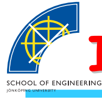


Introduction to DSP

Maurizio Palesi
Dipartimento di Ingegneria Informatica e delle Telecomunicazioni
University of Catania, Italy
mpalesi@diit.unict.it
http://www.diit.unict.it/users/mpalesi

Contents

- **Part I**: Digital Signal Processing: Concepts and Theory
- **Part II**: Signal Processing using a DSP

Part-I
Digital Signal Processing
Concepts and Theory

Maurizio Palesi

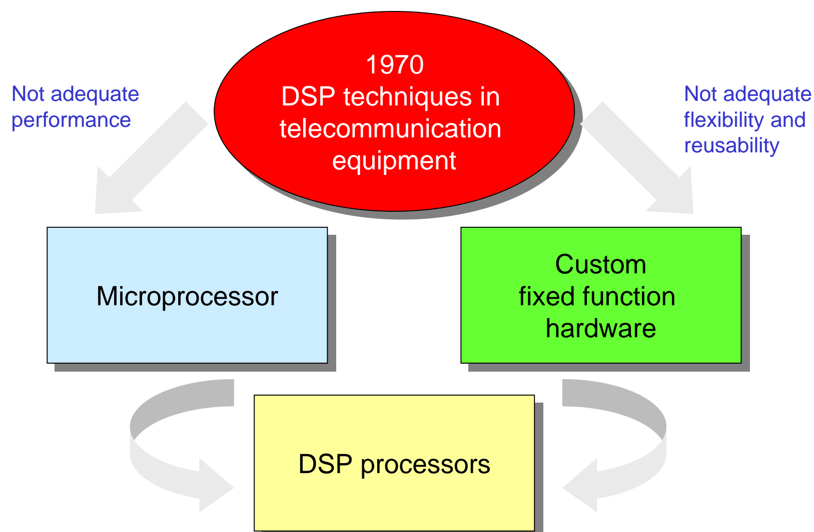
What is a DSP?

- Digital
  - Operating by the use of discrete signals to represent data in the form of numbers
- Signal
  - A variable parameter by which information is conveyed through an electronic circuit
- Processing
  - To perform operations on data according to programmed instructions
- Digital Signal Processing
  - Changing or analysing information which is measured as discrete sequences of numbers

Main Characteristics

- Compared to other embedded computing applications, DSP applications are differentiated by the following
  - Computationally demanding
    - Iterative numeric algorithms
  - Sensitivity to small numeric errors (audible noise)
  - Stringent real-time requirements
  - Streaming data
  - High data bandwidth
  - Predictable (though often eccentric) memory access pattern
  - Predictable program flow (nested loops)

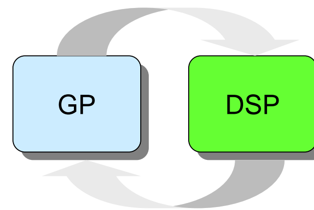
DSP Processors



DSP vs. General Purpose

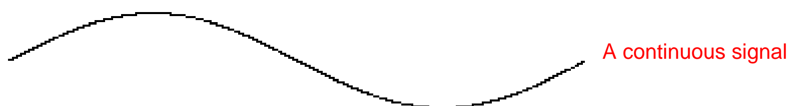
- DSPs adpot a range of specialized features
  - Single-cycle multiplier
  - Multiply-accumulate operations
  - Saturation arithmetic
  - Separate program and data memories
  - Dedicated, specilized addressing hw
  - Complex, specialized instruction sets

VLIW, Superscalar, SIMD, multiprocessing, ...

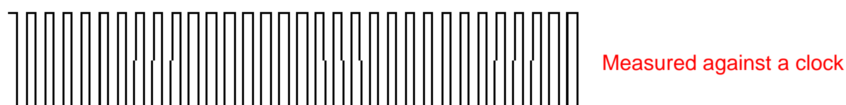


- Today, virtually very commercial 32-bit microprocessor architecture (from ARM to 80x86) has been subject to some kind of DSP-oriented enhancement

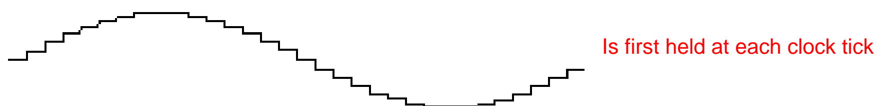
Converting Analogue Signals



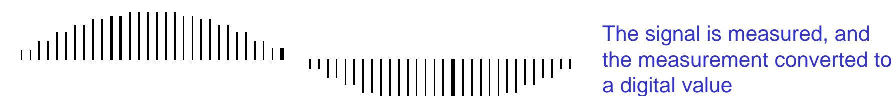
A continuous signal



Measured against a clock



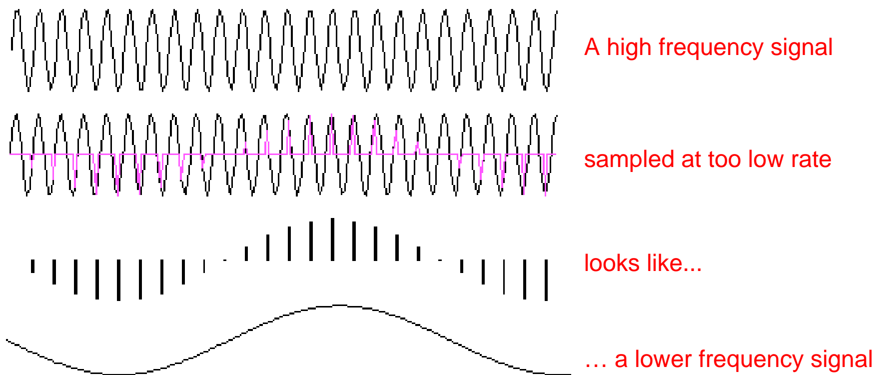
Is first held at each clock tick



The signal is measured, and the measurement converted to a digital value

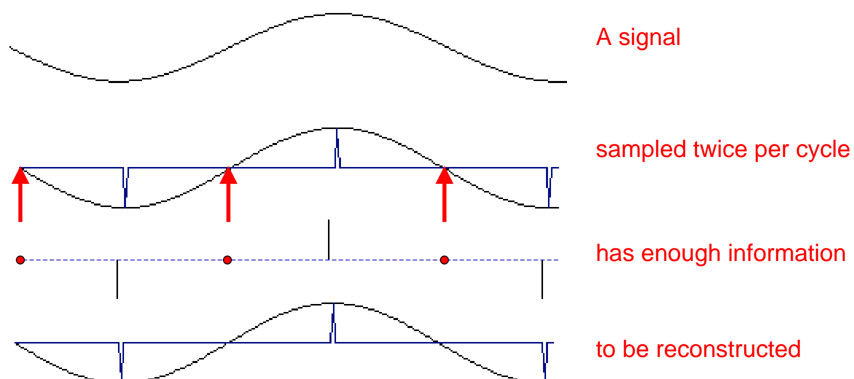
Aliasing

- Some higher frequencies can be incorrectly interpreted
  - ➔ Aliasing problem: One frequency looks like another



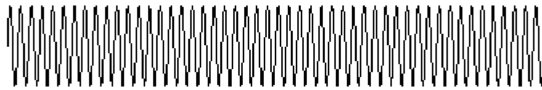
Aliasing

- We must sample faster than twice the frequency of the highest frequency component [Nyquist's theorem]
  - ➔ This avoids aliasing
    - ✓ Actually, Nyquist says that we have to sample faster than the signal bandwidth

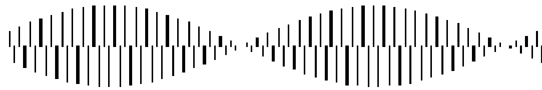


Reconstruction

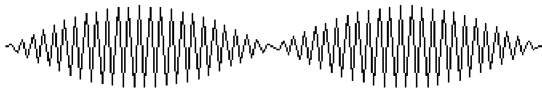
- Although Nyquist showed that we have all the information needed to reconstruct the signal, the sampling theorem does not say the samples will look like the signal



