Understanding the principles of digital systems & processes.

A course for amateur radio enthusiasts wishing to come to grips with micro controller programming.

By Mike Brink. ZS6MEG
www.zs6meg.co.za

1. Gates

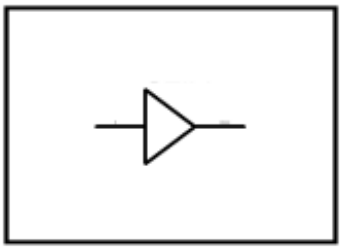
A gate, simply stated, is a switch. Logic systems, digital electronics, rely on the state of a switch to represent information. One can say that when a switch is open, there is no voltage at the output and no path for the current to flow. This represents a '0', 'Low' or a 'No' state. When the switch is closed, there is a voltage at the output and current can flow. This represents a '1', 'High' or 'Yes' state at the output.

The closing of a switch, enables electrons to flow through a conductor and causes a light to burn. The state of this switch, open or closed, therefore represents a condition. This condition is reflected by the light, which is burning or not burning. A cause and an effect.

This is the principle employed by digital system. An input into a digital system will result in an output. It starts off at a very basic and simple level with a small number of very simple building blocks. These building blocks are arranged in complex configurations to perform complex functions. The point however, is that no matter how complex the system becomes, the building blocks themselves remain very simple. It is like looking at a network of traffic lights that are spread throughout a city. Interlocked and sequenced to facilitate the smooth flow of traffic. Quite daunting when viewed in its entirety, yet very simple when you look at it in the context that it is just a collection of lights and switches. So, if you know what a light is and you know what a switch is, well then it becomes considerably easier to understand the whole.

And that is all that a digital system is, just a collection of lights and switches. Inputs and outputs. The key difference is that an output can control an input. A light can control a switch. If the light burns, it causes a switch to close, and if the light is off, then the switch that it is controlling is open. On, Off, Open, Closed, High, Low, Enabled, Disabled, Active, Inactive, Yes, No. These are the states that are indicated in a digital system by a bit of data which can be represented by a 0 or a 1, a logic level of 0Volts or 5Volts.

1.1 The buffer.



The buffer does not change the incoming signal. It the input is a 0, the output is a 0, if the input is a 1, then the output is a 1. Well then why use a buffer? There are numerous reasons.

In a digital system, speed is everything. The problem however is the length of the conductors. The longer the wires, the greater the inductive and capacitive (reactive) components become that interact with the signal being carried by the wire. This causes 'ringing', unwanted oscillations induced by the sharp switching edges of the digital lines as it moves from a high to a low and a low to a high state.

Varying Bus Structures

In a processing environment, there are two regions. Local and remote. Local devices are things like memories which we want close to the processor for high speed interactions. Remote devices are things like printers which, due to their physical nature are located a meter or many meters from the processor.

Data is carried from the processor to the memory along a data bus. Data is also carried to the printer along an extension of the same data bus. To isolate the printer data bus, we use a buffer. This prevents the reactive signals that would be generated by the printer cable from interacting with the CPU bus.

Fanout

Another problem is fanout. Sometimes there are a lot of components being driven by a signal line. This causes a degree of line loading. The more devices, the greater the load. So instead of using one line to drive 20 devices, we use one line to drive 4 buffers. Each buffer in turn drives 5 devices. We would therefore use a buffer to isolate devices that would load the signal line.

Propagation delay.

Every device has what is called a propagation delay. This is its reaction time. It is measured in periods as short as a few nano seconds, but the delay is there. It does become necessary from time to time to ensure that certain events, driven by an input, are sequenced to make one event happen before the other. One would use a buffer to slow

down a signal by making use of its propagation delay. This is useful when one needs a small delay. For longer delays, one would use a delay line, which is a buffer with a preset delay period.

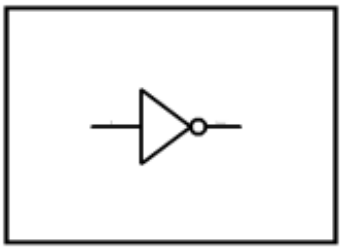
Waveform Shaping (Cleaning)

As a signal is transferred over a long line it tends to deteriorate. A buffer would be used to restore the sharp digital nature of the signal, eliminating signal degradation.

These are the key functions of a buffer.

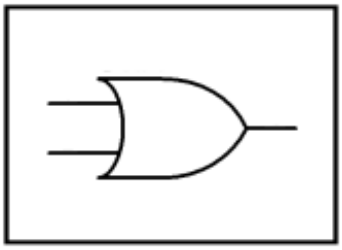
Interconnecting various bus structures
Catering for large fanout configurations
Introducing a propagation delay.
Cleaning up a digital waveform

1.2 Inverter



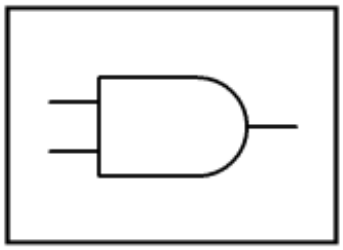
The function of the inverter is to invert the incoming signal. It changes a '1' into a '0' and a '0' into a '1'. Other than that it can also be used as a buffer. It is sometimes referred to as an inverting buffer.

1.3 The OR Gate



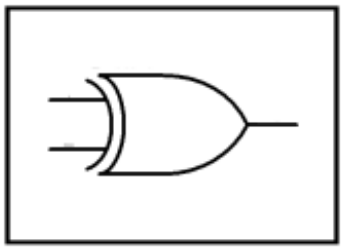
This is a decision gate. Imagine two bridges next to each other across a river. To get to the other side, one can cross over using one bridge or the other. In this way, the gate has two inputs. One can activate one input OR the other in order to generate an output. Another way of stating it is that the output will go high if either of the two inputs goes high.

1.4 The AND gate.



In this decision gate we imagine two bridges crossing two rivers one after the other. To get to the destination we must cross one bridge AND then we must cross the other bridge. So this gate has two inputs. Both inputs to this gate must go high for there has to be a high on the output. Another way of stating it is that in order for the output to go high, both inputs must go high.

1.5 The Exclusive OR gate



The exclusive or is used to indicate a difference between the two inputs. If both inputs are the same, the output is low. If either of the inputs is high and the other is low, if the two inputs are different, the output is high.

In summarizing,

The buffer output reflects the state of its input
The inverter's output reflects an inverted state of its input
The OR gates output will go high if either or both of its inputs go high
The AND gates output will go high only if both of its inputs go high
The Exclusive OR gates output will go high if the two inputs are not equal.

These five devices, these logical elements, are the function blocks around which all digital systems are constructed. No matter how large or complex the system, when one reduces it to its basic components, you will end up with one of these five building blocks.

2. Data

Now we look at the control signals. The input data. Every state is represented by a high or a low, a '0' or a '1'. This is referred to as a data bit. The plural of data is datum, but in digital systems we always refer to it in the singular. In computers we have numbers, letters, words, images, sounds and programs. These are all made up of data in various formats.

2.1 Integers

These data types are made up of multiple groups of data bits that basically represent numbers in what is called a binary progression of 2^n

The first bit, bit 0, can represent the presence of 1 item of data. 1. 2^0 = 1. When combined with a second bit, the two bits can represent 2 items of data. 2^1 = 2. In this way, following the binary progression, 2^0 = 1, 2^1=2, 2^3=4, 2^4=8. Bit 0 therefore represents a 1, bit 1 = 2, bit 2 = 4, bit 3 = 8 and so on for as many bits as you need to represent the data value you want to work with.

If I wanted to represent a 5, it would be represented by a high on bit 0 (=1) and a high on bit 3(=4). A 1 and a 4 together = 5. So in binary 1001 = 5 in decimal. There is a naming convention for data groupings.

1 data element = bit.
4 data bits = nybble
8 data bits = byte
16 data bits = half word
32 data bits = word
64 data bits = double word
128 data bits = long word

In this manner, groups of bits are used to represent integers. Positive value integers.

2.2 Signed and unsigned integers.

In the real world there are however negative as well as positive numbers that we would wish to represent with our binary value. By default, for the sake of explanation, we are going to use a 16-bit value for the sake of the explanation.

A 16 bit value, in integer format can be used to represent a value between 0 and 65535 (which is 2^16). If we had a 16 bit counter that was set to 0. (0000000000000000) and we incremented it, it would now be 1 (0000000000000001) . But if we decremented the counter, it would read –1 (1111111111111111) decrementing it again, it would read –2 (1111111111111110)

We therefore use the most significant bit, D15, to represent the sign of the value being a positive or a negative. 0 = positive, 1 = negative. Using this convention we can now represent a value of - 32768 to + 32767. To invert a number, we take the basic number, for example 1, and we add 32766 to it. 0000000000000001 + 1111111111111110 = 1111111111111111. Lets say we had a number of –1, (1111111111111111) and we added 2 (0000 0000 0000 0010) to it. We would then end up with a total of +1.

And so, we have signed and unsigned integers. We choose the integer range for the value of the number we wish to represent. If the number is less than 128, like a percentage, we choose 8 bits (from –128 to + 127) .

2.3 Real and integer.

The world not just defined by integers. Whatever unit of measure we choose, there is invariably a partial value. So, while integers may be fine for counting individuals in a room or houses in a street, the units of measure, like meters, grams, seconds etc, invariably have a fractional component to them. Like 2.75 meters.

These are referred to as real numbers.

With real numbers we can use one of two systems. Either fixed point or floating point fractional arithmetic. Fixed point and floating point values contain an inherent error of 1 bit of the smallest bit value used. The error is very small but it can tend to accumulate over many thousands of calculations.

A fixed point system would use 64 bits with the integer half of the value being represented by the upper 31 bits and the fractional portion being represented by the lower 32 bits. One can of course split the two halves in whatever ratio is most suitable for ones calculation. One could for example use a split of 40 to 23.

In the fixed point system. Using a 31/32 split, bit number 32 becomes the units. Bit 31 becomes ½ of a unit, bit 30 becomes a ¼ of a unit and so on down the scale where bit 0 is 1/4294967296 of a unit . Using this system, we have a number range from 2147483648 to –2147483648 with an accuracy of 0.0000000002 being a resolution of 10 places after the decimal point.

In a floating point system, we use the last 8 bits of the value as an exponential marker giving us a range of –2^23 to +2^23 with a range of –E127 to +E128 positions of the decimal point with an 8 digit resolution. Ie 0.0000001 E-127 to 0.00000001 E+128. Increasing the number of bits in the integer part increases the resolution while increasing the number of bits in the exponential side increases the range.

