update pointer to` x_j

could be performed in parallel

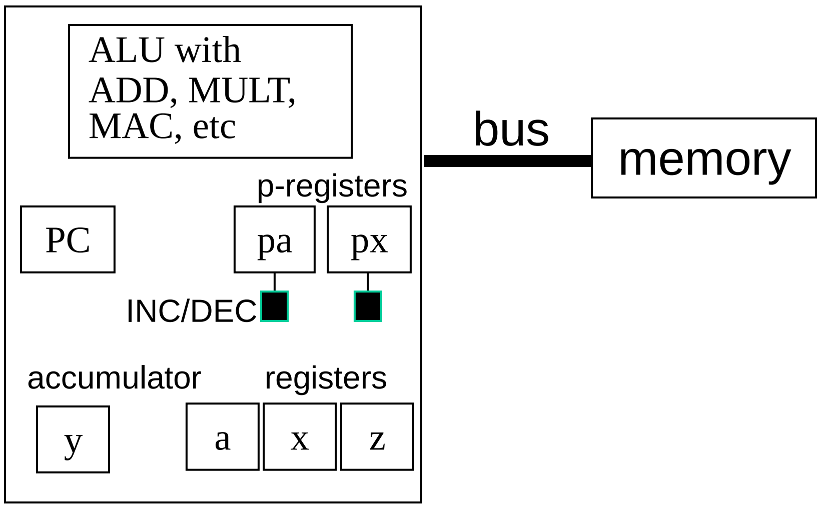
but both are performed by the ALU

So we add pointer arithmetic units

one for each *pointer register*

Special sign || used in DSP assembler

to mean operations in parallel



1. Update pointer to a_i || Update pointer to x_j
2. LOAD contents of a_i into register a
3. LOAD contents of x_j into register x
4. Fetch operation (MAC)
5. Decode operation (MAC)
6. MAC a*x with incremented to accumulator y

However 6 > 1, so this is still NOT a DSP !

Step 3 - memory banks and buses

We would like to perform the **load**s in parallel

but we can't since they both have to go over the same bus

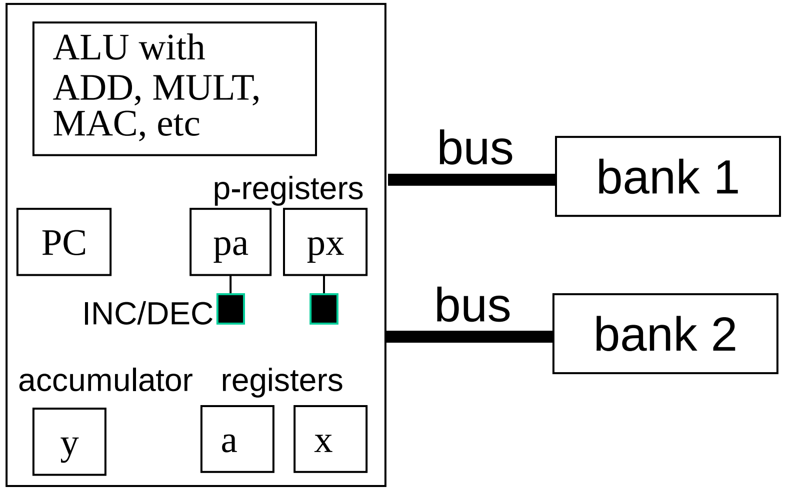
So we add another bus

and segment into *memory banks*

so that there is no contention !

There ***is*** *dual-port memory*

but it has an *arbitrator* which adds delay



1. Update pointer to a_i || Update pointer to x_j
2. LOAD a_i into a || LOAD x_j into x
3. Fetch operation (MAC)
4. Decode operation (MAC)
5. MAC a*x with incremented to accumulator y

However 5 > 1, so this is still NOT a DSP !