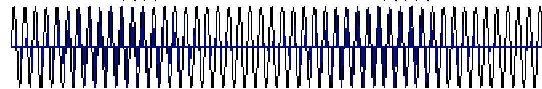
A high frequency signal



sampled fast enough



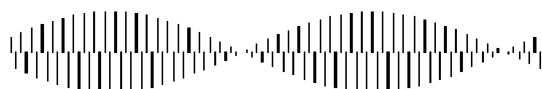
may still look wrong



but can be reconstructed

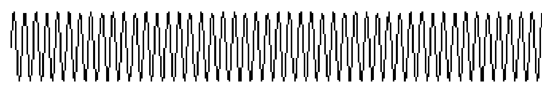
Reconstruction

- The signal is properly reconstructed from the samples by low pass filtering
  - ➔ The low pass filter should be the same as the original antialias filter



A sampled signal

must be low pass filtered



to reconstruct the original

Frequency Resolution

- We cannot see slow changes in the signal if we don't wait long enough
  - ➔ We must sample for at least one complete cycle of the lowest frequency we want to resolve
- **Compromise**
  - ➔ We must sample fast to avoid and for a long time to achieve a good frequency resolution
  - ➔ Sampling fast for a long time means we will have a lot of samples
    - ✓ Lots of samples means lots of computation
  - ➔ So we will have to compromise between resolving frequency components of the signal, and being able to see high frequencies

Quantisation

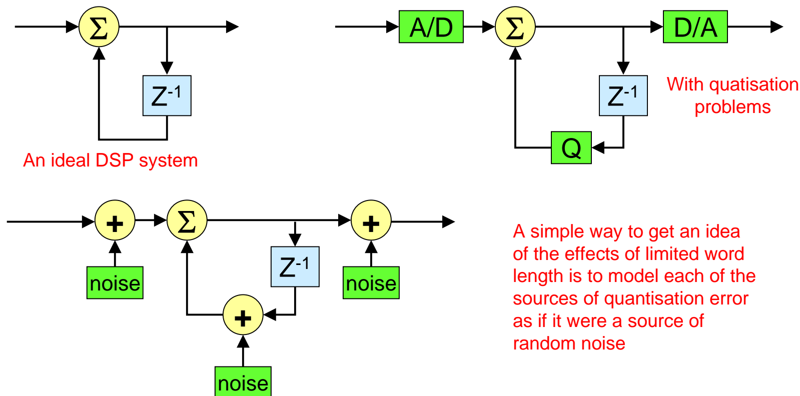
- When the signal is converted to digital form, the precision is limited by the number of bits available
- The errors introduced by digitisation are both
  - ➔ Non linear: We cannot calculate their effects using normal maths
  - ➔ Signal dependent: the errors are coherent and so cannot be reduced by simple means

Limited precision leads to errors...

which are signal dependent

Quantisation error

- A real DSP system suffers from three sources of error
  - Limited precision due to word length in A/D conversion
  - Errors in arithmetic due to limited precision within the processor itself
  - Limited precision due to word length in D/A conversion



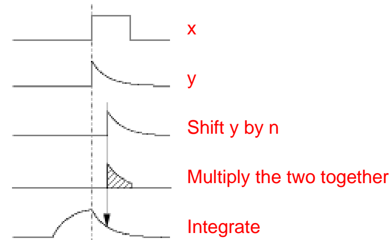
Time Domain Processing

- Correlation
- Autocorrelation to extract a signal from noise
- Cross correlation to locate a know signal
- Cross correlation to identify a signal
- Convolution

Correlation

- Correlation is a weighted moving average

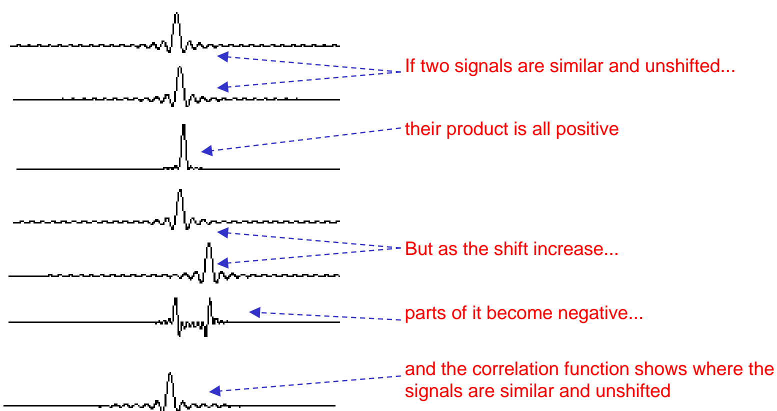
$$r(n) = \sum_k x(k) \times y(k+n)$$



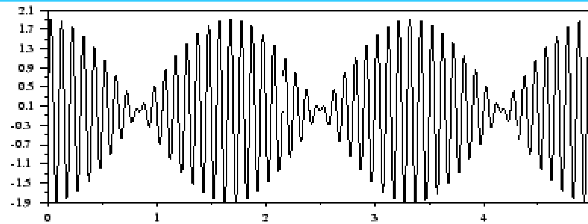
- Requires a lot of calculation
  - If one signal is of length *M* and the other is of length *N*, then we need *(N * M)* multiplications, to calculate the whole correlation function
    - Note that really, we want to multiply and then accumulate the result - this is typical of DSP operations and is called a *multiply & accumulate* operation

Correlation

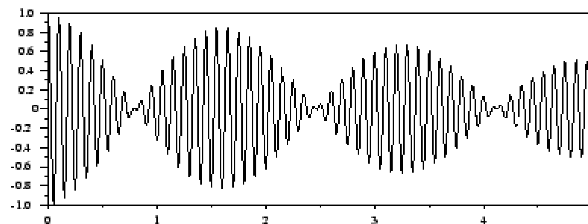
- Correlation is a maximum when two signals are similar in shape
- Correlation is a measure of the similarity between two signals as a function of time shift between them



Detecting Periodicity



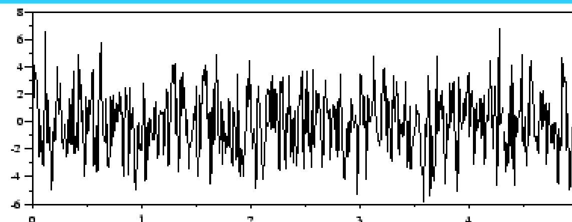
EEG signal



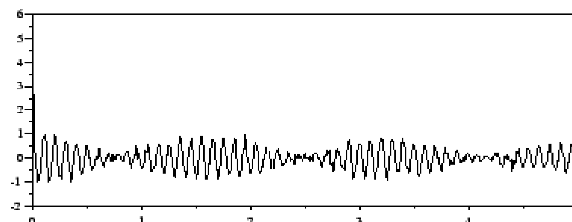
EEG autocorrelation

- Autocorrelation as a way to detect periodicity in signals

Detecting Periodicity



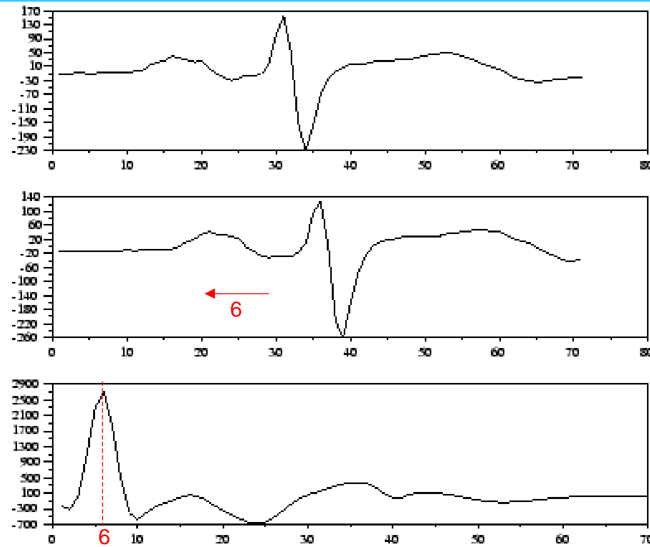
EEG signal with noise



EEG with noise autocorrelation

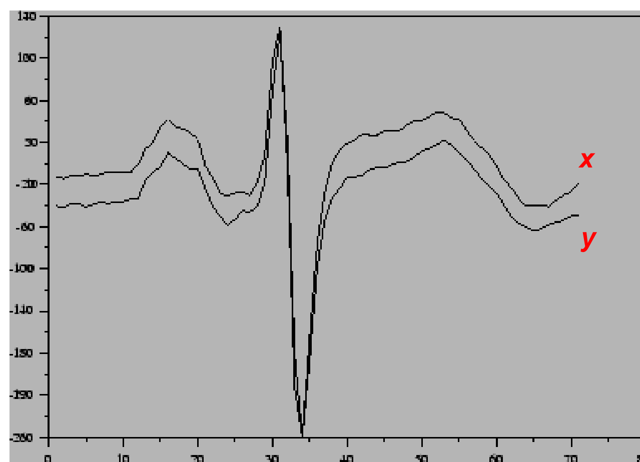
- Although a rhythm is not even visible (upper trace) it is detected by autocorrelation (lower trace)

