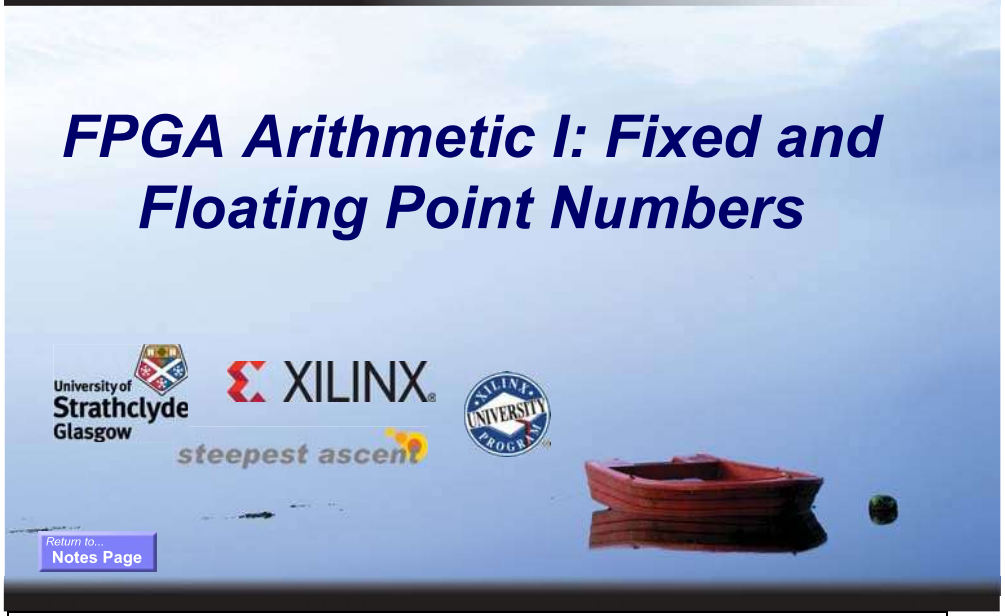
FPGA Arithmetic I: Fixed and Floating Point Numbers



Introduction

4.1

- DSP is all about arithmetic, so the ***representation of numbers*** is a vital aspect of implementing DSP systems.
- This chapter covers ***fixed point*** (which is used for the majority of DSP applications), and ***floating point*** (occasionally used in cases where high dynamic range is required).
- The following key issues are presented in this chapter:
  - ***Number representations:***
    Binary word formats for signed and unsigned integers, 2's complement, fixed point and floating point;
  - ***Fixed point binary arithmetic,*** *including:*
    Overflow and underflow, saturation, truncation and rounding, normalisation;
  - ***Floating point arithmetic***:
    Representations and principles, the IEEE 754 standard.

THIS SLIDE IS BLANK

Notes:

This section of the course will introduce the following concepts:

Integer number representations: unsigned and two's (2's) complement.

Fixed point numbers and the binary point

Properties of fixed point and integer numbers: range and precision.

Simple arithmetic: principles of binary addition and subtraction

Aspects of finite range and precision: wraparound overflow and underflow, truncation and rounding.

Normalisation: the motivation and mechanics.

Floating point: form of floating point numbers, properties, and operations; IEEE 754 standard for floating point; custom formats; block floating point; FPGA tool and architecture support for floating point.

Number Representations

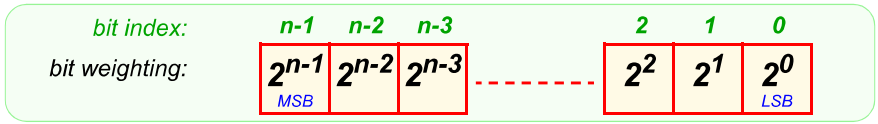
4.2

- DSP, by its very nature, requires quantities to be represented digitally - using a number representation with ***finite precision***.
- This representation must be specified to handle the ***"real-world"*** inputs and outputs of the DSP system.
  - Sufficient ***resolution***
  - Large enough ***dynamic range***
- The number representation specified must also be ***efficient*** in terms of its implementation in hardware.
  - The hardware implementation ***cost*** of an arithmetic operator ***increases with wordlength***.
  - The relationship is not always linear!
- There is a ***trade-off*** between ***cost***, and ***numeric precision and range***.