Using the floating point system allows us to achieve a wider dynamic range of values in a more compact data storage format than fixed point. The processing speed is hover, as

stated before, much slower, so for applications that do a large amount of number crunching, fixed point is much faster.

2.4 Binary coded decimal

This is the most inefficient system in terms of processing speed and data storage. It is however highly accurate and very easy to work with. When one uses BCD, one interacts with the system in a manner parallel to our method of thinking. 4+5 = 9. One byte for 4, one for 5 and one for 9. In applications where one processes a limited amount of data, like in micro controller applications, this system is very nice and easy to use, even if it a bit wasteful of space and processing resources. But, on a 10 MIPS (Million Instructions Per Second) processor, you would not know the difference.

In BCD arithmetic, we specify one byte for the units, tens, hundreds, thousands etc, etc for as far as we want to go to the left of the decimal point and we do likewise to the right hand side of the decimal point.

Implementing BCD arithmetic is relatively straight forward. The difference being that one does not have to convert a 32 bit binary value into a decimal value and vice versa. It is all ready for you in decimal format.

Advantages, ease of use, disadvantages, inefficient use of data and coding but not prohibitively so.

2.5 ASCII.

The American Standard Code for Information Interchange is a standards set of characters that is represented by a sequence of numbers. The value of an ‘A’ is 65. The value of a ‘0’ is 48. In order to get the lower case value, add 32 to the upper case value of a character. The ASCII code table uses the lower 128 numbers of the possible 256 numbers that can be represented by an 8 bit value. The upper 128 chars are called the extended ascii set and are usually reserved for graphics or application specific characters.

3. Mathematical Functions.

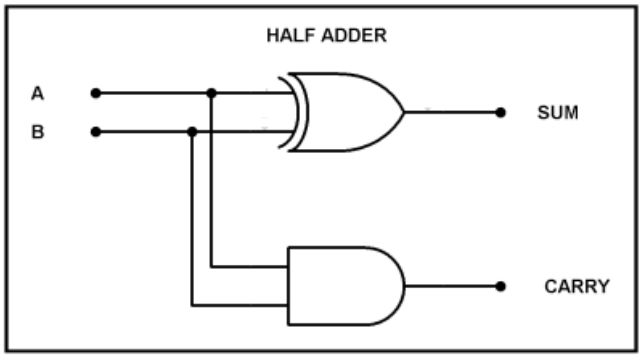
How the processor performs binary calculations.

3.1 The half adder
3.2 Addition using the full adder
3.3 Subtraction using the full adder
3.4 The D latch
3.5 Multiplication and division using the shift register
3.6 Successive approximation methods
3.7 Square roots
3.8 Trigonometric tabular functions
3.9 Digital signal processing and filtering examples

The micro-processor has what is called an arithmetic logic unit. The ALU. It is the function of the ALU to do calculations and make decisions. Decisions can be based on input data or on the result of calculations. A calculation can be performed as a result of a decision made by the ALU.

In this chapter we will look at the process followed by a CPU, using its ALU, to perform Addition, Subtraction, Multiplication, Division, Square, Root, Geometric, Filtering and Complex high level mathematical functions.

3.1 The Half Adder.

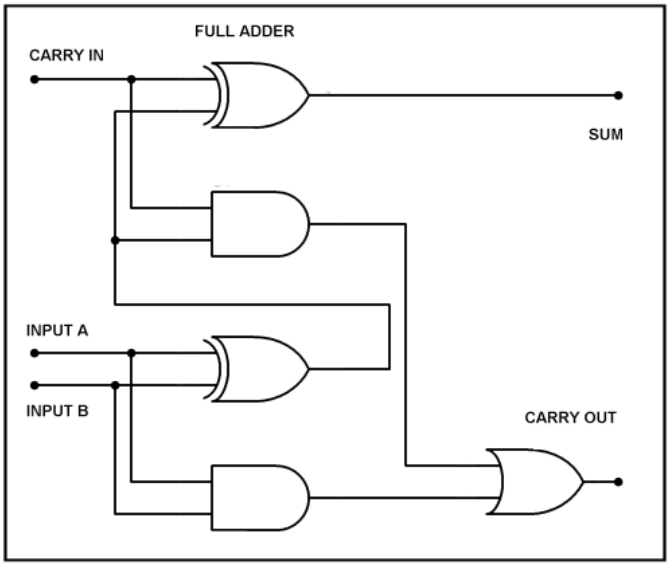


This circuit will add two binary values. A and B. It will give a sum output and it will provide a carry output. It functions according to the following truth table :

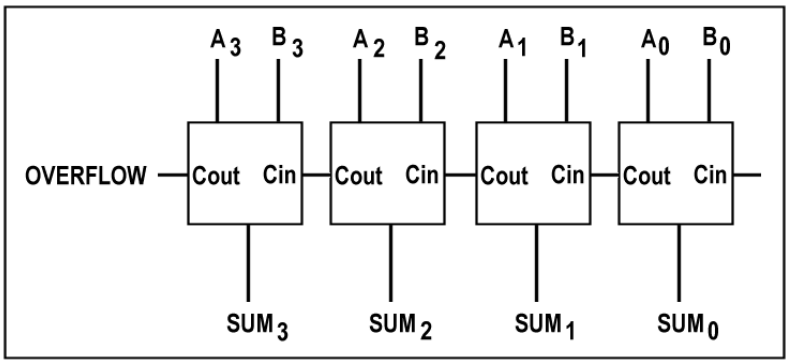
A	B	Sum	Carry
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

The half adder is the basic building block of all mathematical functions.

3.2 Addition using theThe Full Adder.



The full adder makes use of two half adders to make up a complete adding function block. It takes a 'carry in' from the previous full adder and passes a 'carry out' on to the next stage of the adding module. The final carry out will set an overflow flag indicating that the values added were greater than what the adder could handle.



3.3 Subtraction using the full adder.

Subtraction is purely the process of adding a negative number. We obtain the negative value of a number, as stated above, by inverting it and adding 1 to it. So, using 8 bits as an example, 7-6 = 1. so we take 7, 7 = '0000 0111' , and apply it to input A. Then we take 6 and invert it. 6 = '0000 0110', inverted is '1111 1001'

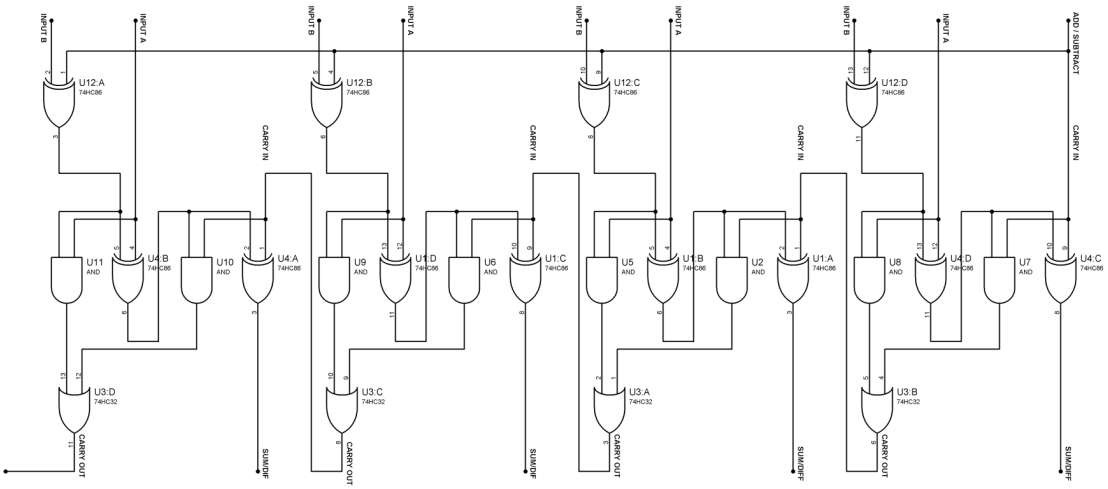
0000 0111
\+ 1111 1001
= 0000 0000 plus the overflow flag is set. Now we add 1.

+ 0000 0001
= 0000 0001. The result of 7 minus 6 is therefore 1.
A quick way to add a 1 without introducing extra code, is to set the carry input.

0000 0111 + carry in
+ 1111 1001
= 0000 0001 + carry out.

The carry out during the process of subtraction indicates that a borrow did not take place. A smaller number was subtracted from a larger number. The inverse of this indicator is called the borrow flag. An overflow during addition is called the carry flag.

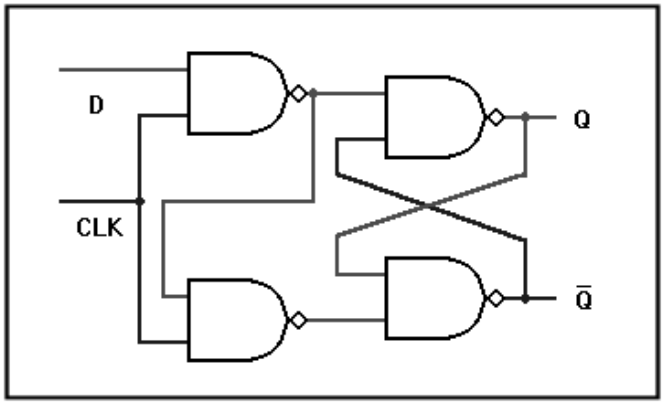
And a quick way to invert the input is to run every input through an inverter. In this case we use the exclusive or as a programmable inverter. If one input is low, the output will reflect the condition of the other input. If the input is high, the output will reflect the inverse of the other input. The high input will also be fed to the carry in indicating that this is a subtraction.