Align Signals



Maurizio Palesi 21

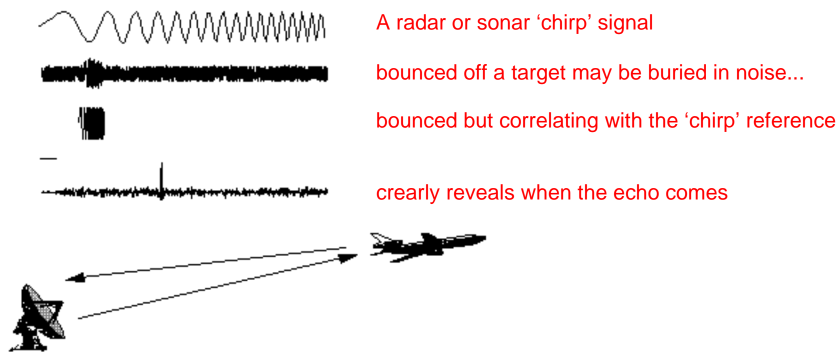
Align Signals



Maurizio Palesi 22

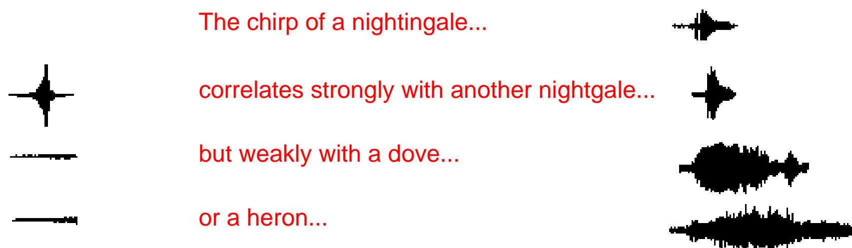
Cross correlation

- Cross correlation (correlating a signal with another) can be used to detect and locate known reference signal in noise



Cross Corelation to Identify a Signal

- Cross correlation (correlating a signal with another) can be used to identify a signal by comparison with a library of known reference signals



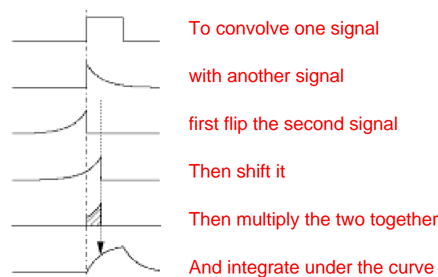
Cross Corelation to Identify a Signal

- Cross correlation is one way in which sonar can identify different types of vessel
  - ➔ Each vessel has a unique sonar *signature*
  - ➔ The sonar system has a library of pre-recorded echoes from different vessels
  - ➔ An unknown sonar echo is correlated with a library of reference echoes
  - ➔ The largest correlation is the most likely match

Convolution

- Correlation is a weighted moving average with one signal flipped back to front

$$r(n) = \sum_k x(k) \times y(k-n)$$



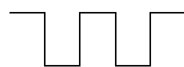
- Requires a lot of calculation
  - ➔ If one signal is of length *M* and the other is of length *N*, then we need (*N* * *M*) multiplications, to calculate the whole convolution function
    - ✓ We need to multiply and then accumulate the result - this is typical of DSP operations and is called a *multiply & accumulate* operation

Convolution vs. Correlation

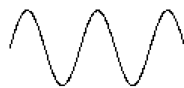
- Convolution is used for digital filtering
  - ➔ Convolving two signals is equivalent to *multiplying the frequency spectra* of the two signals together
    - ✓ It is easily understood, and is what we mean by filtering
  - ➔ Correlation is equivalent to *multiplying the complex conjugate of the frequency spectrum* of one signal by the frequency spectrum of the other
    - ✓ It is not so easily understood and so convolution is used for digital filtering
- Convolving by multiplying frequency spectra is called *fast convolution*

Fourier Transform

- The *Fourier Transform* is a mathematical procedure that allows to convert a signal from the time domain to the frequency domain
- Any signal or waveform could be made up just by adding together a series of sine waves with appropriate *amplitude* and *phase*



A square wave can be made by adding...

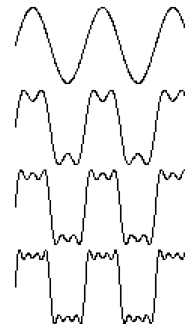


the fundamental

minus 1/3 of the third harmonic

plus 1/5 of the fifth harmonic...

minus 1/7 of the 7th harmonic...



Fourier Transform

- The Fourier transform is an equation to calculate the *frequency*, *amplitude* and *phase* of each sine needed to make up any given signal
  - ➔ The *Fourier Transform* (FT) is a mathematical formula using integrals
  - ➔ The *Discrete Fourier Transform* (DFT) is a discrete numerical equivalent using sums instead of integrals
  - ➔ The *Fast Fourier Transform* (FFT) is just a computationally fast way to calculate the DFT
- The Discrete Fourier Transform involves a summation

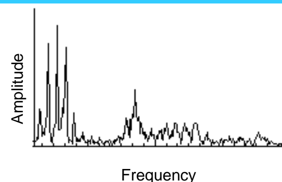
$$H(f) = \sum_k c[k] \times e^{-2\pi jk(f\Delta)}$$

- DFT and the FFT involve a lot of multiply and accumulate the result
  - ➔ This is typical of DSP operations and is called a *multiply & accumulate* operation

Maurizio Palesi 29

Frequency Spectra

A recording of speech



Can be analysed to show the spectrum

- Using the Fourier transform, any signal can be analysed into its frequency components

Maurizio Palesi 30

Frequency Spectra

- With some signals it is easy to see that they are composed of different frequencies
  - ➔ A chord played on the piano is obviously made up of the different pure tones generated by the keys pressed
  - ➔ You can use a piano as an acoustic spectrum analyser to show that a hand clap has a frequency spectrum
    - ✓ Open the lid of the piano and hold down the 'loud' pedal
    - ✓ Clap your hands loudly over the piano
    - ✓ You will hear (and see) the strings vibrate to echo the clap sound
      - – The strings that vibrate show the frequencies
      - – The amount of vibration shows the amplitude

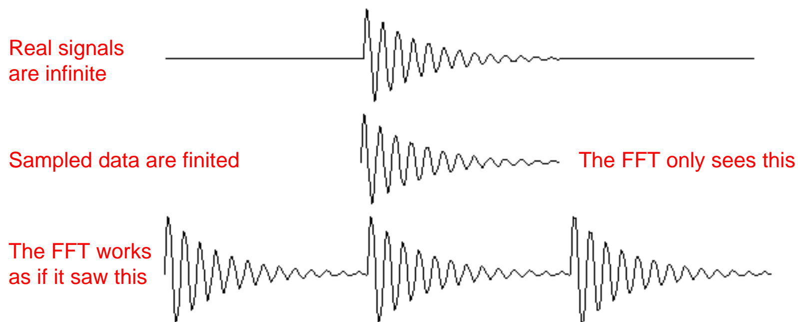
Short Term Fourier Transform

$$H(f) = \sum_k c[k] \times e^{-2\pi jk(f\Delta)}$$

- Fourier transform
  - ➔ The signal is analysed over all time
  - ➔ ...an infinite duration
- Short Time Fourier Transform (*STFT*)
  - ➔ Evaluates the way frequency content changes with time