Harvard architecture

One of the first digital computers
was the **A**utomatic **S**equence **C**ontrolled **C**alculator (the Mark I)
that was designed in Harvard by Howard Aiken (and built by IBM)
and employed >750,000 electromechanical components

It was funded by the US Navy
and later enhanced to become the Harvard Mark II, III, and IV

The Harvard computers were used by John von Neumann
for calculations related to the Manhattan project
and was programmed by Grace Hopper (the originator of the word *bug*)

The overall architecture of the Harvard computers included

- a central processing unit
- program memory (that is immutable during run-time)
- data memory (that can be read and written during run-time)

Von Neumann architecture

The **E**lectronic **N**umerical **I**ntegrator and **C**omputer is often called the 1st
fully programmable, general-purpose, digital computer

It was designed by John Mauchly and J. Presper Eckert
at the University of Pennsylvania, funded by the US army
based on principles described in 1945 by John von Neumann

The overall architecture of the ENIAC included

- a central processing unit
- a single memory that holds both program op-codes and data

Von Neumann merged program and data memory not only to simplify
but to enable changing the program during run-time (*learning*)

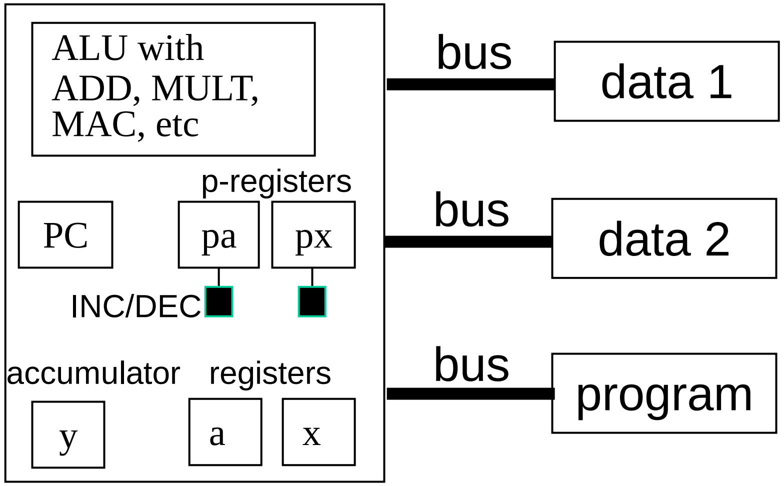
Turing, after reading von Neumann's paper,
abstracted these principles into what is called the *Turing machine*

The von Neumann architecture is used in all modern computers
except DSPs!

Step 4 - Harvard architecture

By adopting Harvard architecture with yet another bus to another memory we needn't count fetch since it is performed in parallel

We can remove the decode cycle as well (we'll see why later)



1. Update pointer to a_i || Update pointer to x_j
2. LOAD a_i into a || LOAD x_j into x
3. MAC a*x with incremented to accumulator y

However 3 > 1, so this is still NOT a DSP !

Step 5 - pipelines

We seem to be stuck

- Update MUST be before Load
- Load MUST be before MAC

But we can use a *pipelined* approach

It takes 1 tick per tap as long as the pipeline is *full*

altogether it takes n+2 clocks (which is n for large n!)

More generally, a pipeline of depth D takes n+D-1 ticks

op

U1	U2	U3	U4	U5		
	L1	L2	L3	L4	L5	
		M1	M2	M3	M4	M5
1	2	3	4	5	6	7

t

Why do we need longer pipelines?

Why would we want D>3 ?

Remember that we said
that we don't have to count ticks for fetch and decode?
These are actually performed in parallel using a pipeline

Doesn't a MAC op-code have to multiply before adding?
Yes, but the DSP chip pipelines them

Remember we said that multiplication
really takes many more than 1 cycle?
We can pipeline these cycles to reduce overall execution time

Of course, adding to the pipeline's depth

- increases the delay
- makes filling the pipeline more challenging
- is subject to diminishing returns (Amdahl's law)

Pipelines in other CPUs

Many modern CPUs employ pipelines – how are DSPs different?