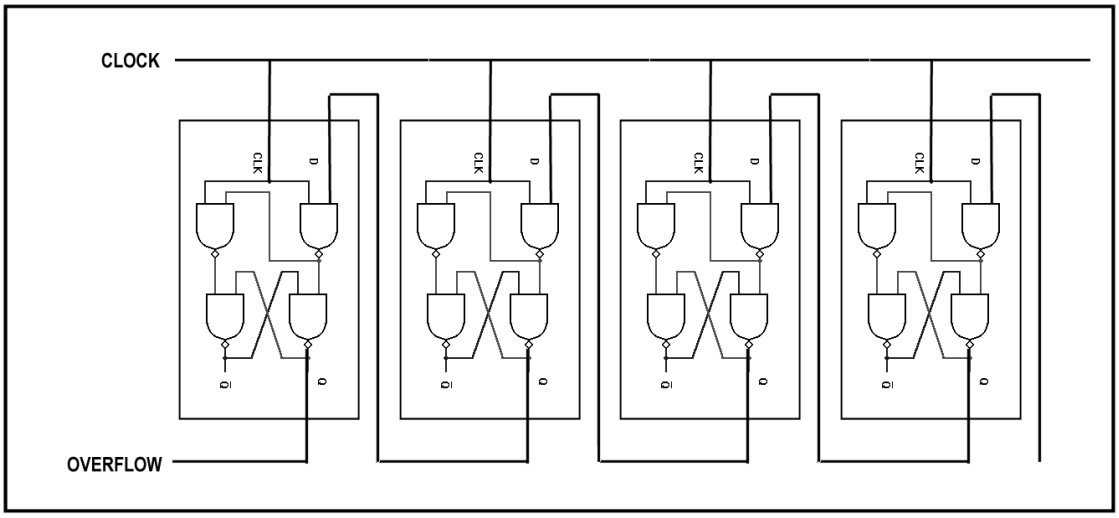
We use a control signal to indicate to this function block whether it is to add numbers together or to subtract numbers from each other. The same control signal goes to carry in, automatically adding 1 to the result in the case of subtraction. In this circuit, with subtraction, the inverted input is always subtracted from the non-inverted input.

And so we have the first function block of our arithmetic logic unit that is capable of addition or subtraction. It is also capable of indicating if there was a carry or borrow as a result of the arithmetic operation.

3.4 The D latch



The D latch is a basic memory cell that remembers the state of an input 'D' when told to by the 'CLK' input. This memory cell is used to retain data and is a key component of a processing system. It is used in memory devices and it is also used in the ALU as a vital part of the calculating process. It is used to hold the data that will be fed to the addition / subtraction unit. It can also be configured to perform multiplication and division.



By connecting the latches as shown, one can create what is known as a shift register. This shift register will shift data from right to left with every rising transition of the clock pulse.

3.5 Multiplication and division using the shift register

Out	D3	D2	D1	D0	
0	0	0	1	1	(3)
0	0	1	1	0 <-	(6)
0	1	1	0	0 <-	(12)
1	1	0	0	0 <-	(24)

As can be seen above, the initial value of 3 is multiplied by 2 every time the shift register performs a left shift under command of the clock pulse.

Now let us say that we wanted to perform a multiplication of say 3 x 6. We could add 3 six times but that would become very long winded for larger numbers. So, we start with the following condition.

	D6	D5	D4	D3	D2	D1	D0	Value
A=	0	0	0	0	0	1	1	(3)
B=	0	0	0	0	1	1	0	(Don't Add A to PR)
PR=	0	0	0	0	0	0	0	(0)

We look at the LSB of B. If it is a 1, we add it to a product total. If it is a 0, we shift A left and we shift B right. So, the two registers now look like this.

A=	0	0	0	0	1	1	0 <-	(6)
B=	0	0	0	0	0	1	1 ->	(Add A to PR)
PR=	0	0	0	0	1	1	0	(6)

The LSB of B is now a 1 so we add the value of A to the product register.

Now we shift A left again, and B right again

A=	0	0	0	1	1	0	0 <-	(12)
B=	0	0	0	0	0	0	1->	(Add A to PR)
PR=	0	0	1	0	0	1	0	(18)

And so, by rotating the two registers past each other in opposite directions it becomes possible to perform multiplication.

Division is done the same way.

A =	0	0	1	0	0	1	0	(18)
B =	0	0	0	0	1	1	0	(6)
RES=	0	0	0	0	0	0	0	(0)

We begin by shifting B to the left by as many times as it takes to place the most significant bit (MSB) of B in the MSB position of A and we remember how many times we did it.

A=	0	0	1	0	0	1	0	(18)
B =	1	1	0	0	0	0	0 <<<	(60)
RES=	0	0	0	0	0	0	0	(0)

Now we look to see if we can subtract B from A without a Borrow. And the answer is no, so we shift B right and we shift RES left.

A=	0	0	1	0	0	1	0	(18)
B=	0	1	1	0	0	0	0 >	(48)

We subtract again, and the answer is no again so we repeat the process till we can.

A=	0	0	1	0	0	1	0	(18)
B=	0	0	1	1	0	0	0 >	(24)

Again

A=	0	0	1	0	0	1	0	(18)
B=	0	0	0	1	1	0	0 >	(12)
RES=	0	0	0	0	0	0	1 <	(1)

And this time we can subtract B from A. So we set the least significant bit of the result register. We shift B left and we shift RES left by 1 position.

A =	0	0	0	0	1	1	0	(18- 12 = 6)
B=	0	0	0	0	1	1	0 >	(6)
RES=	0	0	0	0	0	1	1 <	(2)

Once again, we subtract B from A. If it can be performed without a borrow, we once again set the LSB of RES.

A=	0	0	0	0	0	0	0	(0)
B=	0	0	0	0	0	1	1>	(3)
RES=	0	0	0	0	1	1	0<	(6)

So, to complete the division, we shifted B left 4 times, then we shifted B right 4 times, subtracting B from A every time B was greater or equal to A, and marking the successful transaction in RES. At the end of the process, A=0 and the result is in the RES latches.

In this way, the digital system can perform addition, subtraction, multiplication and division.

3.6 Successive approximation methods.

Some processors have a dedicated hardware multiplier to perform fast multiplications. The process of division is somewhat more complicated and requires more circuitry or software. We can therefore employ a hardware multiplier to take a number of guesses as to what the answer of a division equation is.

When performing multiplication or division, one must perform a repeated operation for every bit of the data value. Division however requires more steps than multiplication. With a processor that is equipped with a fast hardware multiplier, we use the multipliers functionality to speed up the process.

When doing multiplication or division, we have to perform an addition or subtraction operation for every bit of the two factors that are being multiplied or divided. So, an 8 bit by 8 bit division or multiplication requires 8 arithmetic operations.

Using an 8 bit value as an example, we follow this process to obtain the result of 236 / 12.

We start with 12 x 128 = 1536. This causes an overflow so we ignore it.

A=	1	0	0	0	0	0	0	0 ->	(128 x)
B=	0	0	0	0	1	1	0	0	(12)
C=	1	1	1	0	1	1	0	0	(236)
RES =	0	0	0	0	0	0	0	0 <-	

We shift A to the right once, RES to left once and repeat the process

12 x 64 = 768 is greater than 236, so we have an overflow, so we ignore it.

A=	0	1	0	0	0	0	0	0 ->	(64 x)
B=	0	0	0	0	1	1	0	0	(12)
C=	1	1	1	0	1	1	0	0	(236)
RES =	0	0	0	0	0	0	0	0 <-	

We shift A to the right once, RES to left once and repeat the process

12 x 32 = 384 is greater than 236, so we have an overflow, so we ignore it.

A=	0	0	1	0	0	0	0	0 ->	(32 x)
B=	0	0	0	0	1	1	0	0	(12)
C=	1	1	1	0	1	1	0	0	(236)
RES =	0	0	0	0	0	0	0	0 <-	

We shift A to the right once, RES to left once and repeat the process

12 x 16 = 192 is smaller than 236, and we have no overflow, so we process it.
192 is smaller than 236 so we subtract it and mark the transaction by setting the LSB of the RES register

A=	0	0	0	1	0	0	0	0 ->	(16 x)
B=	0	0	0	0	1	1	0	0	(12 = 192)
C=	0	0	1	0	1	1	0	0	(236-192=44)
RES =	0	0	0	0	0	0	0	1 <-	(1)

We shift A to the right once, RES to left once and repeat the process

12 x 8 = 96 is greater than 44, and we have an overflow, so we ignore it.

A=	0	0	0	0	1	0	0	0 ->	(8 x)
B=	0	0	0	0	1	1	0	0	(12 = 96)
C=	0	0	1	0	1	1	0	0	(44)
RES =	0	0	0	0	0	0	1	0 <-	(2)

We shift A to the right once, RES to left once and repeat the process

12 x 4 = 48 is greater than 48, and we have an overflow, so we ignore it.

A=	0	0	0	0	0	1	0	0 ->	(4 x)
B=	0	0	0	0	1	1	0	0	(12 = 48)
C=	0	0	1	0	1	1	0	0	(44)
RES =	0	0	0	0	0	1	0	0 <-	(4)

We shift A to the right once, RES to left once and repeat the process

12 x 2 = 24 is smaller than 44, and we have no overflow, so we process it.
24 is smaller than 44 so we subtract it and mark the transaction by setting the LSB of the RES register

A=	0	0	0	0	0	0	1	0 ->	(2 x)
B=	0	0	0	0	1	1	0	0	(12 = 24)
C=	0	0	0	1	0	1	0	0	(44-24=20)
RES =	0	0	0	0	1	0	0	1 <-	(9)

We shift A to the right once, RES to left once and repeat the process

12 x 1 = 12 is smaller than 20, and we have no overflow, so we process it.

12 is smaller than 20 so we subtract it and mark the transaction by shifting the RES register left and setting the LSB of the RES register to mark a valid transaction.

A=	0	0	0	0	0	0	0	1 ->	(1 x)
B=	0	0	0	0	1	1	0	0	(12 = 12)
C=	0	0	0	0	1	0	0	0	(20 - 12 = 8)
RES =	0	0	0	1	0	0	1	1 <-	(19)

We shift A to the right once, RES to left once and repeat the process

We have now calculated that 236 / 12 = 19 remainder 8 or 19.666

This method of successive approximation asks if the result of 12 x 128 is greater or smaller than the number being divided, the divisor. It then successively asks the same question for 64, 32, 16, 8, 4, 2 and 1. When the answer is greater, the process is ignored for that cycle. If it is smaller, the product of the multiplication is subtracted from the divisor, a mark is made in the RES register, and the process is repeated downwards to 1.

This process of guessing or successive approximation is used in many mathematical and sampling processes to acquire results.

3.7 Square roots

Lets say for example you want to obtain a square root. Lets take for example 10608. We start with 128 and we test it. We move A to B. 128 x 128 = 16384. We see that is bigger than 10608 so we ignore it.