Short Signals

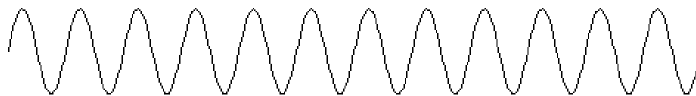
- If we measure the signal for a short time
  - ➔ What happened to the signal before and after we measured it?
  - ➔ FFT makes an assumption about what happened before and after
  - ➔ The FFT assumes the data are periodic for all time



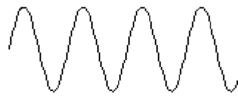
Maurizio Palesi 33

Short Signals

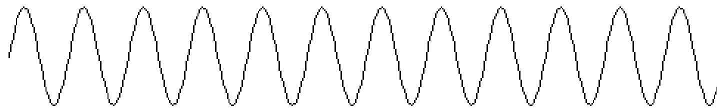
This is the signal



Period fits the sample time



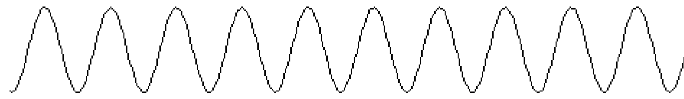
The FFT works as if it saw this



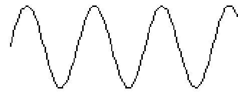
Maurizio Palesi 34

Short Signals

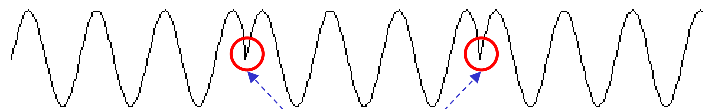
This is the signal



Not quite an integral number of cycles fit into the total duration of the measurement



The FFT works as if it saw this

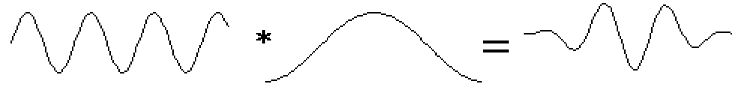


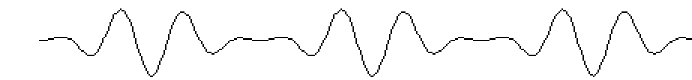
Glitches

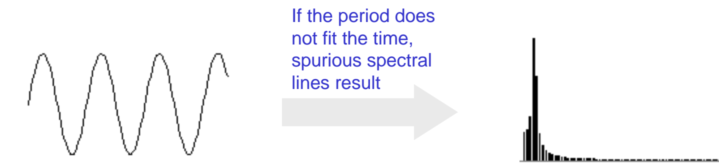
Short Signals

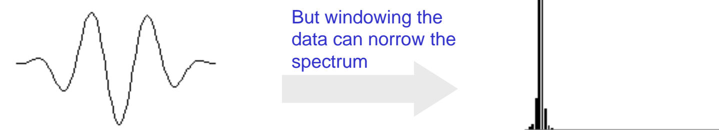
- If the period exactly fits the measurement time
  - ➔The frequency spectrum is correct
- If the period does not match the measurement time
  - ➔The frequency spectrum is incorrect - it is broadened
- The size of the glitch depends on when the first measurement occurred in the cycle
  - ➔The broadening will change if the measurement is repeated

Windowing









Filtering

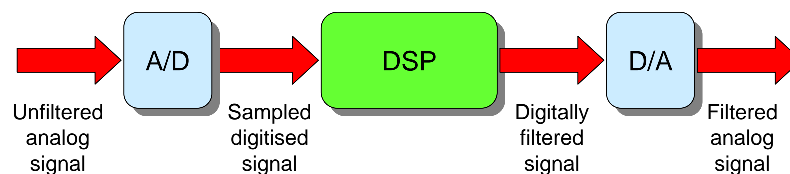
- The function of a filter is to remove unwanted parts of the signal
  - Random noise
  - Extract useful parts of the signal
    - Components lying within a certain frequency range
- Filters
  - Analog
  - Digital

Analog Filters

- An *analog filter* uses analog electronic circuits
  - ➔Use components such as resistors, capacitors and op amps
- Widely used in such applications
  - ➔Noise reduction
  - ➔Video signal enhancement
  - ➔Graphic equalisers in hi-fi systems
  - ➔..., and many other areas

Digital Filters

- A *digital filter* uses a digital processor to perform numerical calculations on sampled values of the signal
  - ➔Specialised DSP chip

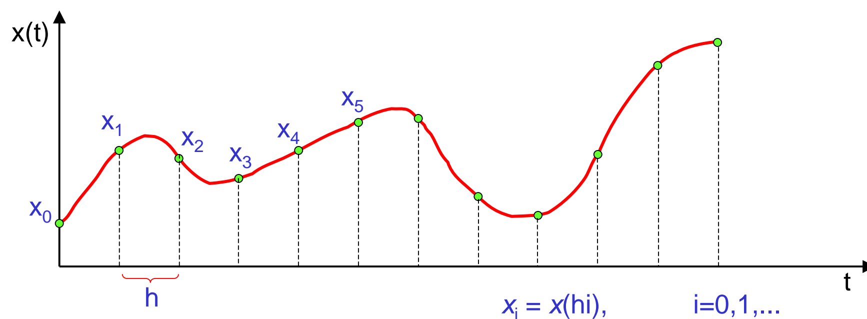


Advantage of Digital Filters

- Programmability
  - ➔ The digital filter can easily be changed without affecting the circuitry
- Analog filter circuits are subject to drift and are dependent on temperature
- Digital filters can handle low frequency signals accurately
- As the speed of DSP technology continues to increase, digital filters are being applied to high frequency signals in the RF domain
- Versatility
  - ➔ Adapt to changes in the characteristics of the signal

Maurizio Palesi 41

Operation of Digital Filters



$x_i = x(hi)$, i=0,1,...

$y_n = a_0x_n + a_1x_{n-1} + a_2x_{n-2} + \ldots + a_mx_{n-m}$ *m*-order filter

Maurizio Palesi 42

Recursive and Non-Recursive

$$y_n = a_0x_n + a_1x_{n-1} + a_2x_{n-2} + \dots + a_mx_{n-m}$$

Non recursive filter
order = m

$$y_n = a_0x_n + a_1x_{n-1} + a_2x_{n-2} + \dots + a_mx_{n-m} + b_1y_{n-1} + b_2y_{n-2} + b_3y_{n-3} + \dots + b_py_{n-p}$$

Recursive filter
order = $\max\{m,p\}$

FIR and IIR Filters

- Finite Impulse Response (*FIR*)
  - Non-recursive filter
- Infinite Impulse Response (*IIR*)
  - Recursive filter

Recursive vs. Non-Recursive

$y_n = a_0x_n + a_1x_{n-1} + a_2x_{n-2} + \ldots + a_mx_{n-m}$

$y_n = a_0x_n + a_1x_{n-1} + a_2x_{n-2} + \ldots + a_mx_{n-m} + b_1y_{n-1} + b_2y_{n-2} + b_3y_{n-3} + \ldots + b_py_{n-p}$

- It might seem that recursive filters require more calculations to be performed
  - To achieve a given frequency response characteristic using a recursive filter generally requires a much lower order filter
    - Fewer terms to be evaluated by the processor

Example of Recursive Filter

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

Recursive	Non-recursive
$y_0 = x_0 + y_{-1}$	$y_0 = x_0 + y_{-1}$
$y_1 = x_1 + y_0$	$y_1 = x_1 + x_0 + y_{-1}$
$y_2 = x_2 + y_1$	$y_2 = x_2 + x_1 + x_0 + y_{-1}$
$y_3 = x_3 + y_2$	$y_3 = x_3 + x_2 + x_1 + x_0 + y_{-1}$
…	…
$y_{10} = x_{10} + y_9$	$y_{10} = x_{10} + x_9 + x_8 + \ldots + x_3 + x_2 + x_1 + x_0 + y_{-1}$