- DSPs employ pipelining as a *last resort* (when logically *stuck*)
  other CPUs use pipelining as the main (only) parallelization

  Thus, non-DSP CPUs *can* pipeline *all* stages of a MAC
  resulting in lower ticks/tap
  but more delay and less determinism
  Advanced non-DSP CPUs even employ speculative lookahead
  to attempt to keep the pipeline full with conditional branches

- DSPs allow programmers to monitor and manipulate the pipeline

  for other CPUs pipelining is basically transparent

- DSPs actually get more from pipelining
  due to memory banks and Harvard architecture

DSP programming

DSP programming is harder than regular programming

(which is why it is today mostly done in India and eastern Europe)

For maximal efficiency :

- one needs to program in assembly
- one needs to know the DSP’s architecture
- one needs to program in *parallel* assembly
- one needs to place data in the correct memory banks
- one needs to keep the pipeline full

The last portion often requires painstakingly rewriting and reordering

The usual technique is to start with many NOPs

and iteratively improve the program eliminating pipeline *holes*

DSP programmers

There are three types of DSP programmers

1. algorithm designers
   - use floating point
   - care more about theory than real-time
   - usually code in MATLAB, Python, C++

2. low-level coders
   - structure code for real-time
   - convert algorithms from floating point to fixed point
   - usually code in C

3. DSP coders
   - convert real-time oriented C to parallel assembly
   - work directly on the silicon
   - program critical routines in DSP assembly language
   - program non-critical routines in C with pragmas

Zero-overhead interrupts

How do the input sample values get into the buffers?

All CPUs have (serial or parallel) I/O ports
    with memory for one value (bit or byte or whatever)

There are two methods for transferring from an input port to the buffer:

1. Polling – the CPU repeatedly checks if something is in port memory
    this is very inefficient since we need to check overly frequently

2. Interrupts – when the input port is ready it raises an *interrupt*
    causing the CPU to perform a *context switch*

Context switches are very expensive on regular CPUs
    since all registers need to be saved and later restored

Most DSPs have a limited zero-overhead interrupt mechanism
    where certain registers are copied into shadow registers in 1 cycle
        and restored when returning form the interrupt handler

Such handlers are usually limited to a small number of instructions
        (just enough to copy and increment the buffer length)
    and are themselves non-interruptable

Fixed point

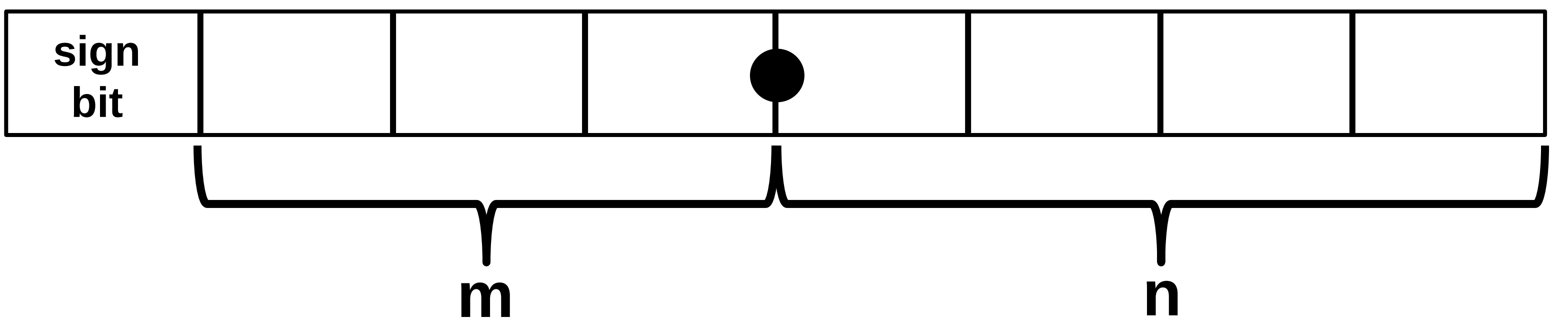
In the real world signal values are real numbers

that can be well approximated by rational numbers

but not usually by integers

Fixed point representation represents a rational number as a integer

by *fixing* the (binary) decimal *point,* described as Qm**.**n notation



We often take m=0 and use Qn (scientific) notation

in which the integer value $\mathbf{I}$ represents the rational $\mathbf{Q} = \mathbf{I} / 2^n$