A=	1	0	0	0	0	0	0	0 ->	(128)
B =	1	0	0	0	0	0	0	0	(128)
RES=	0	0	0	0	0	0	0	0 <-	(0)

Shift the A left and RES right 1 position. We see that the square of 64 is 4096 and that is smaller than 10608 so we process it. We move A to B, add C to B, and move the result back to C. Mark the transaction in the RES register and shift A and C right and left.

A=	0	1	0	0	0	0	0	0 ->	(64) x 64 = 4096
B =	0	1	0	0	0	0	0	0	(64)
C=	0	1	0	0	0	0	0	0	(64)
RES=	0	0	0	0	0	0	0	1 <-	(1)

We have now guessed that the root of 10608 is smaller than 128 and larger than 64. We now repeat the process to a higher level of resolution. We checked with a resolution of 64, now we repeat the process for 32. We know that the answer is somewhere between 64 and 128. We must now decide if it is larger or smaller than 96. (64+32)

Shift the A left and RES right 1 position. We see that the square of 96 is 9216 and that is smaller than 10608 so we process it. We move A to B, add C to B, and move the result back to C. Mark the transaction in the RES register and shift A and C right and left.

A=	0	0	1	0	0	0	0	0 ->	(32)
B =	0	1	1	0	0	0	0	0	(96) x 96 = 9216
C=	0	1	1	0	0	0	0	0	(96)
RES=	0	0	0	0	0	0	1	1 <-	(3)

We have now guessed that the root of 10608 is smaller than 112 and larger than 96. We now repeat the process to a higher level of resolution and look at the half way point between the 2 numbers.

Shift the A left and RES right 1 position. We see that the square of 112 is 12544 and that is larger than 10608 so we ignore it. We shift A and C right and left.

A=	0	0	0	1	0	0	0	0 ->	(16)
B =	0	1	1	0	0	0	0	0	(112) x 112 = 12544
C=	0	1	1	0	0	0	0	0	(96)
RES=	0	0	0	0	0	1	1	0 <-	(6)

We have now guessed that the root of 10608 is smaller than 112 and larger than 96. We now repeat the process to a higher level of resolution and look at the half way point between the 2 numbers.

Shift the A left and RES right 1 position. We see that the square of 104 is 10816 and that is larger than 10608 so we ignore it. We shift A and C right and left.

A=	0	0	0	0	1	0	0	0 ->	(8)
B =	0	1	1	0	1	0	0	0	(104) x 104 = 10816
C=	0	1	1	0	0	0	0	0	(64)
RES=	0	0	0	0	1	1	0	0 <-	(12)

We have now guessed that the root of 10608 is smaller than 104 and larger than 96. We now repeat the process to a higher level of resolution and look at the half way point between the 2 numbers.

Shift the A left and RES right 1 position. We see that the square of 100 is 10000 and that is smaller than 10608 so we process it. We move A to B, add C to B, and move the result back to C. Mark the transaction in the RES register and shift A and C right and left.

A=	0	0	0	0	0	1	0	0 ->	(4)
B =	0	1	1	0	0	1	0	0	(100) x 100 = 10000
C=	0	1	1	0	0	1	0	0	(100)
RES=	0	0	0	1	1	0	0	1 <-	(25)

We have now guessed that the root of 10608 is smaller than 104 and larger than 100. We now repeat the process to a higher level of resolution and look at the half way point between the 2 numbers.

Shift the A left and RES right 1 position. We see that the square of 102 is 10404 and that is smaller than 10608 so we process it. We move A to B, add C to B, and move the result back to C. Mark the transaction in the RES register and shift A and C right and left.

A=	0	0	0	0	0	0	1	0 ->	(2)
B =	0	1	1	0	0	1	1	0	(102) x 102 = 10404
C=	0	1	1	0	0	0	1	0	(102)
RES=	0	0	1	1	0	0	1	1 <-	(51)

We have now guessed that the root of 10608 is smaller than 104 and larger than 100. We now repeat the process to a higher level of resolution and look at the half way point between the 2 numbers.

Shift the A left and RES right 1 position. We see that the square of 102 is 10404 and that is smaller than 10608 so we process it. We move A to B, add C to B, and move the result back to C. Mark the transaction in the RES register and shift A and C right and left.

A=	0	0	0	0	0	0	0	1 ->	(1)
B =	0	1	1	0	0	1	1	1	(103) x 103 = 10609
C=	0	1	1	0	0	1	1	1	(103)
RES=	0	1	1	0	0	1	1	1 <-	(103)

We have now guessed that the root of 10609 is smaller than 104 and larger than 102. We now repeat the process to a higher level of resolution and look at the half way point between the 2 numbers.

And we obtain an answer of 103, being the square root of 10609

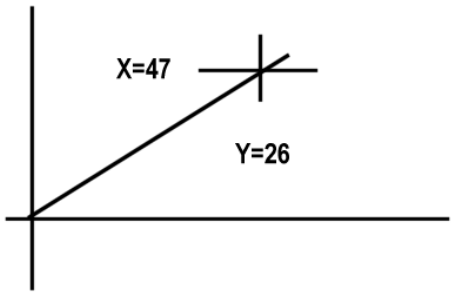
The shit register, made of a sequence of D latches, or memory cells, linked sequentially allows us to perform multiplication, division, square and square root functions.

3.8 Trigonometric Tabular functions.

Now we move up to the more complex trigonometric functions like sine, cosine and tangents. Again we are faced with the dilemma of speed. We can have a highly accurate process, but that takes time to process. On the other hand we desire speed so we use a slightly less accurate algorithm, but one that is blindingly fast. To do this we draw a lookup table. Into each item of the matrix, we place a value. The table represents the first quadrant where both the X and the Y values are positive values.

7	8.131	15.943	23.199	29.745	35.538	40.601	45
6	9.462	18.435	26.565	33.690	39.806	45	49.399
5	11.310	21.801	30.964	38.660	45	50.194	54.462
4	14.036	26.565	36.870	45	51.340	56.310	60.255
3	18.435	33.690	45	53.130	59.036	63.435	66.801
2	26.565	45	56.310	63.435	68.199	71.565	74.055
1	45	63.435	71.565	75.964	78.690	80.538	81.87
0	1	2	3	4	5	6	7

Now we take the vector whose angle we wish to determine.



First we determine the vector, for this determines the scaling size. The length of the vector, L = SQR(x^2 + y^2) and in this case, L = 53.
One uses a table size suited to the resolution one needs. Above we have use an 8 x 8 table to demonstrate the concept, but it is common to use even larger tables. From the above we can see that a vector of X=4 and Y=2 equals an angle of 63.435°. That would hold true for X=40 and Y = 20.
But what about X=47 and Y= 26?
We would take
X = 4 + 4 + 4 + 5 + 5 + 5 + 5 + 5 +5 + 5 = 47
Y = 2 + 2 + 2 + 2 + 3 + 3 + 3 + 3 + 3 + 3 = 26
The Angle for X=4,Y=2 is 64°, X=5,Y=2 is 68° and X=5,Y=3 is 59°.
So the angular sequence for the above 10 vectors are :
63.435° + 63.435° + 63.435° + 68.199° + 59.036° + 59.036° + 59.036° + 59.036° + 59.036° + 59.036° = 612.72/ 10 = 61.272°
The formula for determining an angle is : Theta = Arctan (Y/X) * radians

The exact angle according to the formula is 61.049°.

There is a slight discrepancy because we are only working with an 8 x 8 matrix. That is only a 3 bit resolution but you can see from the principle of the calculation, that the result is actually amazingly close for just a 3 bit resolution. Set up a 4 bit resolution table and you half the error. As it is, the error is less than half a bit or half a degree. 4 bits would be a ¼ degree error, 5 bits a 8^{th} , 6 bits a 16^{th} , 7 bits a 32^{nd} and 8 bits a 64^{th} and so on. A 64^{th} of a degree is 0.01 degrees. So, the higher the resolution you require, the more bits you use.

As you can see we can use relatively small tables to do quite accurate lookups.

We can do this for all the higher trig functions and even for custom math functions. One can for example have a table for the length of the diagonal of a right-angled triangle.

5	5.099	5.385	5.831	6.403	7.071
4	3.873	4.472	5.000	5.657	6.403
3	3.162	3.606	4.243	5.000	5.831
2	2.236	2.828	3.606	4.472	5.385
1	1.414	2.236	3.162	3.873	5.099
0	1	2	3`	4	5

4.472 + 4.472 + 4.472 + 5.385 + 5.381 + 5.281 + 5.831 + 5.831 + 5.831 + 5.831 = 53.787
The Square (46^2 + 27^2) is actually 53.7122

Our low resolution lookup tables generate a vector length of 53.783 units at 61.272° for a X/Y offset of X=47 and Y=26.

Once again, a very small lookup table is used to generate a surprisingly accurate result. The nice thing about lookup tables is of course that they are blindingly fast. To really speed things up, one could even put the answers to addition, subtraction, multiplication, division and square root operations into lookup tables.

X = 6, Y = 5, And the answer, 30, is at the memory location. No lengthy calculations involved. Just go and fetch the answer at the relevant location of the matrix. It is common practice to compile lookup tables for the more time consuming functions in programs that need to perform at very high speed.

Lets say you had a micro controller that read a 2 axis hall effect sensor in order to provide a digital compass function. How accurate do you need it to be. 1 degree resolution with a less than .5 degree error is more than adequate for 99% of the applications. That would be an ideal application for a high speed low resolution lookup table.

This then has been an over view of the processes that are followed by computers to perform mathematical operations. The ALU (Arithmetic Logic Unit) in the computers CPU (Central Processing Unit) is able to perform almost any mathematical function through bit manipulation in dedicated hardware using adders, multipliers, shift registers,

or by following longhand mathematical processes and by using high speed lookup tables for the higher mathematical functions.

3.9 Digital signal processing & filtering examples

Having looked at how the processor performs mathematical functions, we now look at how a signal is processed by the CPU.

The process of Digital Signal Processing requires that we sample an input signal and process it digitally. The result of that process is then used to generate a relevant output.