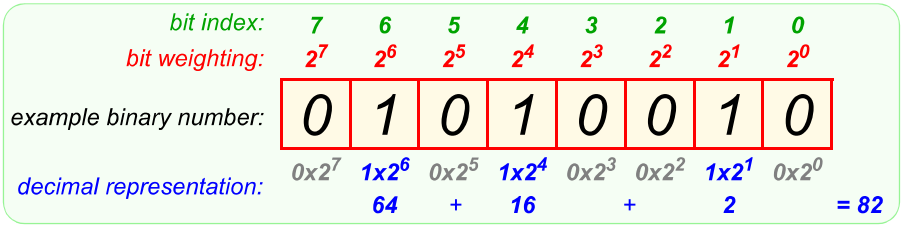
Unsigned Integer Numbers

4.3

- ***Unsigned*** integers are used to represent non-negative numbers.
- The weightings of individual bits within a generic ***n***-bit word are:



- The decimal number ***82*** is "***01010010***" in unsigned format as shown:



- The numerical range of an ***n***-bit unsigned number is: ***0*** to ***2^n - 1.*** For example, an ***8***-bit word can represent all integers from ***0*** to ***255***.

Notes:

The use of binary numbers is a fundamental of any digital systems course, and is well understood by most engineers. However when dealing with large complex DSP systems, there can be literally billions of multiplies and adds per second. Therefore any possible cost reduction by reducing the number of bits for representation is likely to be of significant value.

For example, assume we have a DSP filtering application using 16 bit resolution arithmetic. The cost of a parallel multiplier (in terms of silicon area - speed product) can be approximated as the number of full adder cells. Therefore for a 16 bit by 16 bit parallel multiply the cost is of the order of 16 x 16 = 256 "cells". The wordlength of 16 bits has been chosen (presumably) because the designer at sometime demonstrated that 17 bits was too many bits, and 15 was not enough - or did they? Probably not! Its likely that we are using 16 bits because... well, that's what we usually use in DSP processors and we are creatures of habit! In the world of FPGA DSP arithmetic you can choose the resolution. Therefore, if it was demonstrated that in fact 9 bits was sufficient resolution, then the cost of a multiplier is 9 x 9 cells = 81 cells. This is approximately 30% of the cost of using 16 bits arithmetic.

Therefore its important to get the wordlength right: too many bits wastes resources, and too few bits loses resolution. So how do we get it right? Well, you need to know your algorithms and DSP.

Notes:

Taking the example of an 8-bit unsigned number, the range of representable values is 0 to 255:

Integer Value	Binary Representation
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
⋮	⋮
64	01000000
65	01000001
⋮	⋮
131	10000011
⋮	⋮
255	11111111

Note that the minimum value is 0, and the maximum value (255) is the sum of the powers of two between 0 and 8, where 8 is the number of bits in the binary word:

i.e. $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 = 255 = 2^8 - 1$

2's Complement (Signed Integers)

4.4

- 2's Complement caters for ***positive and negative*** numbers in the range -2^{n-1} to $+2^{n-1}-1$, and has ***only one representation of 0 (zero)***.
- In 2's complement, the ***MSB*** has a ***negative weighting***:

bit index:	n-1	n-2	n-3	.....	2	1	0
bit weighting:	-2^{n-1} MSB	2^{n-2}	2^{n-3}	.....	2^2	2^1	2^0 LSB

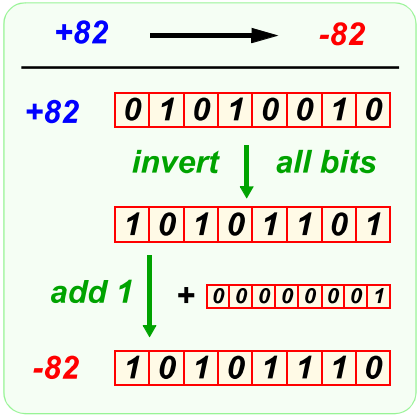
- The ***most negative*** and ***most positive*** numbers are represented by:

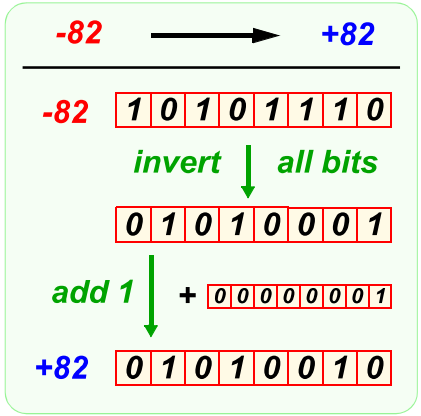
bit index:	n-1	n-2	n-3	.....	2	1	0
most negative:	1 MSB	0	0	.....	0	0	0 LSB
bit index:	n-1	n-2	n-3	.....	2	1	0
most positive:	0 MSB	1	1	.....	1	1	1 LSB

2's Complement Conversion

4.5

- For 2's Complement, ***converting between negative and positive numbers*** involves ***inverting all bits, and adding 1***.
- For example, we have just considered 2's complement representations of the decimal numbers ***-82*** and ***+82***. They are converted as shown:





Notes:

As examples, we can convert the following two 8-bit 2's complement words to decimal.