Symmetrical Form

$$y_n = (a_0x_n + a_1x_{n-1} - b_1y_{n-1})/b_0$$

$$b_0y_n + b_1y_{n-1} = a_0x_n + a_1x_{n-1}$$

$$b_0y_n + b_1y_{n-1} + \ldots + b_py_{n-p} = a_0x_n + a_1x_{n-1} + \ldots + a_mx_{n-m}$$

Transfer Function (IIR)

- z^{-1} unit delay operator
  - ➔ $z^{-1}x_n = x_{n-1}$
  - ➔ $z^{-2}x_n = x_{n-2}$

$$b_0y_n + b_1y_{n-1} + b_2y_{n-2} = a_0x_n + a_1x_{n-1} + a_2x_{n-2}$$

$$(b_0 + b_1z^{-1} + b_2z^{-2})y_n = (a_0 + a_1z^{-1} + a_2z^{-2})x_n$$

$$\frac{y_n}{x_n} = \frac{(a_0 + a_1z^{-1} + a_2z^{-2})}{(b_0 + b_1z^{-1} + b_2z^{-2})}$$

Transfer function for a second-order recursive (IIR) filter

Transfer Function (FIR)

- z^{-1} unit delay operator
  - ➔ $z^{-1}x_n = x_{n-1}$
  - ➔ $z^{-2}x_n = x_{n-2}$

$$y_n = a_0 x_n + a_1 x_{n-1} + a_2 x_{n-2}$$

$$y_n = (a_0 + a_1 z^{-1} + a_2 z^{-2}) x_n$$

$$\frac{y_n}{x_n} = a_0 + a_1 z^{-1} + a_2 z^{-2}$$

Transfer function for a second-order non-recursive (FIR) filter

Digital Filter Equation

- Output from a digital filter is made up from previous inputs and previous outputs, using the operation of convolution

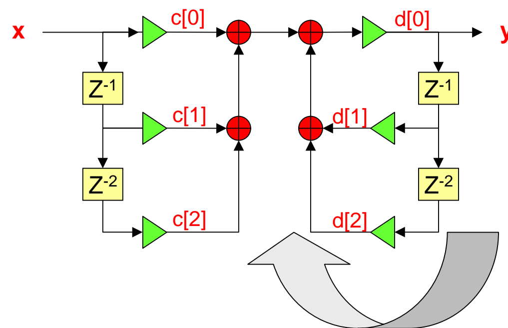
Output — Previous input — Previous output

$$y[n] = \sum_k c[k] \times x[n-k] + \sum_j d[j] \times y[n-j]$$

Coefficients

Digital Filter Equation

$$y[n]=\sum_k c[k]\times x[n-k]+\sum_j d[j]\times y[n-j]$$



Filter Frequency Response

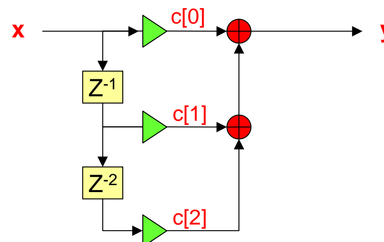
$$H(f)=\frac{\sum_k c[k]\times e^{-2\pi jk(f\Delta)}}{1-\sum_k d[k]\times e^{-2\pi jk(f\Delta)}}$$

- When designing a digital filter we want to do the *inverse* operation
  - ➔ Calculate the filter coefficients having first defined the desired frequency response
  - ➔ Additional constraint
    - ✓ We usually want to design a filter that meets the requirement but which requires the least possible amount of computation
      - Using the smallest number of coefficients

FIR Filters

- The filter equation is simplified by excluding the possibility of feedback

$$y[n] = \sum_k c[k] \times x[n-k]$$



- The filter response is

$$H(f) = \sum_k c[k] \times e^{-2\pi jk(f\Delta)}$$

  - ➔ This frequency response is just the Fourier transform of the filter coefficients

FIR Filters

- So the coefficients for an FIR filter can be calculated simply by taking the *Inverse Fourier Transform* of the desired frequency response
- Here is a recipe for calculating FIR filter coefficients
  - ➔ Decide upon the desired frequency response
  - ➔ Calculate the inverse Fourier transform
  - ➔ Use the result as the filter coefficients
- …BUT…

FIR Filters

- ...BUT...
  - ➔ The iFT has to take samples of the continuous desired frequency response
  - ➔ To define a sharp filter needs closely spaced frequency samples - so a lot of them
  - ➔ So the iFT will give us a lot of filter coefficients
  - ➔ But we don't want a lot of filter coefficients
- A better recipe for calculating FIR filter is:
  - ➔ Specify the desired frequency response using lots of samples
  - ➔ Calculate the inverse iFT
  - ➔ This gives us a lot of filter coefficients
  - ➔ So truncate the filter coefficients to give us less
  - ➔ Then calculate the FT of the truncated set of coefficients to see if it still matches our requirement
- ...BUT...

Maurizio Palesi 55

FIR Filters

- Truncating the filter coefficients means we have a truncated signal...
- ...And a truncated signal has a broad frequency spectrum
- Applying a window function is a simple way to sharpen up the frequency spectrum of a truncated signal
  - ➔ Specify the desired frequency response using lots of samples
  - ➔ Calculate the inverse Fourier transform
  - ➔ This gives us a lot of filter coefficients
  - ➔ So truncate the filter coefficients to give us less
  - ➔ Apply a window function to sharpen up the filter's frequency response
  - ➔ Then calculate the Fourier transform of the truncated set of coefficients to see if it still matches our requirement
- This is called the window method of FIR filter design

Maurizio Palesi 56

Summary so far

- Converting analogue signals
  - ➔Aliasing and Quantisation problems
- Time domain processing
  - ➔Correlation and Convolution
- Frequency domain processing
  - ➔Fourier transform and windowing
- Digital filters
  - ➔FIR and IIR filters
- More reading
  - ➔http://www.bores.com

Part-II
Signal Processing using a DSP

Maurizio Palesi

DSP Processors

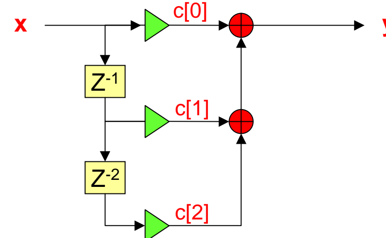
- Characteristic features of DSP processors
- Special features for arithmetic
- Addressing modes
- Data formats
- Programming a DSP

Characteristics of DSP Processors

- DSP processors are mostly designed with the same few basic operations in mind
- They share the same set of basic *characteristics*
  - ➔ Specialised high speed arithmetic
  - ➔ Data transfer to and from the real world
  - ➔ Multiple access memory architectures

Characteristics of DSP Processors

- The basic DSP operations
  - Additions and multiplications
    - Fetch two operands
    - Perform the addition or multiplication (usually both)
    - Store the result or hold it for a repetition
  - Delays
    - Hold a value for later use
  - Array handling
    - Fetch values from consecutive memory locations
    - Copy data from memory to memory



Characteristics of DSP Processors

- To suit these fundamental operations DSP processors often have
  - Parallel multiply and add
  - Multiple memory accesses (to fetch two operands and store the result)
  - Lots of registers to hold data temporarily
  - Efficient address generation for array handling
  - Special features such as delays or circular addressing

DSP Processors: Mathematics

Single bus

32

32

40

Type conversion

45

40

P

S

40

40

a0

a1

a2

a3

32

Lucent DSP32C datapath

Multiply and accumulate operate in parallel

Address Generation

- The ability to generate new addresses efficiently is a characteristic feature of DSP processors
- Usually, the next needed address can be generated during the data fetch or store operation, and with no overhead
- DSP processors have rich sets of address generation operations