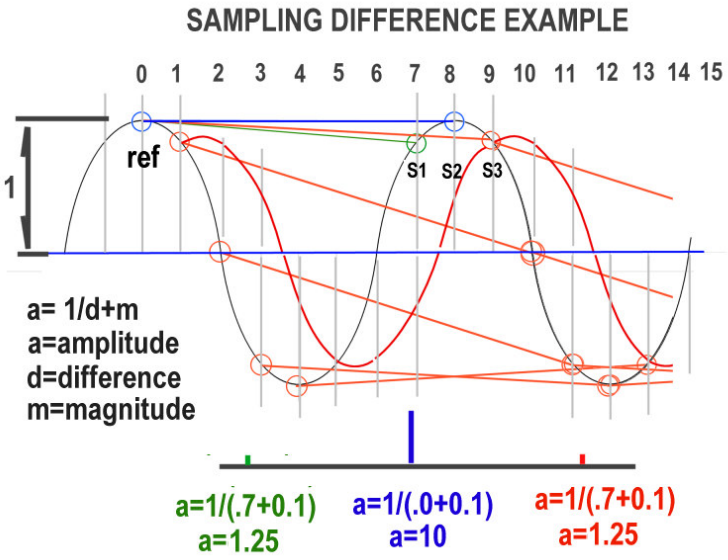
There are a large number of DSP algorithms for a wide range of applications.

We will look at 2 examples related to amateur radio.

Digital filtering and Spectral analysis

The first step is to take a sample of the incoming analog waveform and convert it into a digital value, represented in binary format inside the processor. There is a certain amount of time that must pass from the time that a sample is taken till the time that the next sample is taken. This is referred to as the sample time delay.

Lets look at a simple sine wave digitizing process in the sketch below.

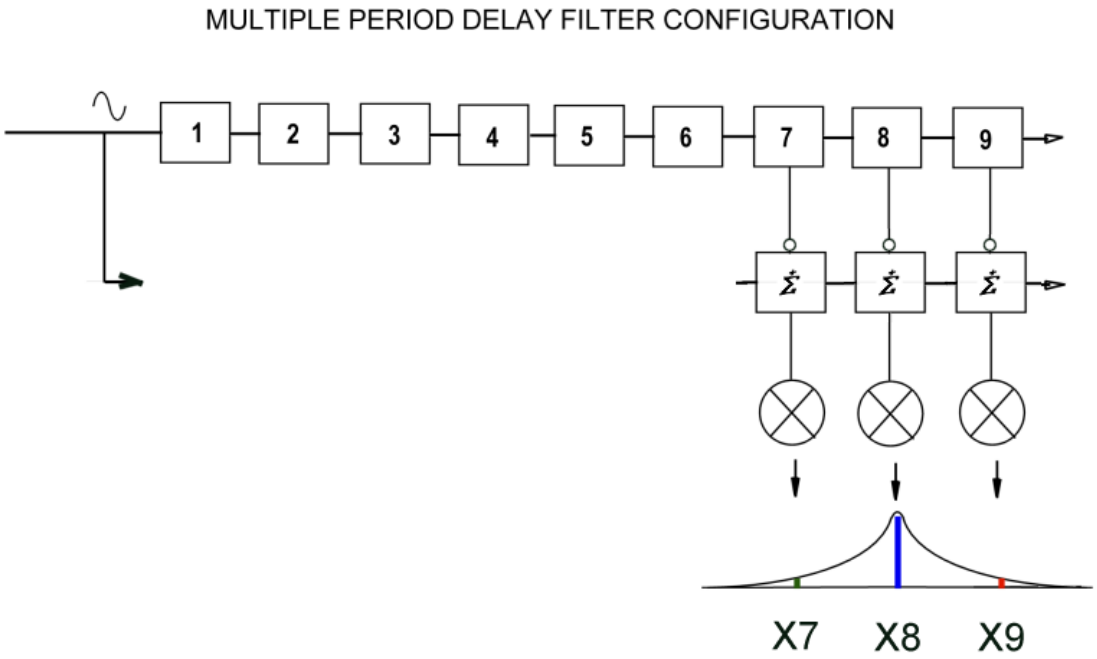


A sine wave is digitized over a number of samples. Sample 0 is referred to as the reference point, or V(0). This value is pushed into memory, and another digitization produces the next value V(1). This process continues till we have a queue of values lined up in the memory of the processor.

What we now do is subtract V(1) fro V(0), V(2) from V(0), V(3) from V(0) and so on. In the above case where the sine wave length is 8 samples long, you will see that the difference between V(0) and V(8) is always zero. The difference between V(0) and V(7) is 0.7 and the difference between V(0) and V(9) is 0.7

If we apply the formula a=1/(d+s) where a= amplitude, d= difference and s = scaling factor, we can get an amplitude for each sampling delay period. In this case, S(7) = 1.25, S(8) = 10 and S(9) = 1.25.

This process is presented diagrammatically in the following sketch.



The output amplitude signal of each channel is represented as X(7), X(8), X(9) etc, etc.

The curve along the bottom shows the resonant peak appearing at X(8). In the above formula, there is a subtraction process where the two samples are subtracted from each other, and a multiplication process to obtain a scaling factor. (The process of dividing by 2 is the same as multiplying by 0.5, so even though the formula shows a division, we indicate the process with a multiplying function block.)

What we do have however is an 8 channel spectral output for a single frequency input. This is a single frequency model. What we now need to do is a multi frequency analysis model.

The following model indicates a Fast Fourier Transform (FFT) process on the 8 signal channels derived above.

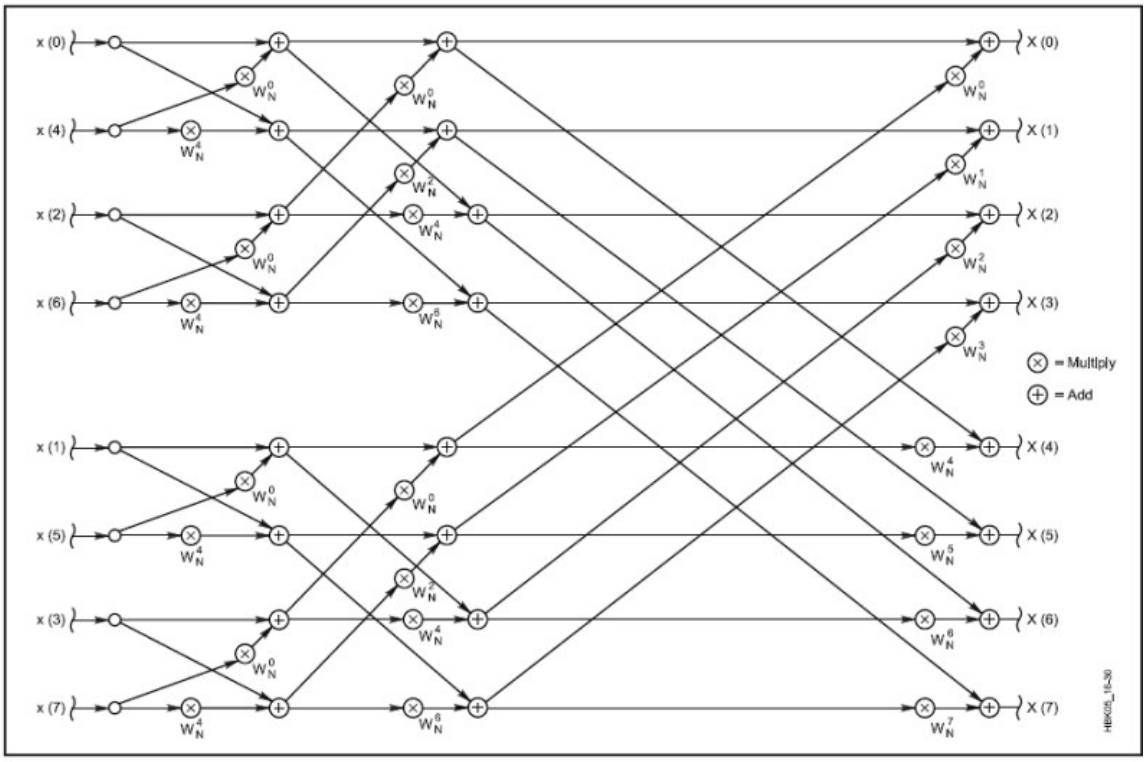


Fig 16.30—Flow chart of an 8-sample FFT.

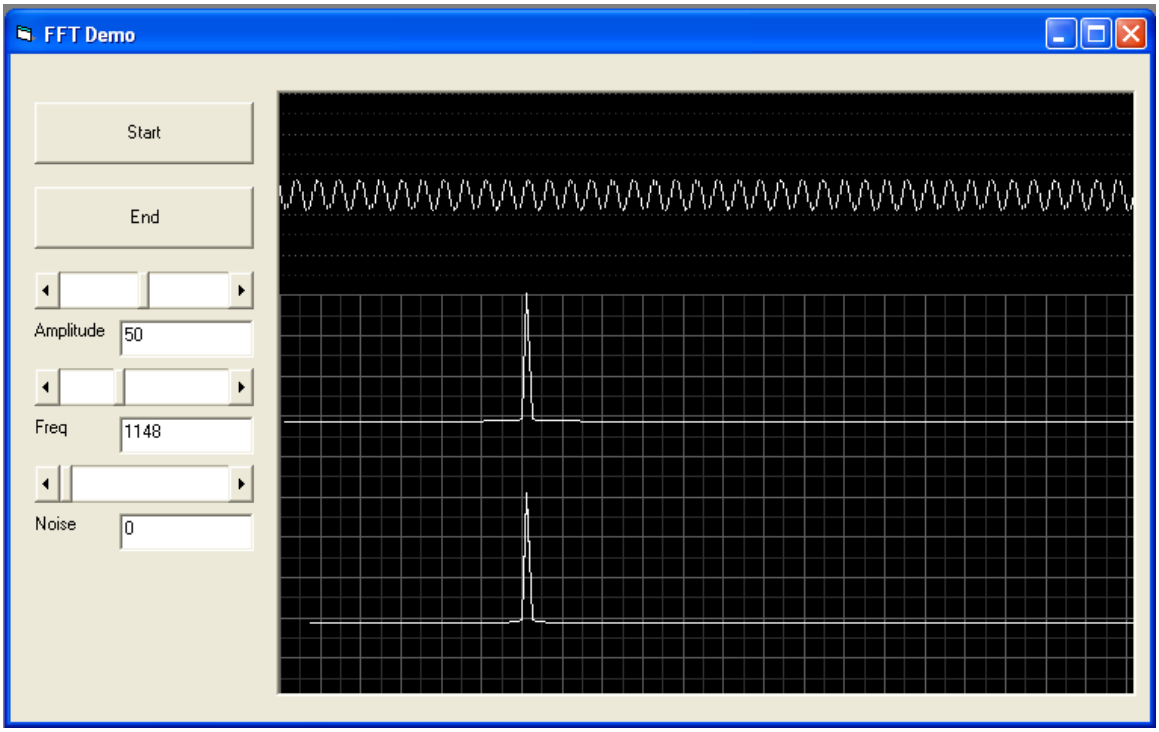
The channels X(0) – X(7) are our inputs. The lines from left to right represent a time delay. The addition and multiplication processes work the same as above. Signals are applied out of phase, at different time intervals to generate an 8 channel multi frequency spectral profile.