As for the unsigned representation, the decimal number ***82*** is "***01010010***" in 2's complement signed format:

bit index:	7	6	5	4	3	2	1	0	
bit weighting:	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
example binary number:	0	1	0	1	0	0	1	0	
decimal representation:	$0x-2^7$	$1x2^6$	$0x2^5$	$1x2^4$	$0x2^3$	$0x2^2$	$1x2^1$	$0x2^0$	
		64	+	16		+	2		= 82

Meanwhile the decimal number ***-82*** is "***10101110***" in 2's complement signed format:

bit index:	7	6	5	4	3	2	1	0	
bit weighting:	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
example binary number:	1	0	1	0	1	1	1	0	
decimal representation:	$1x-2^7$	$0x2^6$	$1x2^5$	$0x2^4$	$1x2^3$	$1x2^2$	$1x2^1$	$0x2^0$	
	-128	+	32	+	8	+ 4	+ 2		= -82

Notes:

Positive Numbers	
Integer	Binary
0	00000000
1	00000001
2	00000010
3	00000011
⋮	⋮
125	01111101
126	01111110
127	01111111

Invert all bits and ADD 1

Negative Numbers	
Integer	Binary
0	100000000
-1	11111111
-2	11111110
-3	11111101
⋮	⋮
-125	10000011
-126	10000010
-127	10000001
-128	10000000

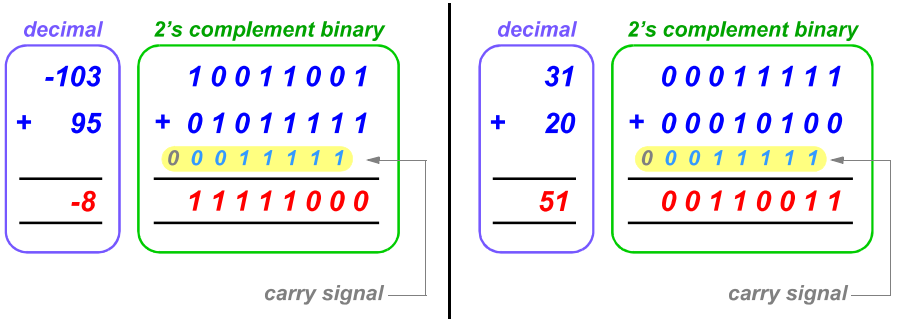
Note that when negating positive values, a ninth bit is required to represent negative zero. However, if we simply ignore this ninth bit, the representation for the negative zero becomes identical to the representation for positive zero.

Notice from the above that -128 can be represented but +128 cannot.

2's Complement Addition and Subtraction

4.6

- Examples of ***2's complement addition*** are shown below. Note that the ***final carry out*** is discarded, and ***does not*** form part of the result.



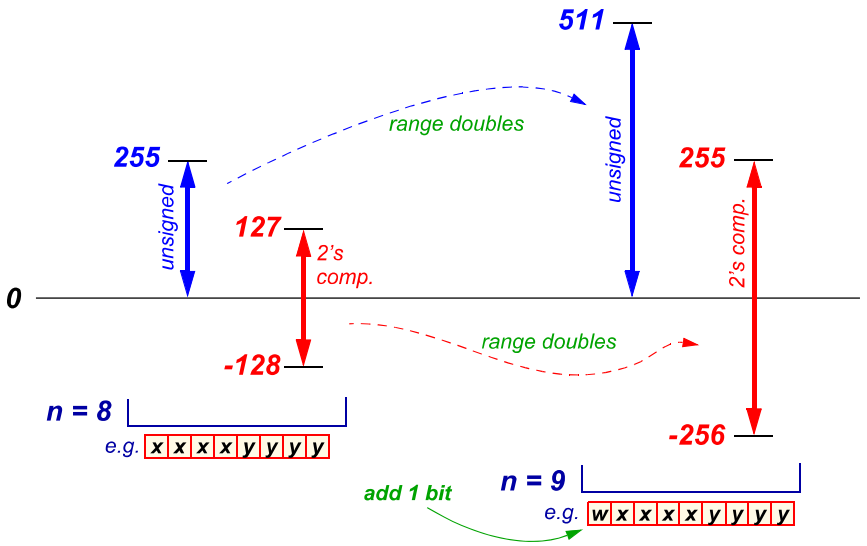
- ***Subtraction*** is very easily accomplished - it is an equivalent operation to addition, but with ***negative*** of the second operand, e.g.