In each part of the program

all values are represented in the same Qn

but in different parts different Qn are used

Q representation examples

On a machine with 16 bit registers

binary	integer	Q15 value	Q8 value	Q4 value
0100000000000000	16384	0.5	64.0	1024.0
0010000000000000	8192	0.25	32.0	512.0
0001000000000000	4096	0.125	16.0	256.0
1100000000000000	-16384	-0.5	-64.0	-1024.0
1010000000000000	-8192	-0.25	-32.0	-512.0
1001000000000000	-4096	-0.125	-16.0	-256.0

since

0.100000000000000 = 0.5

01000000.00000000 = 64.0

010000000000.0000 = 1024.0

Saturation Arithmetic

Many DSPs are fixed point, i.e. handle (2s complement) integers only

Floating point is more expensive and slower
(because of the need to renormalize after calculation)

Floating point numbers can underflow

Fixed point numbers can overflow

We saw that *accumulators* have guard bits to protect against overflow

When regular fixed point CPUs overflow

- numbers greater than MAXINT become negative
- numbers smaller than -MAXINT become positive

Fixed point DSPs have a *saturation arithmetic* mode

- numbers larger than MAXINT become MAXINT
- numbers smaller than -MAXINT become -MAXINT

this is still an error, but a smaller error

There is a tradeoff between safety from overflow and SNR

What else is special?

We have already mentioned that DSPs support bit-reversed addressing which speeds calculation of FFTs

However, it is important to consider what DSPs don't have:

- most DSPs run at modest clock rates compared to modern CPUs (50MHz, 100 MHz, 200 MHz)
- many DSPs are fixed point
- many DSPs have modest word sizes (16/24 bits, 32/40 bits)
- DSPs do not have program or data cache memory
- DSPs do not use modern accelerations, e.g., speculative execution
- most DSPs do not have a division op-code
- DSPs do not have a square-root op-code

That's why DSPs are amazing at DSP tasks (but miserable at others) but can be small and require little power

What – no division?

Most DSPs do not have an op-code for division, which is often needed

For example, **A**utomatic **G**ain **C**ontrol divides by the RMS

If time is not critical one can use a library routine
but for real-time we need something better

It is enough to know how to invert y = 1/x
for which there are many iterations
that converges to the right answer