*rP	register indirect	read the data pointed to by the address in register rP
*rP++	postincrement	having read the data, postincrement the address pointer to point to the next value in the array
*rP--	postdecrement	having read the data, postdecrement the address pointer to point to the previous value in the array
*rP++rI	register postincrement	having read the data, postincrement the address pointer *by the amount held in register rI* to point to *rI* values further down the array
*rP++rIr	bit reversed	having read the data, postincrement the address pointer to point to the next value in the array, *as if the address bits were in bit reversed order*

Bit Reversed Addressing

- DSPs are tightly targeted to a small number of algorithms
  - ➔ It is surprising that an addressing mode has been specifically defined for just one application (the FFT)

Addresses generated by a radix-2 FFT

0 (000_2)	➔	0 (000_2)
1 (001_2)	➔	4 (100_2)
2 (010_2)	➔	2 (010_2)
3 (011_2)	➔	6 (110_2)
4 (100_2)	➔	1 (001_2)
5 (101_2)	➔	5 (101_2)
6 (110_2)	➔	3 (011_2)
7 (111_2)	➔	7 (111_2)

- Whithout special support such address transformations would
  - ➔ Take an extra memory access to get the new address
  - ➔ Involve a fair amount of logical instructions

Memory Addressing

- As DSP programmers migrate toward larger programs, they are more attracted to compilers
  - ➔ Such compilers are not able to fully exploit such specific addressing modes
  - ➔ DSP community routinely uses library routines
    - ✓ Programmers may benefit even if they write at a high level

Addressing mode	Percent
Immediate	30,02%
Displacement	10,82%
Register indirect	17,42%
Direct	11,99%
Autoincrement, postincrement	18,84%
Autoincrement, preincrement with 16 bit immediate	0,77%
Autoincrement, preincrement with circular addresing	0,08%
Autoincrement, postincrement by contents of AR0	1,54%
Autoincrement, postincrement by contents of AR0, with circular addressing	2,15%
Autodecrement, postdecrement	6,08%

~90% (Immediate through Autoincrement, postincrement)

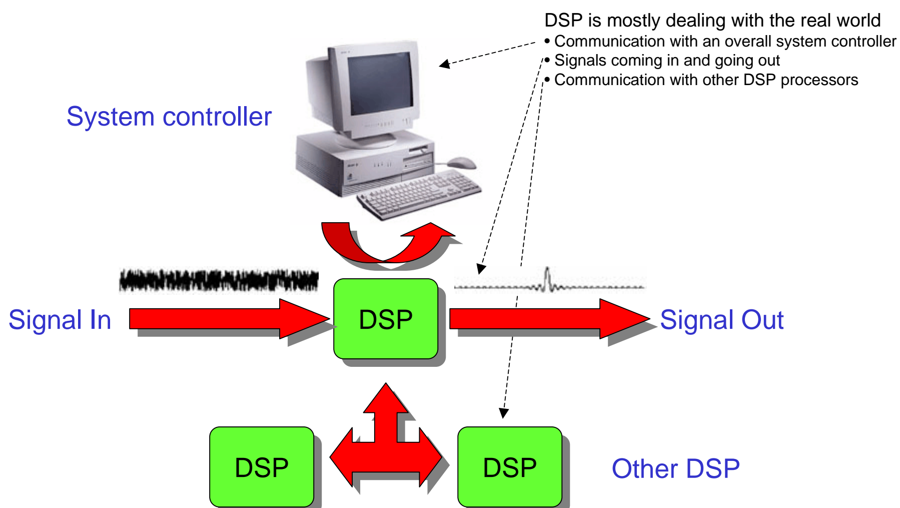
Compiler for DSPs

- Despite the well documented advantages in programmer productivity and software maintenance...

TMS320C54 D (C54) for DSPstone kernels	Ratio to assembly in execution time (>1 means slower)	Ratio to assembly in code space (>1 means bigger)
Convolution	11,8	16,5
FIR	11,5	8,7
Matrix 1x3	7,7	8,1
FIR2dim	5,3	6,5
Dot product	5,2	14,1
LMS	5,1	0,7
N real update	4,7	14,1
IIR n biquad	2,4	8,6
N complex update	2,4	9,8
Matrix	1,2	5,1
Complex update	1,2	8,7
IIR one biquad	1,0	6,4
Real update	0,8	15,6
C54 geometric mean	*3,2*	*7,8*

TMS320C6203 (C62) for EEMBC Telecom kernels	Ratio to assembly in execution time (>1 means slower)	Ratio to assembly in code space (>1 means bigger)
Convolution encoder	44,0	0,5
Fixed-point complex FFT	13,5	1,0
Viterbi GSM decoder	13,0	0,7
Fixed-point bit allocation	7,0	1,4
Autocorrelation	1,8	0,7
C62 geometric mean	*10,0*	*0,8*

DSP Processors: Input/Output



Signals

- They are usually handled by high speed synchronous serial ports
- Serial ports are inexpensive
  - Having only two or three wires
  - Well suited to audio or telecommunications data rates up to 10 Mbit/s
  - Usually operate under DMA
    - Data presented at the port is automatically written into DSP memory without stopping the DSP

Data Formats

- DSP processors store data in *fixed* or *floating point* formats

Integer

0	1	0	1	0	0	1	1
-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$= 2^6 + 2^4 + 2^1 + 2^0 = 83$

Fixed point

0	1	0	1	0	0	0	0
-2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}

$= 2^{-1} + 2^{-3} = 0.5 + 0.125 = 0.625$

- The programmer has to make some decisions
  - If a fixed point number becomes too large for the available word length, he has to scale the number down, by shifting it to the right
  - If a fixed point number is small, he has to scale the number up, in order to use more of the available word length

Fixed Point

- *Fixed point* can be thought of as just low-cost floating point
  - ➔ It does not include an exponent in every word
  - ➔ No hw that automatically aligns and normalizes operands
    - ✓ DSP programmer take cares to keep the exponent in a separate variable
    - ✓ Often this variable is shared by a set of fixed-point variables
      - – Blocked floating point

Floating Point

- *Floating point* format has the remarkable property of automatically scaling all numbers by moving, and keeping track of, the binary point so that all numbers use the full word length available but never overflow

Mantissa

0	1	1	0	1	0	0	0	0
-2^{-1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}

Exponent

0	1	1	0
-2^{3}	2^{2}	2^{1}	2^{0}

Mantissa = $2^0 + 2^{-1} + 2^{-3}$= 1 + 0.5 + 0.125 = 1.625

Exponent = $2^2 + 2^1 = 6$

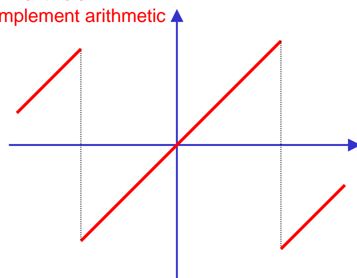
Decimal value = 1.625×2^6

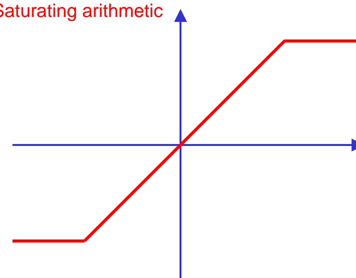
Data Formats

- In Floating Point the HW automatically *scales* and *normalises* every number
- Errors due to truncation and rounding depend on the size of the number
- These errors can be seen as a source of quantisation noise
  - Then the noise is modulated by the size of the signal
  - The signal dependend modulation of the noise is undesiderable because is audible
    - The audio industry prefers to use fixed point DSP processors over floating point

Saturating Arithmetics

- DSPs are often used in real-time applications
  - No exception on arithmentic overflow
    - It could miss an event
  - To support such an environment, DSP architectures use saturating arithmetic
    - If the result is too large to be represented, it is set to the largest representable number





Programming a DSP Processor

- A simple FIR filter program
- Using pointers
- Avoiding memory bottlenecks
- Assembler programming

A Simple FIR Filter

- The simple FIR filter equation is

$$y[n] = \sum_k c[k] \times x[n-k]$$

- Which can be implemented quite directly in *C* language

```
y[n] = 0.0;
for (k=0; k<N; k++)
    y[n] = y[n] + c[k] * x[n-k];
```