$$S = A + (-B)$$

Numerical Range

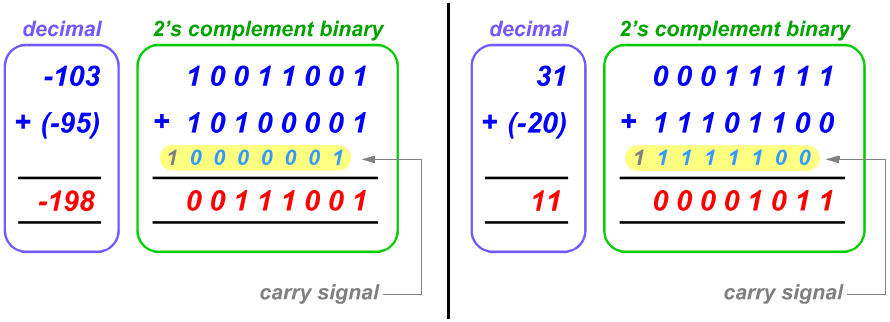
4.7

- To consider only ***unsigned*** and ***2's complement*** numbers...



Notes:

Inverting a number in 2's complement requires that you negate each bit, and add 1. For instance, in the main slide the *B* operand was 95 in the first example, and 20 in the second, so the computation then becomes the following (notice that the value of *B* has changed in both cases):



Later we will consider how these operations are implemented by a hardware circuit.

However, notice that in the first example, the binary result appears to be wrong - two negative numbers have been added together (... and hence the result should also be negative), but actually it appears as binary 00111001, or +57 in decimal. What has happened here?

In fact the arithmetic has ***overflowed*** - the magnitude of the result is too large to be represented in the allotted number of bits. Without intervention, ***wraparound*** occurs, i.e. a result is computed, but it is typically at the opposite end of the numerical range, and very inaccurate!

Notes:

The main slide confirms that increasing the wordlength of an unsigned or two's complement number increases its numerical range by a factor of two. Further examples:

n	unsigned		2's complement		Total range
	min	max	min	max	
7	0	127	-64	+63	128
8	0	255	-128	+127	256
9	0	511	-256	+255	512
10	0	1023	-512	+511	1024

If we define the dynamic range in decibels to be

$$\text{dynamic range (dB)} = 20 \times \log_{10}\left(\frac{\text{range}}{\text{interval}}\right) = 20 \times \log_{10}(2^n)$$

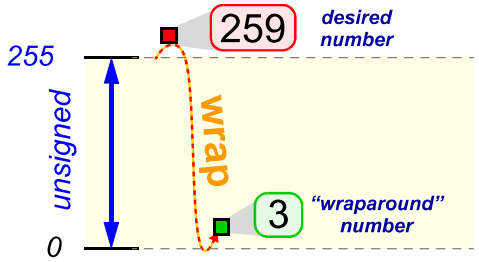
then a 16-bit number, for example, provides approximately 96dB of dynamic range.

For both the unsigned and 2's complement numbering systems, the dynamic range increases by just over 6dB for each bit added to the wordlength.

Overflow: Wraparound

4.8

- Overflow occurs when the ***magnitude*** of a number is ***too big*** to be represented in the ***available word format***. This usually happens as a result of a conversion (e.g. ADC), or an arithmetic operation.
- Due to the mechanics of binary arithmetic, ***wraparound*** naturally occurs on overflow. For example, as the number extends beyond the top of the range, it ***"wraps around"*** to the bottom end (and vice versa).

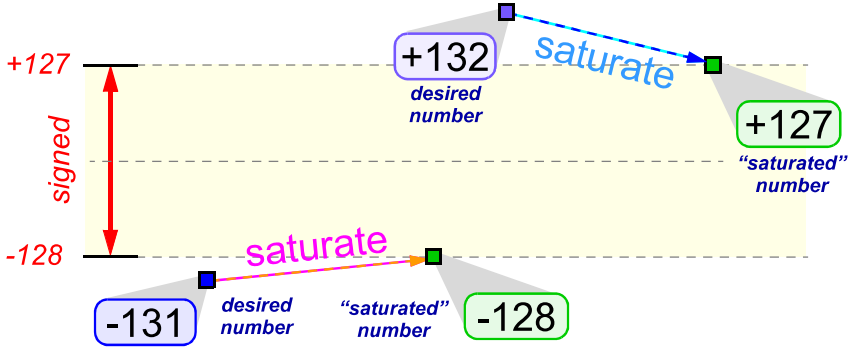


- Wraparound applies to both ***unsigned*** and ***2's complement*** numbers.

Overflow: Saturation

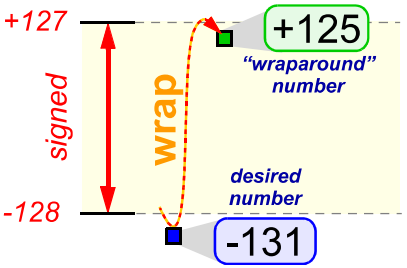
4.9

- It should be apparent that when numbers wraparound on overflow, the result is ***extremely inaccurate***!
- An alternative is to ***"saturate" on overflow***, i.e. detect that overflow has occurred, and switch in the minimum or maximum value.
- For example, replace all values ***higher than +127 with +127***, and all values ***lower than -128 with -128***.