The simplest one is

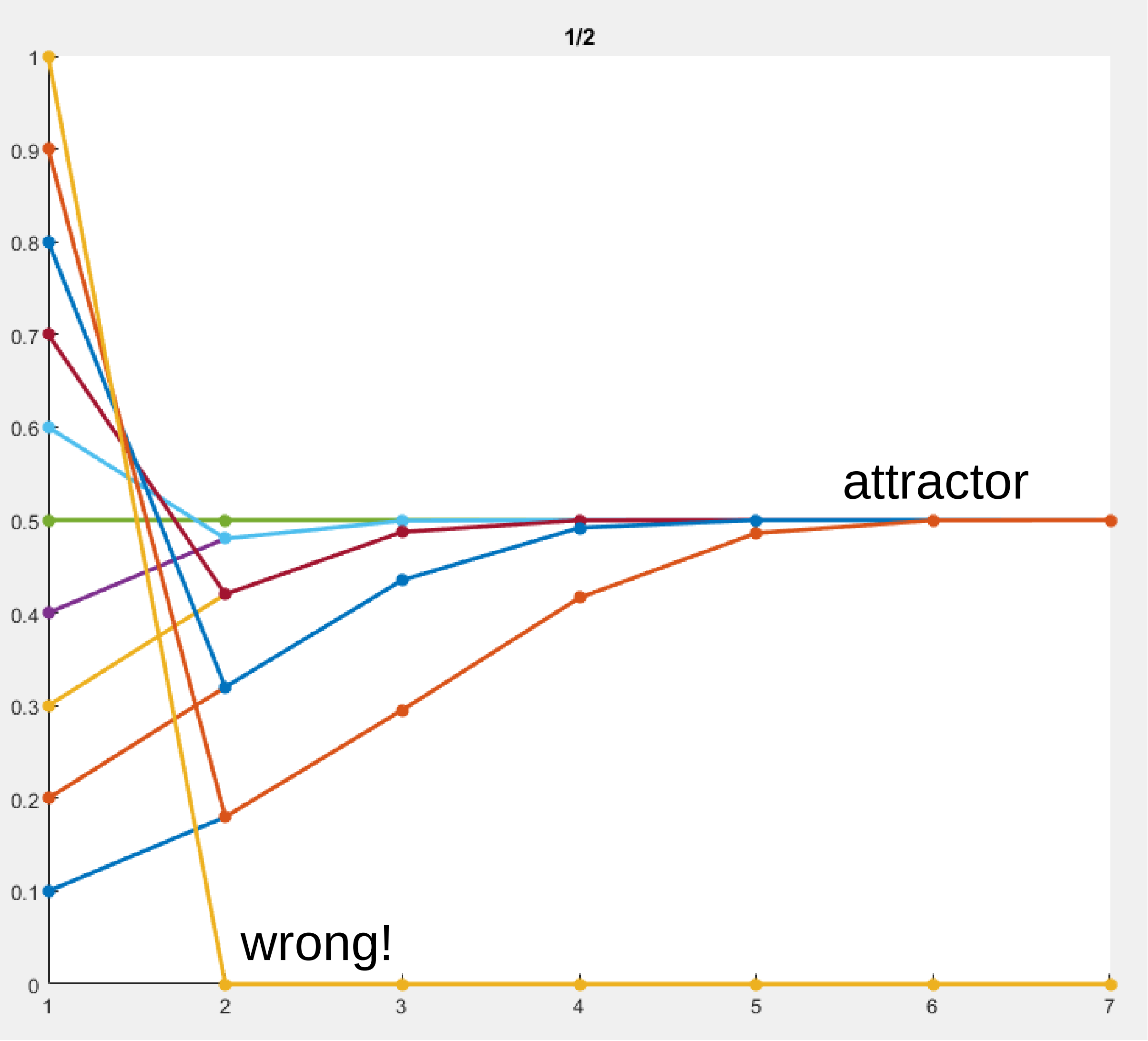
```
Start with a reasonable guess for y
Loop
      y ← y * (2 - y*x)
```

If you start with a good guess*, this will converge in a few iterations

For AGC, initializing with the previous value, 3 iterations is often enough

* many DSPs have an inverse-seed opcode

Example : how much is ½?



Full division

If you need to divide y = N/D
and don’t want to invert and multiply
then *Goldschmidt division* uses a similar trick

```
N′ ← N
D′ ← D
Loop
    y ← 2 – D′
    N′ ← N′ * y
    D′ ← D′ * y
```

More generally
many operations can be carried out by finding a recursion
for which the answer is an *attractive fixed point*

Square root

Square roots are often needed in DSP

and some DSPs have a *square-root-seed* op-code

but none have a full square-root

The most common non-DSP iteration for square root $y = \sqrt{x}$

is the Newton-Raphson iteration which converges quadratically

(and is great for finding square roots in your head!)

```
y ← square-root-seed(x)
Loop
    y ← ½ (y + x/y)
```

but this requires a division!