The resolution of using 8 channels is however very coarse. I have therefore applied the same algorithm to a 512 channel FFT process.

1->2->4->8->16->32->64->128->256->512

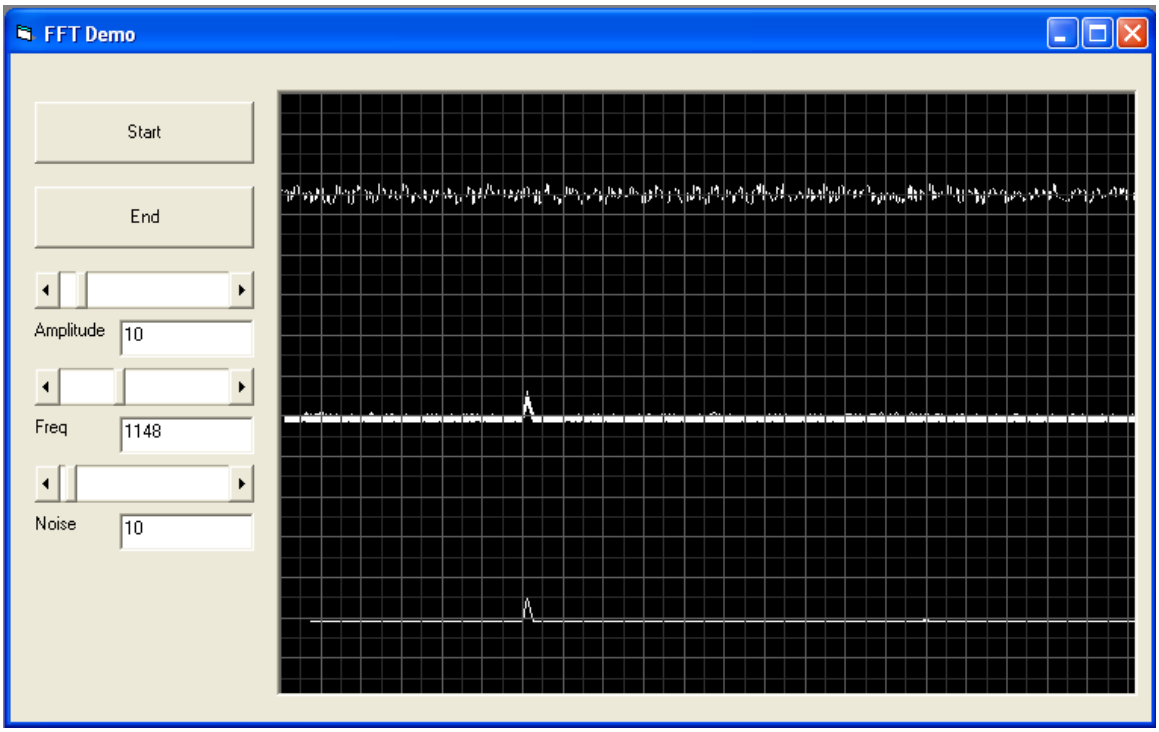
Here are the results for a FFT spectrum analyzer using the above process.

A single frequency FFT analysis using 512 samples into 512 channels.

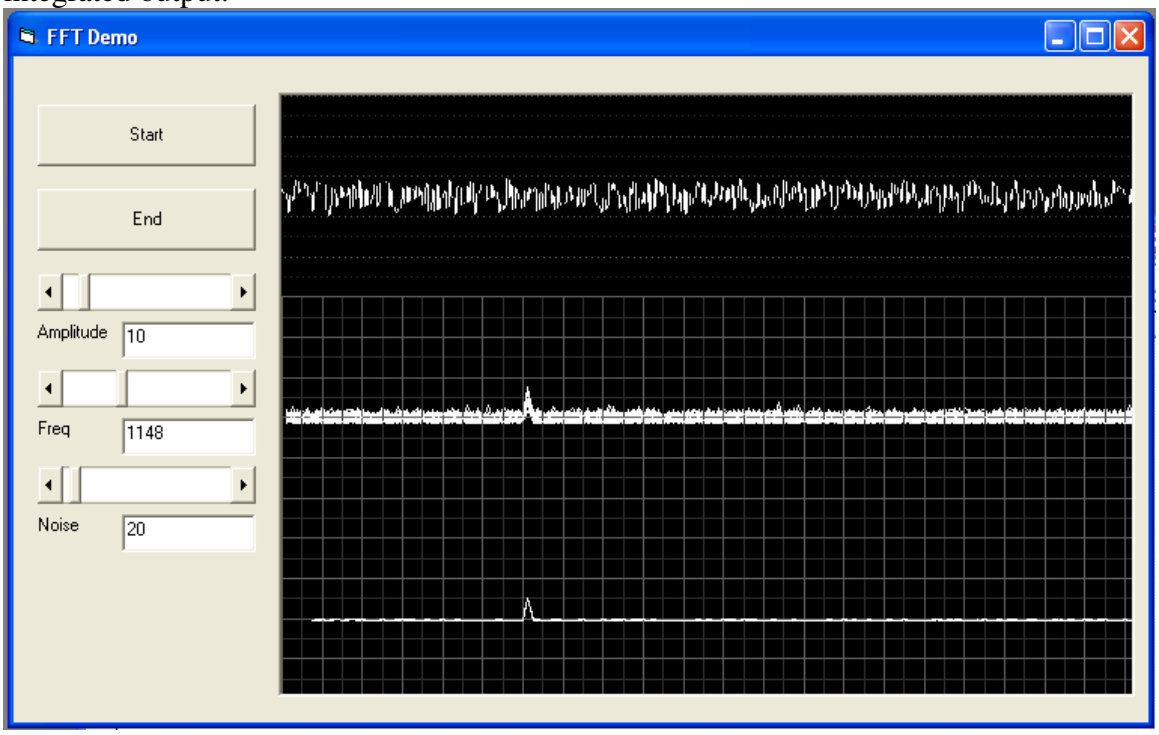


A single frequency of 1148 Hz is applied into the model at 50mV. The top trace shows the applied sine wave. The second trace shows the spectral peak of that frequency. The third trace shows the second trace data integrated to a depth of 128 samples to clean up the noise factor in the signal. We are therefore looking at a three dimensional matrix of 512 elements by 512 elements by 128 elements.

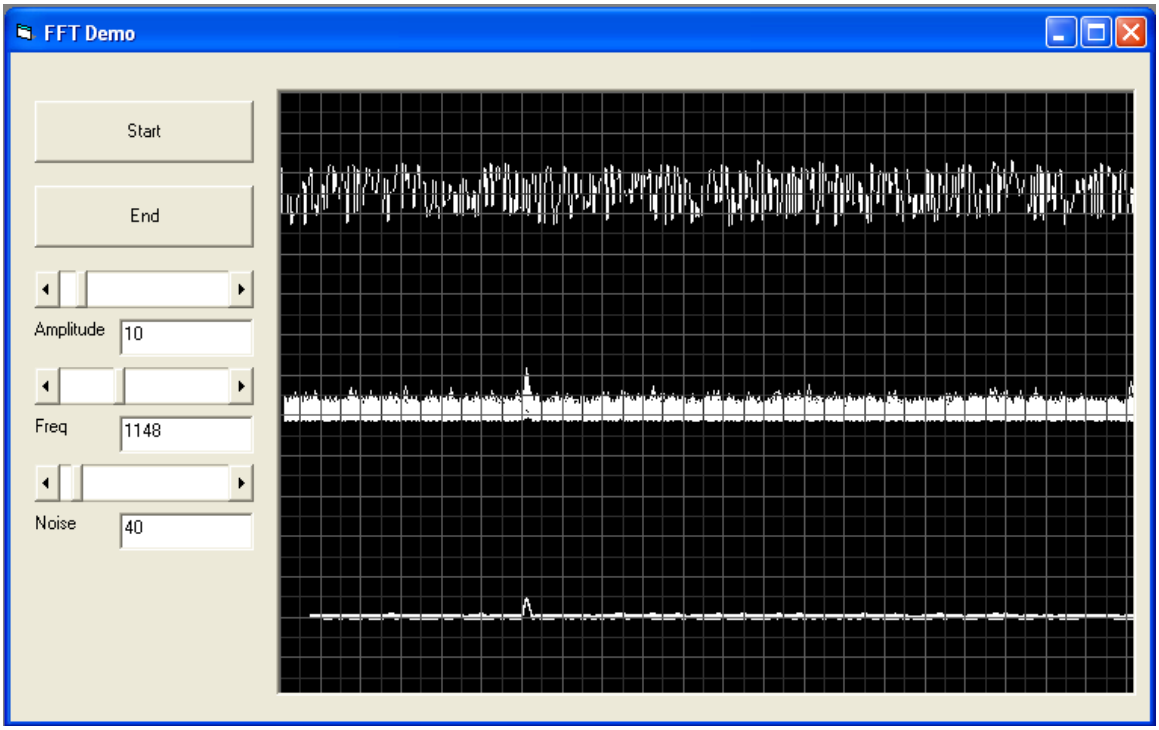
Our simulator allows us to feed the algorithm with a sine wave of variable amplitude and frequency and frequency and a noise factor of varying levels. I use this program to test various DSP algorithms before committing them to assembler code in a micro controller.