Notes:

The wraparound effect occurs in the same way for 2's complement signed numbers. As shown in the example on the right, if a value falls outside the range of the word format, it wraps around to the opposite end of the range.



Note that, if the "desired" number is 4 greater than the maximum extent of the range (as in the unsigned example on the main slide), the "wrapped around" number is offset from the bottom of the range by 4. Similarly, in the 2's complement example to the right, the desired number is 3 less than the lower extent of the range, while the wrapped around number is offset from the top of the range by 3.

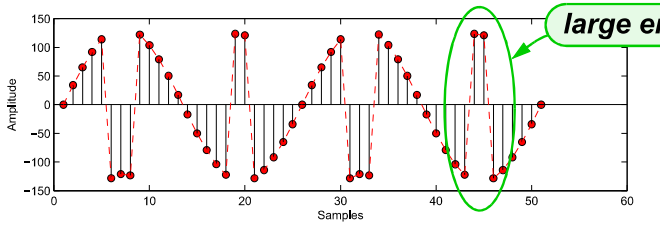
To explain further why this happens, consider the numbers below. You should notice that 9 bits would be required to represent the "desired" numbers, whereas if we remove this 9th bit, the "wrapped around" version of the number is obtained. Of course, if working with 8-bit words we never really have this 9th bit to begin with!

Unsigned example		***2's complement example***	
9 bits: ***+259***	***1 0 0 0 0 0 0 1 1***	*9 bits:* ***-131***	***1 0 1 1 1 1 1 0 1***
8 bits: ***+3***	***0 0 0 0 0 0 1 1***	*8 bits:* ***+125***	***0 1 1 1 1 1 0 1***

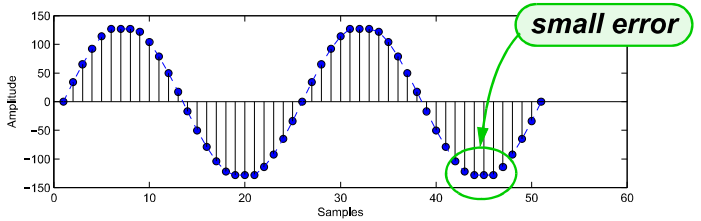
The same would apply if 10, 11,... *n* bits were required to represent the original number.

Notes:

In the two diagrams below, a sine wave of amplitude 135 is represented with 8 bits, using 1) wraparound, and 2) saturation. Notice that when saturation is used, the error resulting from overflow is much less significant - in this case the error manifests as a slight "flattening" at the peaks and troughs of the sine wave.



Wraparound

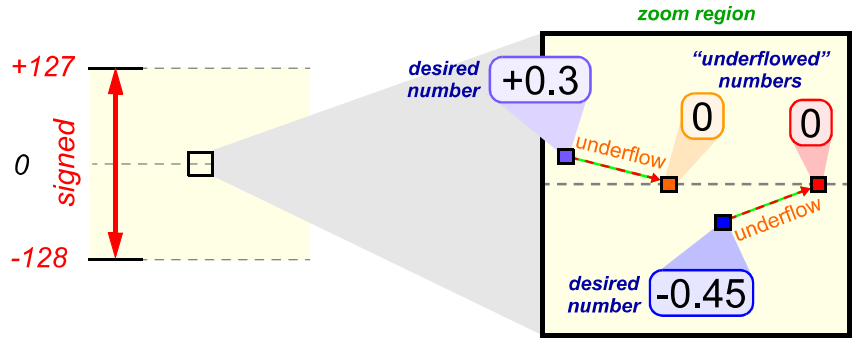


Saturation

Underflow

4.10

- Underflow occurs when the ***magnitude*** of the number is ***too small*** to be represented by the number format.
- In this case, ***non-zero values are quantised to zero*** - i.e. there is a complete loss of amplitude information.
- Underflow is very ***destructive*** - once underflow has occurred, there is effectively no signal left to process!



Non-Integer Values

4.11

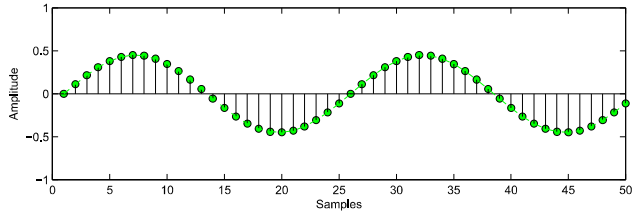
- So far we have considered only ***integer values***, which is rather limiting! The ***quantisation step*** is 1, and the smallest magnitude values we can represent are **+1** and **-1**.
- Often it is useful / necessary to represent values with a ***fractional part***.
- If we wanted to do this using ***decimal*** numbers, we would simply add a ***decimal point*** and continue to add digits after the point. The more digits are added, the greater the precision.
- The same principle can be used in ***binary*** (using a ***binary point***).