Sometimes one can use the fact that $\log(\sqrt{x}) = \frac{1}{2} \log(x)$

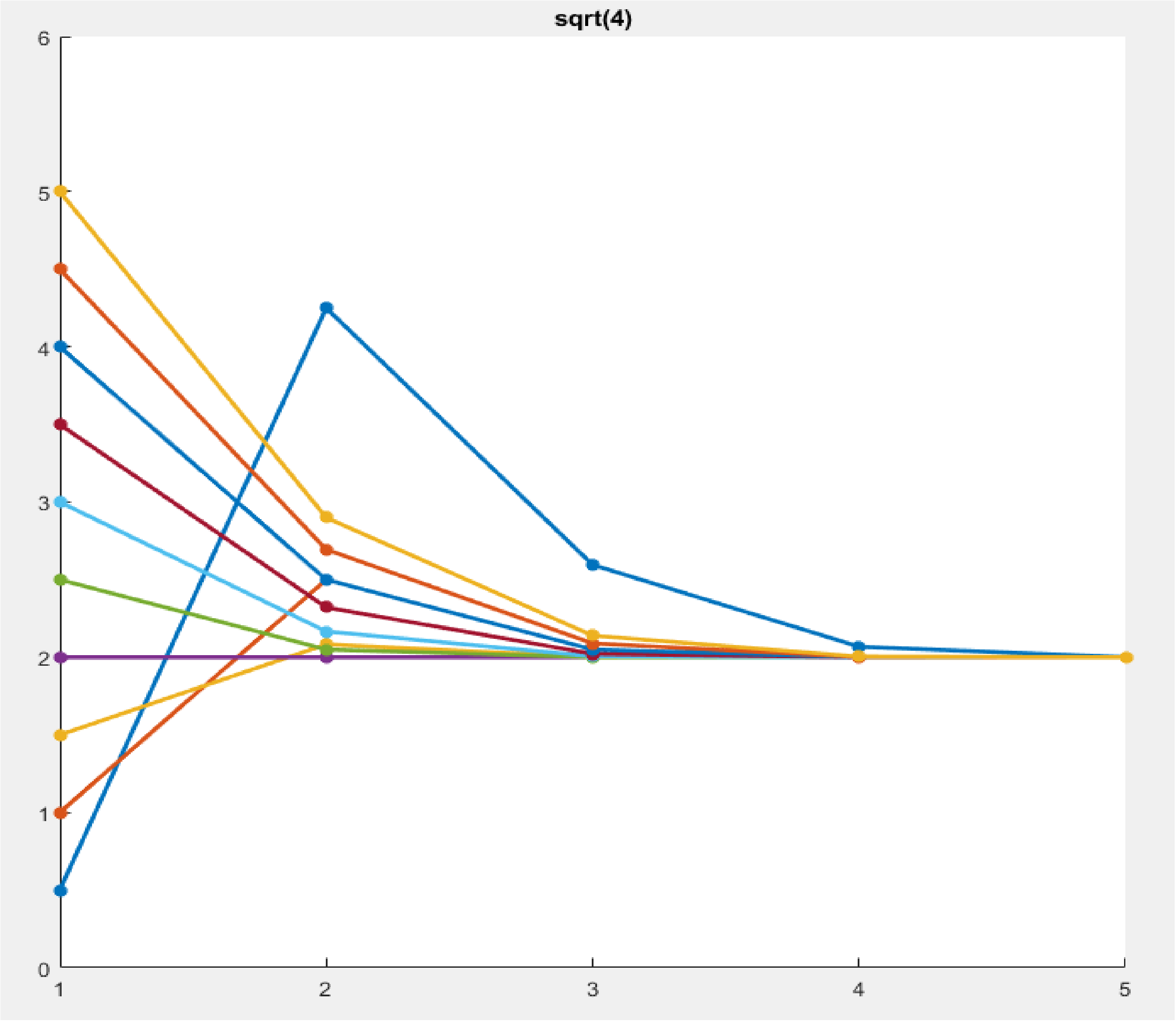
along with algorithms for log and power

For small intervals one can use polynomial approximations

such as $y \approx -0.5973x^2 + 1.4043x + 0.1628$

But there is often an alternative

Example : how much is √4 ?