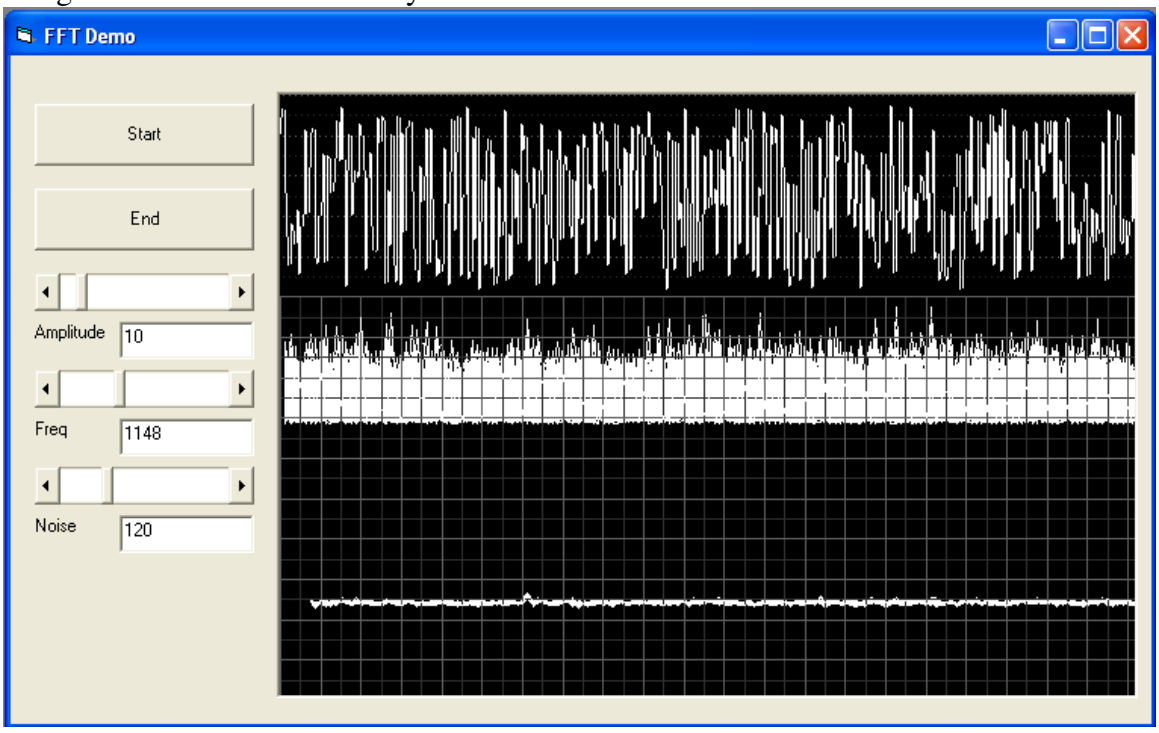
We now reduce the signal amplitude to 10mV signal and 10mV noise. As can be seen, we are still able to discern a clear peak at 1148 Hz on the bottom trace which represents the integrated output.



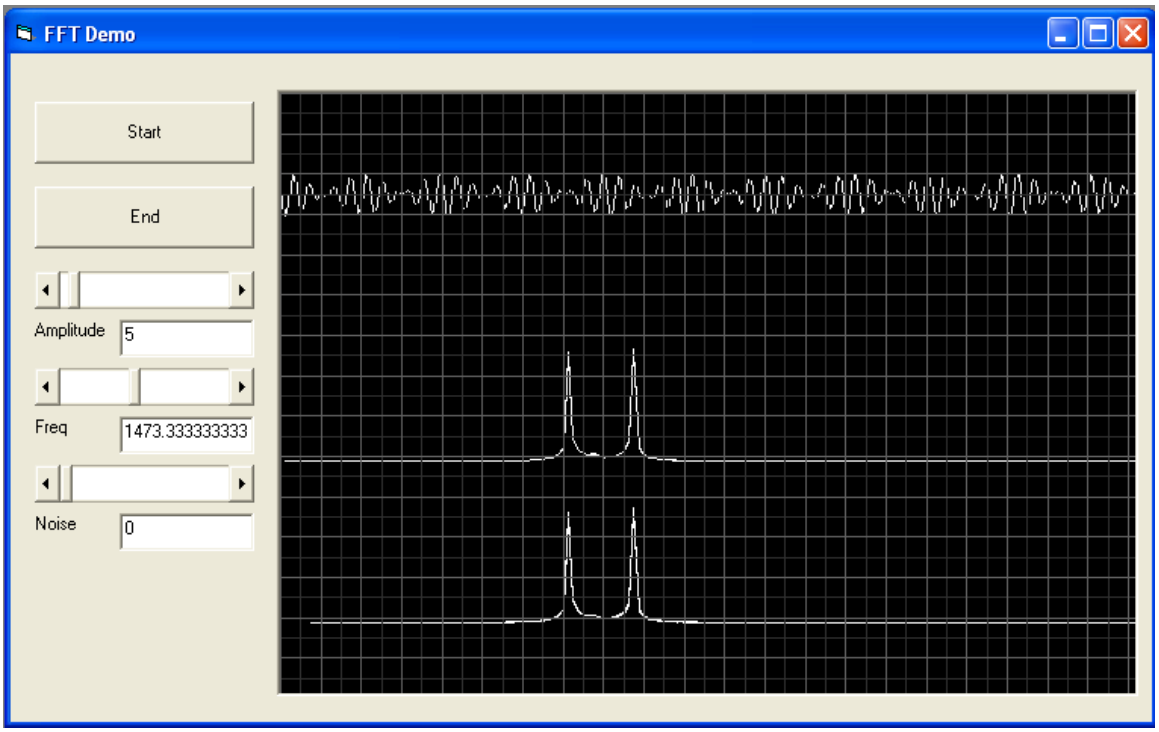
Increasing the noise level to 20mV still produces a discernable spike.



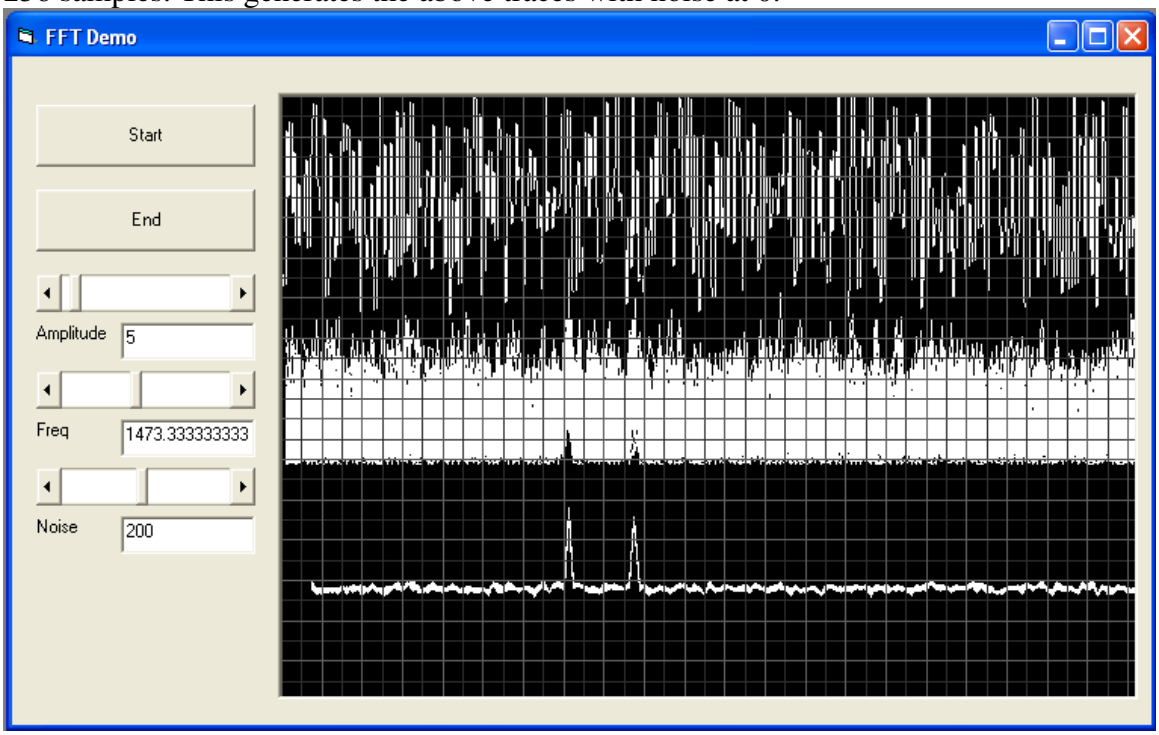
A signal of 10 mV is still clearly visible on 40mV of noise.



At 120mV noise to 10 mV signal, we reach the limits of our algorithm. We can no longer discern any spike on the spectrum trace and the integrator output is barely able to pull a signal up out of the noise floor.



We have now re-scaled the display to represent 2 frequencies superimposed on each other, we have reduced the signal amplitude to 5mV and increased the integrator depth to 256 samples. This generates the above traces with noise at 0.

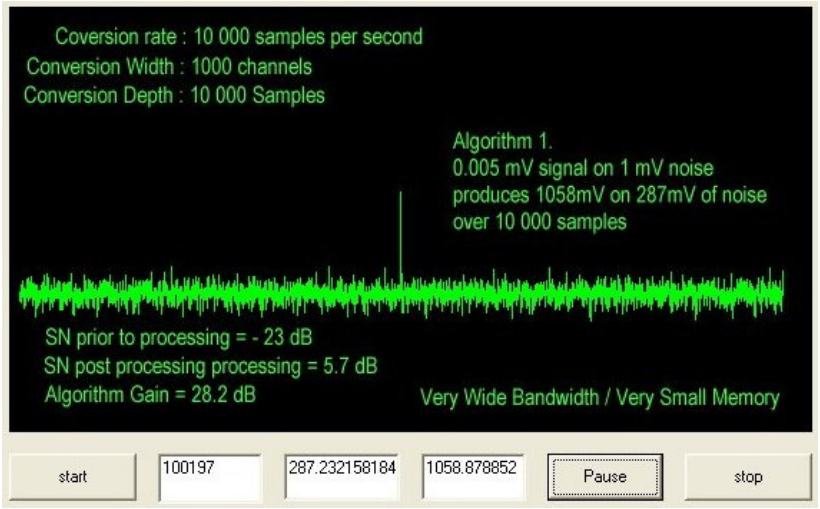


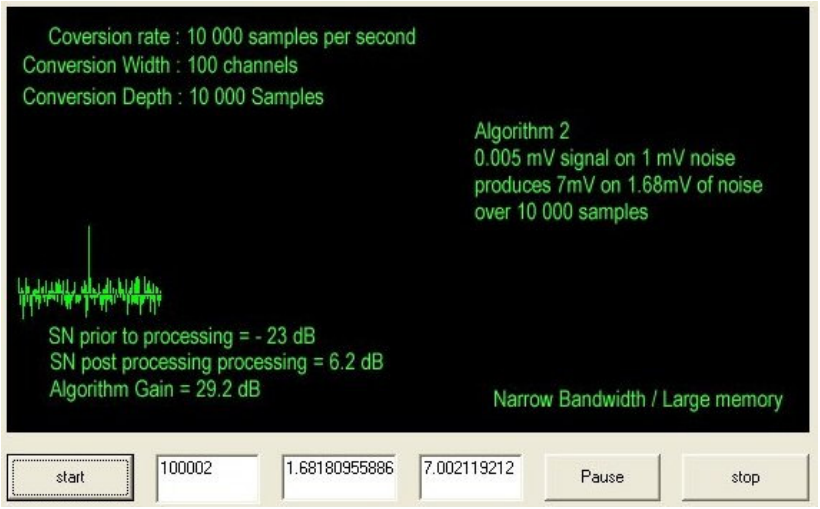
Increasing the noise to 200mV, (23dB) produces the following trace. As can be seen above, increasing the integrator depth allows one to pull a very weak signal up from below the noise floor. The problem is that the deeper the integrator, the longer the process.