27. *(implicit decimal point)*	***0 0 0 1 1 0 1 1 •*** *(implicit binary point)*
27.25	***0 0 0 1 1 0 1 1 • 0 1***
27.28125	***0 0 0 1 1 0 1 1 • 0 1 0 0 1***

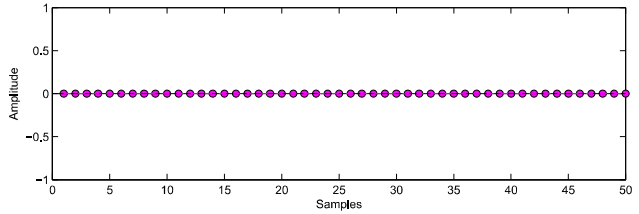
Notes:

In the example below, a sine wave with amplitude 0.45 has been passed through an ADC, but as we are using an 8-bit all-integer representation, every sample of the signal has been rounded to zero. This is an example of underflow.

Underflow could also occur as a result of an arithmetic operation. For example, suppose you had a sine wave of amplitude 45, and then multiplied it by a constant of 0.01. This would give the same sine waveform as below, so underflow would occur if the result of the multiplication were represented with 8 integer bits.



Original

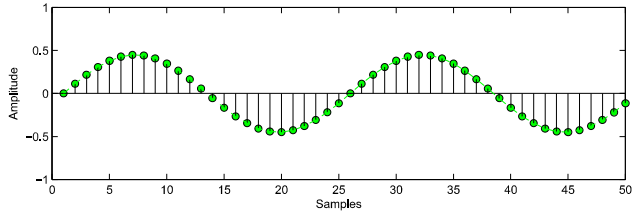


Underflow

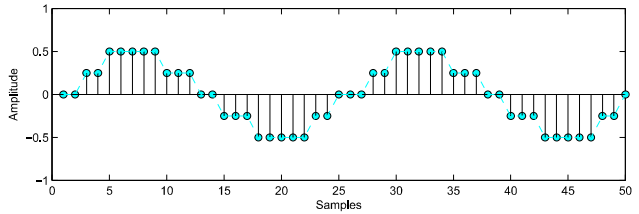
Notes:

In the last slide we considered the problem of underflow, where numbers are too small to be represented by the available word format. A sine wave of amplitude 0.45 was shown, and it was seen that when represented with an all-integer word format, every sample became zero as a result of underflow.

Suppose we now add two bits to the right of the binary point. Now, seven quantisation levels are available between +1 and -1, these being {-0.75, -0.5, -0.25, 0, +0.25, +0.5, +0.75}, and the sine wave information is preserved, albeit in heavily quantised form.



Original



2 bits added after the binary point

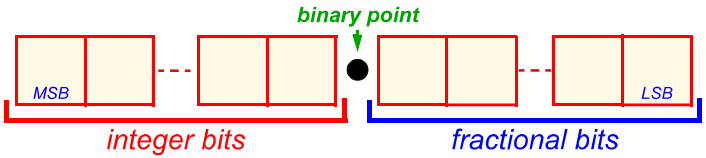
Fixed Point Binary Numbers

4.12

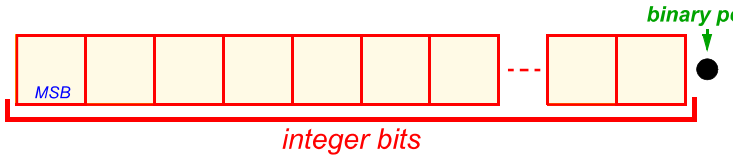
- We can now define what is known as a “fixed-point” number:

  ...a number with a ***fixed*** position for the ***binary point***.

- Bits on the ***left*** of the binary point are termed ***integer bits***, and bits on the ***right*** of the binary point are termed ***fractional bits***.



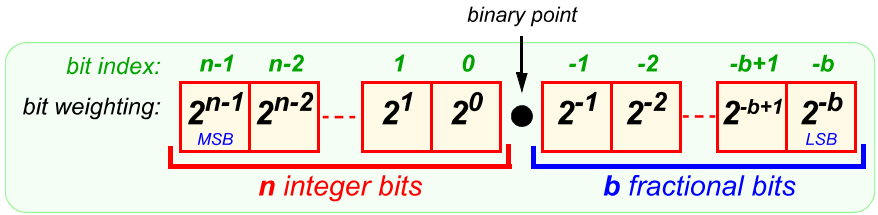
- ***Integer words*** can still be considered fixed point: the binary point is placed at the ***far right*** of the word, and there are ***no fractional bits***.