Pythagorean addition

In DSP applications square root is mostly required as part of

Pythagorean addition $$x \oplus y \equiv \sqrt{x^2 + y^2}$$

for which there are approximations such as

$$x \oplus y \approx \text{abmax}(x, y) + k \ \text{abmin}(x, y)$$

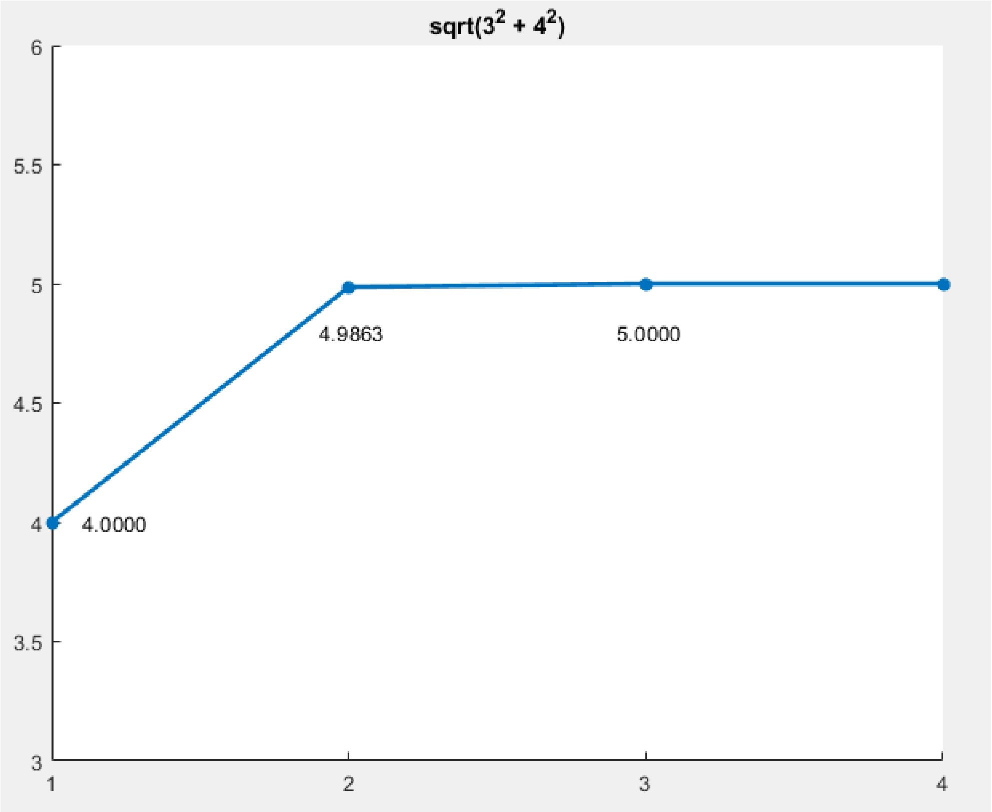
where 0.25 < k < 0.31

- k=0.267304 gives the exact mean
- k=0.300585 gives minimum variance

More importantly the **Moler-Morrison** algorithm
which requires 2 divisions
and the CORDIC algorithm requires only shift and add
and converges exponentially, gaining 1 bit per iteration

Moler Morrison

```
p ← max(|x|, |y|)
q ← min(|x|, |y|)
while q > ε
        r ← (q/p)²
        s ← r / (4 + r)
        p ← p + 2sp
        q ← sq
return p
```



Note that in each iteration

- sum of squares remains the same
- s decreases

Sine and Cosine

Of course, we need sin and cos all the time!

Non-DSP libraries use Taylor expansions, which are inefficient

However, we most often need to generate both sin(ωn) and cos(ωn) for increasing n = 0, 1, 2, 3, 4 ...

1. We know how to update sin(ωn) using a difference equation

   $$\sin(\omega(n+1)) = 2\cos(\omega)\sin(\omega n) - \sin(\omega(n-1))$$

   which requires 2 initial values

2. Both sin and cos together is easy since $e^{i\omega(n+1)} = e^{i\omega}\, e^{i\omega n}$ which is the same as the trig identities:

   $$\sin(\omega(n+1)) = \cos(\omega)\sin(\omega n) + \sin(\omega)\cos(\omega n)$$

   $$\cos(\omega(n+1)) = \cos(\omega)\cos(\omega n) - \sin(\omega)\sin(\omega n)$$

   So from a single pair we can continue

However, both methods may suffer from *error accumulation*

CORDIC

The **CO**ordinate **R**otation for **DI**gital **C**omputers (CORDIC) algorithm
is an iteration for calculating elementary functions
using only addition and binary shift

It was described in 1959 by Volder (and refined Walther)
and was used in the first scientific hand-held calculator (HP-35)

It computes 1 bit / iteration
and so is great for hardware implementations
but has a *conditional* and so breaks pipelines

CORDIC can simultaneously compute these pairs of functions

- sin(θ) and cos(θ)
- sinh(θ) and cosh(θ)
- $\sqrt{x^2 + y^2}$ and $\tan^{-1}(\frac{y}{x})$
- $\sqrt{x^2 - y^2}$ and $\tanh^{-1}(\frac{y}{x})$
- $\sqrt{x}$ and ln(x)
- e^x (alone)