Accessed repeatedly

Accessing by array index is inefficient

Arithmetic is needed to calculate this array index

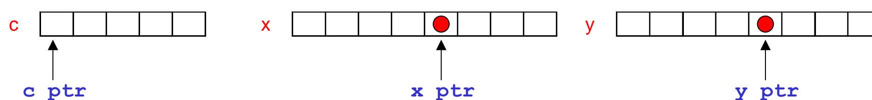
Problem in Addressing

- Five operation to calculate the address of the element **x[n-k]**
  - ➔Load the start address of the table in memory
  - ➔Load the value of the index *n*
  - ➔Load the value of the index *k*
  - ➔Calculate the offset [*n* - *k*]
  - ➔Add the offset to the start address of the array
- Only after all five operations can the compiler actually read the array element

Using Pointers

```
y[n] = 0.0;
for (k=0; k<N; k++)
   y[n] = y[n] + c[k] * x[n-k];
```

```
float *y_ptr, *c_ptr, *x_ptr;
y_ptr = &y[n];
for (k=0; k<N; k++)
   *y_ptr = *y_ptr + *c_ptr++ * *x_ptr--;
```



Using Pointers

```
float *y_ptr, *c_ptr, *x_ptr;
y_ptr = &y[n];
for (k=0; k<N; k++)
   *y_ptr = *y_ptr + *c_ptr++ * *x_ptr--;
```

- Each pointer still has to be initialised
  - ➔ But only once, before the loop
  - ➔ Not requiring any arithmetic to calculate offsets
- Using pointers is more efficient than array indices on any processor
  - ➔ It is especially efficient for DSP processors
    - ✓ Address increments often come for free

Using Pointers

*rP	register indirect	read the data pointed to by the address in register rP
*rP++	postincrement	having read the data, postincrement the address pointer to point to the next value in the array
*rP--	postdecrement	having read the data, postdecrement the address pointer to point to the previous value in the array
*rP++rI	register postincrement	having read the data, postincrement the address pointer *by the amount held in register rI* to point to *rI* values further down the array

- The address increments are performed in the same instruction as the data access to which they refer
  - ➔ They incur no overhead at all
  - ➔ Most DSP processors can perform two or three address increments for free in each instruction
    - ✓ So the use of pointers is crucially important for DSP processors

Limiting Memory Accesses

```
float *y_ptr, *c_ptr, *x_ptr;
y_ptr = &y[n];
for (k=0; k<N; k++)
    *y_ptr = *y_ptr + *c_ptr++ * *x_ptr--;
```

Store — Load — Load — Load

- *Four* memory accesses
  - ➔ Even without counting the need to load the instruction, this exceeds the capacity of a DSP processor
- Fortunately, DSP processors have lots of registers

Maurizio Palesi 81

Limiting Memory Accesses

```
register float temp;
temp = 0.0;
for (k=0; k<N; k++)
    temp = temp + *c_ptr++ * *x_ptr--;
```

This initialization is wasted!

```
register float temp;
temp = *c_ptr++ * *x_ptr--;
for (k=1; k<N; k++)
    temp = temp + *c_ptr++ * *x_ptr--;
```

Maurizio Palesi 82

Introduction

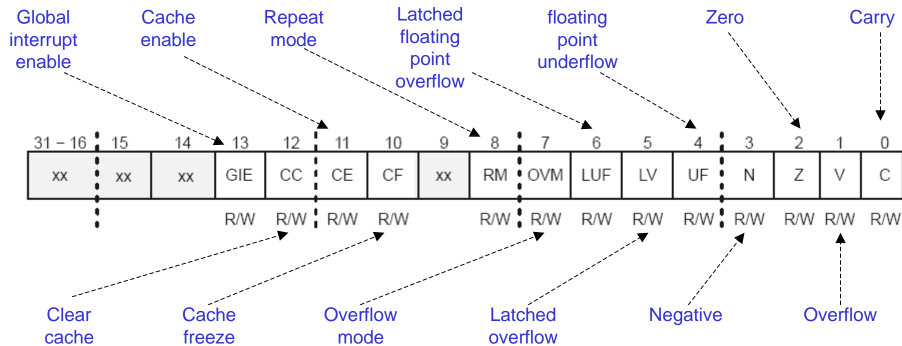
- The TMS320C3x generation of DSPs are high performance 32-bit floating-point devices in the TMS320 family
- Extensive internal busing
  - → 2 operands from memory & 2 operands from the registers file
- Powerful DSP instruction set
- 60 MFLOPS
- High degree of on-chip parallelism
  - → Up to 11 operations in a single instruction

General Features

- General-purpose register file
- Program cache
- Dedicated auxiliary register arithmetic units (ARAU)
- Internal dual-access memories
- Direct memory access (DMA)
- Short machine-cycle time

Status Register (ST)

- Contains global information about the state of the CPU
  - ➔Operations usually set the condition flags of the status register according to whether the result is 0, negative, etc.



Repeat Counter (RC) and Block Repeat (RS,RE)

- **RC** is a 32-bit register that specifies the number of times a block of code is to be repeated when a block repeat is performed
  - ➔If RC=n, the loop is executed n+1 times
- **RS** register contains the starting address of the program-memory block to be repeated when the CPU is operating in the repeat mode
- **RE** register contains the ending address of the program-memory block to be repeated when the CPU is operating in the repeat mode

Instruction Cache

- 64×32-bit instruction cache
  - ➔ 2-way set associative
  - ➔ LRU replacement policy
- It allows the use of slow, external memories while still achieving single-cycle access performances
- The cache also frees external buses from program fetches so that they can be used by the DMA or other system elements

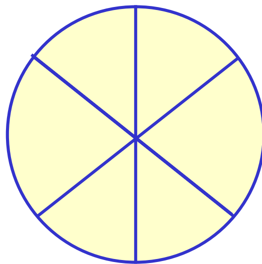
Addressing Modes

- Five types of addressing
  - ➔ Register addressing
  - ➔ Direct addressing
  - ➔ Indirect addressing
  - ➔ Immediate addressing
  - ➔ PC-relative addressing
- Plus two specialized addressing modes
  - ➔ Circular addressing
  - ➔ Bit-reverse addressing

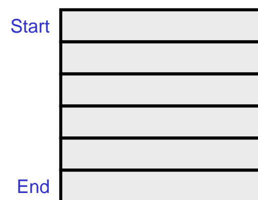
Circular Addressing

- Many DSP algorithms, such as convolution and correlation, require a circular buffer in memory
- In convolution and correlation, the circular buffer acts as a sliding window that contains the most recent data to process
- As new data is brought in, the new data overwrites the oldest data

Logical representation

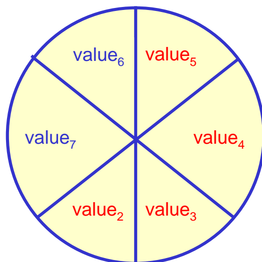


Physical representation

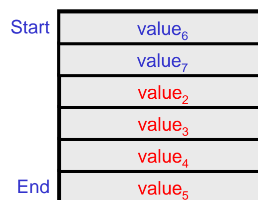


Circular Addressing

Logical representation



Physical representation

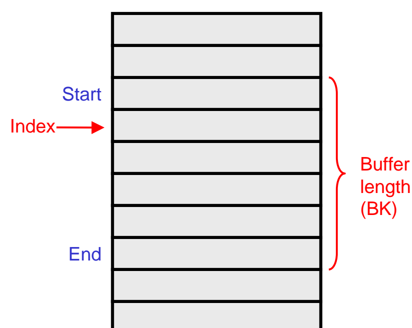


Implementation

- **BK** ← Length of the circular buffer
  - → (16 bit, <64K)
- The *K* LSB of the start address of the buffer must be 0
  - → *K* is such that 2^K > buffer length

Length of buffer	BK register value	Starting address of buffer
31	31	xxxxxxxxxxxxxxxxxxxxx00000
32	32	xxxxxxxxxxxxxxxxxxxx000000
1024	1024	xxxxxxxxxxxxxxx0000000000