Fixed Point Format

4.13

- The format of a generic fixed point word, comprising ***n*** *integer bits* and ***b*** *fractional bits*, is:



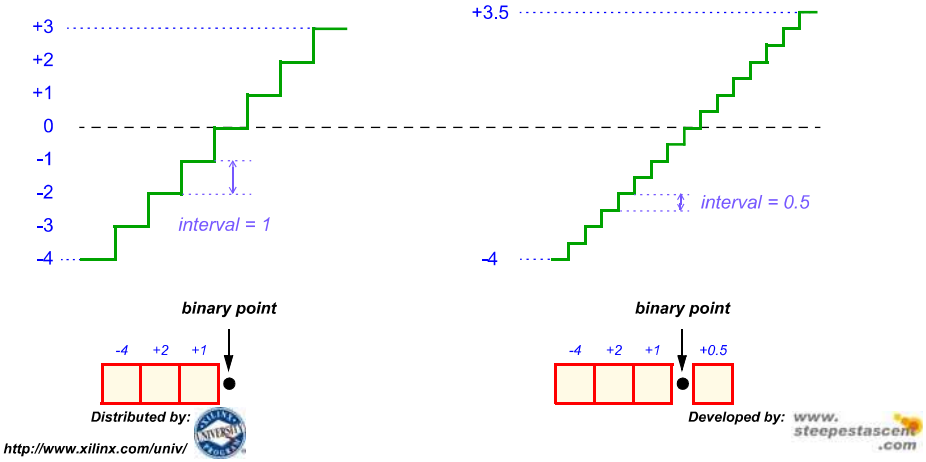
- The weightings of the ***fractional bits*** are:
  - 2^{-1} = **0.5**
  - 2^{-2} = **0.25**
  - 2^{-3} = **0.125**.... .... and so on (halving in value each time).
- The MSB has ***-ve weighting*** for 2’s complement (as for integer words).

Notes:

As fixed point allows fractional numbers to be represented, adding bits to the right of the binary point (fractional bits) has the effect of creating new quantisation levels, equally spaced between the original ones, while retaining (almost!) the same numerical range.

For example, given a 3 bit word comprising all integer bits: the quantisation step is 1; the minimum representable value is -4 (100); and the maximum is +3 (011).

If 1 fractional bit is included (making 4 bits in total): the quantisation step is 0.5; the minimum representable value is still -4 (100.0); and the maximum is +3.5 (011.1).



Notes:

As examples, we consider the 2’s complement word “11010110” with the binary point in two different places.

Firstly, with the binary point to the left of the third bit, i.e. 5 integer bits and 3 fractional bits:

bit index:	4	3	2	1	0	-1	-2	-3	
bit weighting:	-2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	
binary number:	1	1	0	1	0 •	1	1	0	
decimal representation:	$1x-2^4$	$1x2^3$	$0x2^2$	$1x2^1$	$0x2^0$	$1x2^{-1}$	$1x2^{-2}$	$0x2^{-3}$	
	-16	+ 8	+	2	+	0.5	+ 0.25		= -5.25

...and secondly, with the binary point to the left of the third bit, i.e. 3 integer bits and 5 fractional bits:

bit index:	2	1	0	-1	-2	-3	-4	-5	
bit weighting:	-2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	
binary number:	1	1	0 •	1	0	1	1	0	
decimal representation:	$1x-2^2$	$1x2^1$	$1x2^0$	$1x2^{-1}$	$0x2^{-2}$	$1x2^{-3}$	$1x2^{-4}$	$0x2^{-5}$	
	-4	+ 2	+	0.5	+	0.125	+ 0.0625		= -1.3125

Note that these results are related by a factor of $2^2 = 4$, i.e. 4 x -1.3125 = -5.25.

Fixed Point Range and Precision

4.14

- As with integer representations (which are also effectively fixed point numbers, but with the binary point at position 0), the ***binary range*** of fixed point numbers extends from:

unsigned:	**00000.....0000** →	**11111.....1111**
signed (2's comp.):	**10000.....0000** →	**01111.....1111**

- The same number of ***quantisation levels*** is present, e.g. for an ***8-bit*** binary word, ***256 levels*** can be represented.