The main idea behind CORDIC for sin/cos

An arbitrary angle θ in the 1st quadrant $[0,\pi/2]$ can always be written as a sum of angles $\pm\alpha_i$ where $\tan(\alpha_i) = 2^{-i}$

$$\theta = \sum_{k=0}^{\infty} (\pm \tan^{-1} 2^{-k})$$

For example,

$90^o = \alpha_0 + \alpha_1 + \alpha_2 + \alpha_3 - \alpha_4 + \alpha_5 + \ldots$

$60^o = \alpha_0 + \alpha_1 - \alpha_2 + \alpha_3 - \alpha_4 - \alpha_5 + \ldots$

$30^o = \alpha_0 - \alpha_1 + \alpha_2 - \alpha_3 + \alpha_4 + \alpha_5 + \ldots$

$15^o = \alpha_0 - \alpha_1 - \alpha_2 + \alpha_3 + \alpha_4 - \alpha_5 + \ldots$

k	$\tan^{-1} 2^{-k}$
0	45^o
1	26.566^o
2	14.036^o
3	7.125^o
4	3.576^o
5	1.790^o
...	...

Note that multiplication by $\tan(\alpha_i)$ is actually a right shift

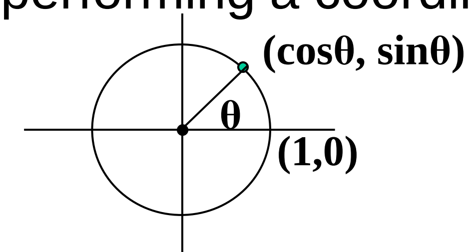
CORDIC for sin/cos

Recall that coordinate rotations in the plane are performed by

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{R}(\theta) \begin{pmatrix} x \\ y \end{pmatrix}$$

$$\mathbf{R}(\theta) \equiv \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} = \cos(\theta) \begin{pmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{pmatrix}$$

We can reach an arbitrary point on the unit circle $(\cos(\theta), \sin(\theta))$
by starting from the point (1,0) $[\theta=0]$
and performing a coordinate rotation



$$\mathbf{R}(\theta) \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix}$$

The coordinate rotation can be decomposed
into the sum of angles $\pm\alpha_i = \pm \tan^{-1} 2^{-k}$

So the R(θ) can be written as
the product of matrices of the form

$$\mathbf{M}_i = \begin{pmatrix} 1 & -\frac{1}{2^i} \\ \frac{1}{2^i} & 1 \end{pmatrix}$$

No multiplications!

Multiplying by the M matrices
only requires addition/subtractions and left shifts

All that is needed is to finish off is to multiply once by all the cos(α_i)
but since cos is an even function we can precompute the product

$$K \equiv \prod_{i=0}^{\infty} \cos(\alpha_i) \approx 0.607$$

And instead of multiplying by K at the end
we can simply start with the vector (K,0) instead of (1,0) !
Note that since the multiplicands are all inverse powers of 2
each iteration gives us another bit of accuracy
(exponentially fast convergence!)

We can now give the full CORDIC algorithm
to simultaneously calculate the cos and sin
of any angle in the 1st quadrant

What do we do for the other quadrants?

The CORDIC algorithm

The full derivation is in the course text!

$$\begin{aligned}
x &\leftarrow K \\
y &\leftarrow 0 \\
z &\leftarrow \theta \\
\texttt{for} \quad & i \leftarrow 0 \ \texttt{to} \ b-1 \\
& s \leftarrow \texttt{sgn}(z)
\end{aligned}$$

depends on sgn(z) $\left\{ \begin{aligned} x &\leftarrow x - s \cdot y \cdot 2^{-i} \\ y &\leftarrow y + s \cdot x \cdot 2^{-i} \\ z &\leftarrow z - s \cdot \tan^{-1}(2^{-i}) \end{aligned} \right.$

$$\begin{aligned}
\cos(\theta) &\leftarrow x \\
\sin(\theta) &\leftarrow y \\
\texttt{error} &\leftarrow z
\end{aligned}$$