The greater the depth of the integrator, the slower the algorithm becomes, but the weaker the signals that can be detected. In a 512 x 512 x 256 algorithm, we are performing the calculations on 67108864 elements of the matrix.

This is the basic principle of a noise canceling spectrum analyzer. Using this system, one can detect multiple signals. One can then look at the relationships of these signals and extract data for digital modes.

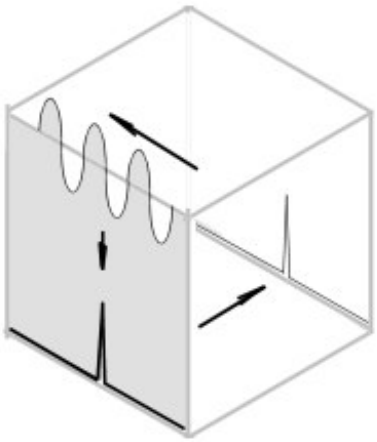
With this system we can extract data from signals that are up to 30dB below the noise floor. Ie, 1mV of signal to 1000 mV of noise using a 10 000 sample integrator as shown in the screen captures below that demonstrate the output of two different algorithms.



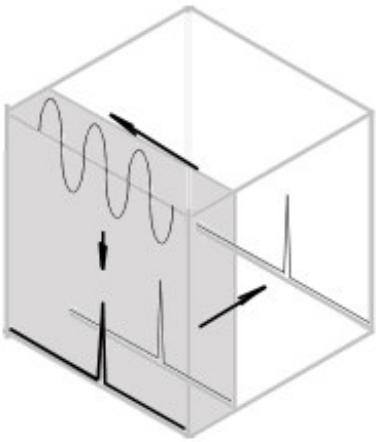


Optimization.

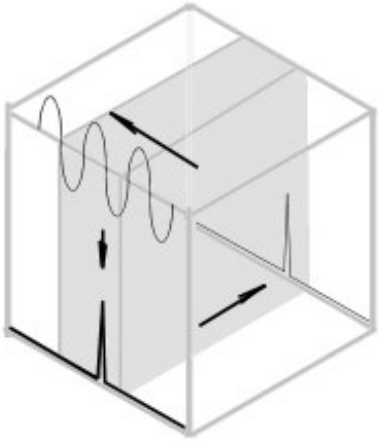
Once we have an algorithm that performs a task to a satisfactory level, it becomes an inevitable consequence that we want it to be faster. Lets look at the 3D matrix.



To begin with, if we have a good signal with low noise, we do not need to do any integration to remove the noise. We can simply transpose the output of the FFT stage to the output of the integrator stage.



As the noise level increases so we can dynamically increase the integration depth. The noisier the signal becomes, the greater the level of integration.



Once we have isolated the desired signals in the spectrum, we do not need to integrate the entire spectrum. We can just integrate the narrow portion of the spectrum that contains our signal of interest.

These optimization techniques vastly improve the overall performance of the system. Instead of processing the entire matrix every cycle, we only process the portion of the matrix of interest to us.

We have looked at the mathematical processes used by a micro controller. We have followed the functions, from the simplest addition to the highest level.

Now we shall take a more in depth look at the components of the micro controller and we will see how it uses its various function blocks to perform the various processes described above.

4. Central processing unit

The central processing unit is where, as the name denotes, the processing of information takes place. To achieve this an number of function blocks are laid out in a certain configuration. This configuration is referred to as the 'architecture'

There are four types of architecture that can be employed to provide processing functions. These are :

a. Harvard Architecture
b. Van Neumann Architecture
c. Neural Network Architecture
d. Self Evolved Network Architecture

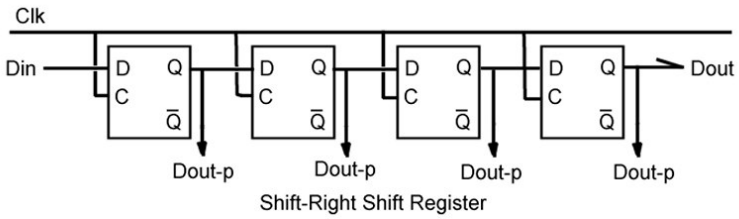
We shall examine these in greater detail once we have looked at the function blocks that make up a CPU.

4.1 Registers

Registers are the data retention blocks of a processing environment. The register is a collection of 'D' type latches. A 'D' type latch is a Data latch. The condition at the Data input is latched into the storage cell by the transition of a clock pulse.

Registers are groups of 'D' latches connected up in different configurations to perform different functions.

4.1.1 The Shift-Right Register (Division)



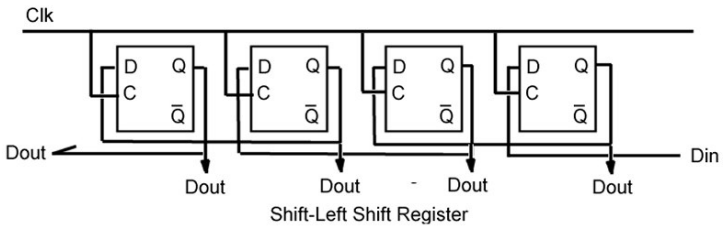
Shift-Right Shift Register

The Sift right register will shift its contents to the right by 1 position with every transition of the clock pulse.

```
1011 -> 0101
0101 -> 0010
0010 -> 0001
0001 -> 0000
```

This process is the equivalent of dividing by 2 and losing the remainder with each successive shift. The initial value of 11 becomes 5, then 2, then 1, then zero. One has the option to connect Dout to Din, which will then allow the data to circulate through the shift register, left to right.

4.1.2 The Shift-Left Register (Multiplication)



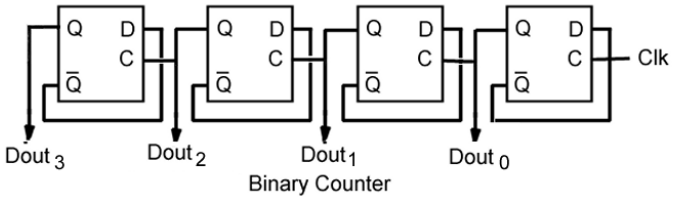
Shift-Left Shift Register

The Shift left register will shift its contents to the left by 1 position with every transition of the clock pulse.

```
0011 <- 0110
0110 <- 1100
1100 <- 1000
1000 <- 0000
```

This is the equivalent of multiplying by 2 up to the maximum width of the shift register. After that point is reached, the overflow is lost with each successive shift. The initial value of 3 becomes 6, which then becomes 12, and then 16. It would have become 24 if the register was wider than 4 bits.

4.1.3 The Binary counter.



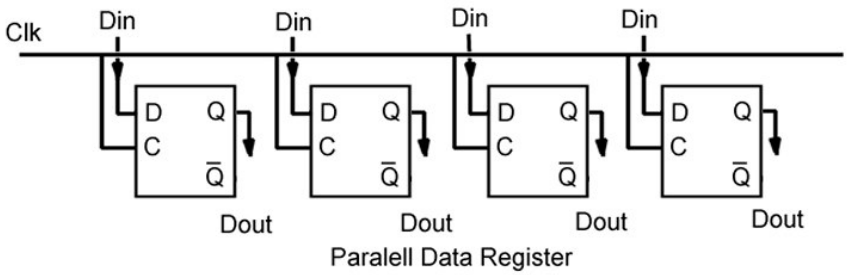
Binary Counter

The inverted Q output is fed back to the D input and the Q output changes state with every positive going transition of the clock input. The result is that Every Q output changes state once for every 2 state changes on the input. Each successive D latch therefore becomes a ‘divide-by-two’ counter.

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
```

A count-up function is provided at the Q outputs while a count-down function is provided at the /Q outputs. The up counter is therefore also a down counter,

4.1.4 The Parallel register



The parallel data register will load the data presented at the inputs Din into the D latches. This value will then become available at the Q outputs. This is a general-purpose data storage register that is used to retain data.

4.2 The register set.

There are a large number of registers in the processing core of the CPU. These are used for a wide range of functions.

4.2.1 The accumulator or working register. This is the register that receives the result of an operation within the ALU (Arithmetic Logic Unit.) If an addition, subtraction, shift, transfer or logical operation is carried out, the destination of the operation usually ends up in the accumulator. In certain cases it is possible to bypass the working register

4.2.2 The general-purpose variable or data registers. All processors have a set of registers in which they store variables or data that is being processed. Some processors have a large number of such registers and some have a lesser number.

4.2.3 Control registers are used to control devices. Writing to a control register will control the operation of a device in the processing environment.

4.2.4 Status registers are used to indicated the status of a device. When a mathematical operation results in an overflow, the status bit that indicates an overflow will be set. By performing an operation and checking the overflow bit in the ALU status register, we can see if an arithmetic operation has resulted in an overflow.

4.2.5 Input registers contain data that has come in from external devices. Things like timers, counters, serial ports, analog to digital converters.

4.2.6 Output registers contain data that is destined for an external device.

4.2.7 Counters. There are various types of counters. The most important is the program counter. This counter contains the address of the instruction being processed by the CPU. Upon the completion of each instruction, the CPU will increment the program counter, recall the next sequential instruction and execute it.

ALU

I/O

Interrupts

Opcodes

Serial data communications

Parallel data communications

Counters and timers

Analog to digital conversion

Digital to analog conversion

Memory