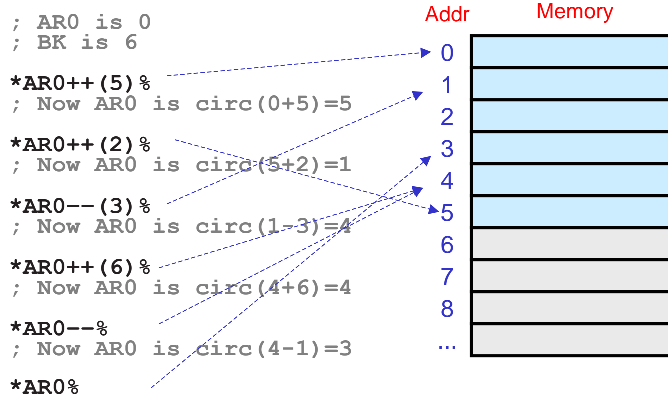
Algorithm for Circular Addressing




```
circ(index, step) =
   if (0 ≤ index+step < BK)
      index = index+step;
   else if (index+step ≥ BK)
      index = index+step-BK;
   else
      index = index+step+BK;
```

Circular Addressing - Example

```
*ARn++(disp)%    ; addr = ARn
                 ; ARn = circ(ARn+disp)
```



Parallel Operations

- The 13 parallel-operations instructions make a high degree of parallelism possible
- Some of the ’C3x instructions can occur in pairs that are executed in parallel
  - ➔ Parallel loading of registers
  - ➔ Parallel arithmetic operations
  - ➔ Arithmetic/logical instructions used in parallel with a store instruction

Parallel Operations

- Parallel *arithmetic* with *store* instructions

Mnemonic	Description
ABSF || STF	Absolute value of a floating-point number and store floating-point value
ABSI || STI	Absolute value of an integer and store integer
ADDF3 || STF	Add floating-point values and store floating-point value
ADDI3 || STI	Add integers and store integer
AND3 || STI	Bitwise-logical AND and store integer
ASH3 || STI	Arithmetic shift and store integer
FIX || STI	Convert floating-point to integer and store integer

••••••••••••• Many other

Parallel Operations

- Parallel *load* instructions

Mnemonic	Description
LDF || LDF	Load floating-point value
LDI || LDI	Load integer

- Parallel *multiply* and *add/subtract* instructions

Mnemonic	Description
MPYF3 || ADDF3	Multiply and add floating-point value
MPYF3 || SUBF3	Multiply and subtract floating-point value
MPYI3 || ADDI3	Multiply and add integer
MPYI3 || SUBI3	Multiply and subtract integer

Matrix-Vector Multiplication

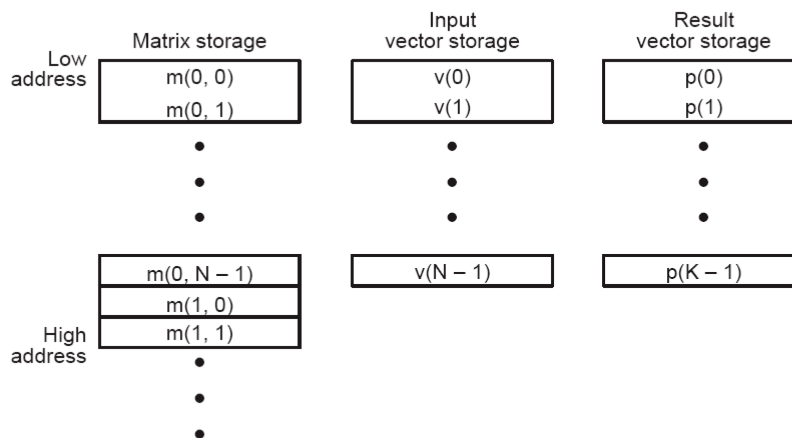
- $[P]_{K\times1}=[M]_{K\times N} \times [V]_{N\times1}$

```
for (i=0; i<K; i++)
{
        p[i] = 0.0;
        for (j=0; j<N; j++)
                p[i] = p[i] + m[i][j] * v[j];
}
```

```
for (i=0; i<K; i++)
{
        p[i] = m[i][0] * v[0];
        for (j=1; j<N; j++)
                p[i] = p[i] + m[i,j] * v[j];
}
```


Matrix-Vector Multiplication

- Data memory organization



Matrix-Vector Multiplication

```
* AR0 : ADDRESS OF M(0,0)
* AR1 : ADDRESS OF V(0)
* AR2 : ADDRESS OF P(0)
* AR3 : NUMBER OF ROWS - 1 (K-1)
* R1  : NUMBER OF COLUMNS - 2 (N-2)

MAT     LDI     R1,IR0              ; Number of columns-2 -> IR0
        ADDI    2,IR0               ; Number of columns -> IR0
ROWS    LDF     0.0,R2              ; Initialize R2
        MPYF3   *AR0++(1),*AR1++(1),R0 ; m(i,0) * v(0) -> R0
        RPTS    R1                  ; Multiply a row by a column
        MPYF3   *AR0++(1),*AR1++(1),R0 ; m(i,j) * v(j) -> R0
     || ADDF3   R0,R2,R2            ; m(i,j-1) * v(j-1) + R2 -> R2
        SUBI    1,AR3
        BNZD    ROWS                ; Counts the no. of rows left
Delay { ADDF    R0,R2               ; Last accumulate
slot  { STF     R2,*AR2++(1)        ; Result -> p(i)
      { NOP     *--AR1(IR0)         ; Set AR1 to point to v(0)
```


C Programming Tips

- After writing your application in C language, debug the program and determine whether it runs efficiently
- If the program does not run efficiently
  - Use the optimizer with –o2 or –o3 options when compiling
  - Use registers to pass parameters (–ms compiling option)
  - Use inlining (–x compiling option)
  - Remove the –g option when compiling
  - Follow some of the efficient code generation tips
    - Use register variables for often-used variables
    - Precompute subexpressions
    - Use *++ to step through arrays
    - Use structure assignments to copy blocks of data

Use Register Variables

- Exchange one object in memory with another

```
register float *src, *dest, temp;

do {
      temp = *++src;
      *src = *++dest;
      *dest = temp;
} while (--n);
```


Precompute Subexpression and use *++

```
main() {
      float a[10], b[10];
      int i;
      for (i = 0; i < 10; ++i)
            a[i] = (a[i] * 20) + b[i];
}
```

19 cycles

```
main() {
      float a[10], b[10];
      int i;
      register float *p = a, *q = b;
      for (i = 0; i < 10; ++i)
            *p++ = (*p * 20) + *q++;
}
```

12 cycles

Structure Assignments

- The compiler generates very efficient code for structure assignments
  - ➔ Nest objects within structures and use simple assignments to copy them

```
int x1, y1, c1;
int x2, y2, c2;

x1 = x2;
y1 = y2;
c1 = c2;
```

```
struct Pixel {
    int x, y, c;
};

struct Pixel p1, p2;

p1 = p2;
```

Maurizio Palesi 103

Hints for Assembly Coding

- Use delayed branches
  - ➔ Delayed branches execute in a *single cycle*
  - ➔ Regular branches execute in *four cycles*
  - ➔ The next three instructions are executed whether the branch is taken or not
    - ✓ If fewer than three instructions are required, use the delayed branch and append NOPs
      - – A reduction in machine cycles still occurs

Maurizio Palesi 104

Hints for Assembly Coding

- Apply the repeat single/block construct
  - In this way, loops are achieved with no overhead
  - Note that using RPTS instruction the executed instruction is not refetched for execution
    - This frees the buses for operand fetches

Hints for Assembly Coding

- Use parallel instructions
- Maximize the use of registers
- Use the cache
- Use internal memory instead of external memory
- Avoid pipeline conflicts

Summary and Conclusions

- Characteristic features of DSP processors
- Special features for arithmetic & data formats
- Addressing modes
- Programming a DSP
- More reading
  - TMS320C3x General-Purpose Applications User's Guide
    - http://focus.ti.com/general/docs/tecdocs.tsp