- ***Numerical range*** scales according to the ***binary point position***, e.g.:

1000 0000. **(-128)** → **0111 1111.** **(+127)**

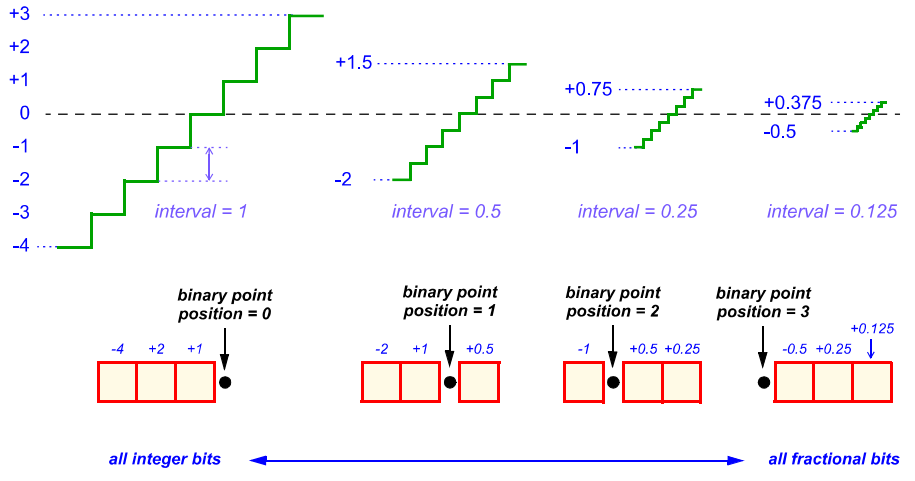
↓1 x0.5 ↓1 x0.5

1000 000.0 **(-64)** → **0111 111.1** **(+63.5)**

- ***Dynamic range*** (**range / interval**) is independent of the binary point position, e.g. (**127**-(**-128**))/1 = 255 = (**63.5**-(**-64**))/0.5

Notes:

To illustrate this further, lets consider the very simple case of a 3-bit 2's complement word, with the binary point in all four possible positions. Clearly the numerical range is affected by the binary point position, but the relationship between the interval and range remains the same.



Fixed Point Addition and Subtraction

4.15

- Adding and subtracting fixed point numbers is ***no different*** from integer binary numbers. The ***binary point*** does not affect the mechanics of the calculation.

- Below are two examples of ***adding fixed point numbers***:

decimal	2's complement binary
-6.4375	1001•1001
+ 5.9375	+ 0101•1111
	0001 1111
-0.5	1111•1000

carry signal

decimal	2's complement binary
1.9375	0001•1111
+ 1.25	+ 0001•0100
	0001 1111
3.1875	0011•0011

carry signal

- Compare these to the calculations shown back on Slide 4.6 - notice that the binary numbers are actually ***the same sequences of bits***.

Notes:

We can easily add together values with different numbers of fractional bits. This effectively means adding "0" bits to the far right hand side, to equalise the number of fractional bits in the each word. However, note that this is an implicit operation and does not need to be written down, nor does it imply any additional calculation (as 1+0 = 1, and 0+ 0 =0)! For example,

1001•1001 + 0101•111101 ≡ 1001•100100 + 0101•111101

Just to confirm that fixed point subtraction is equivalent to integer subtraction, consider the example on the right, and compare to the right hand example on the notes page of Slide 4.6. It should be obvious that the same sequences of bits appear in both cases.

Also note that all of the decimal numbers shown in these examples are 1/16th of those in the original integer examples. Why might this be?

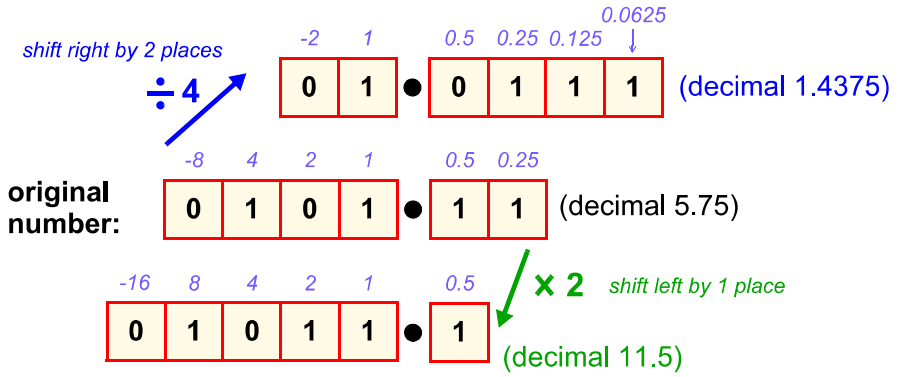
In each case the introduction of 4 fractional bits has effectively shifted the number 4 places to the right. This is equivalent to dividing through by a factor of 2^4, hence the relationship.

decimal	2's complement binary
1.9375	0001•1111
+ (-1.25)	+ 1110•1100
	1111 1100
0.6875	0000•1011

carry signal

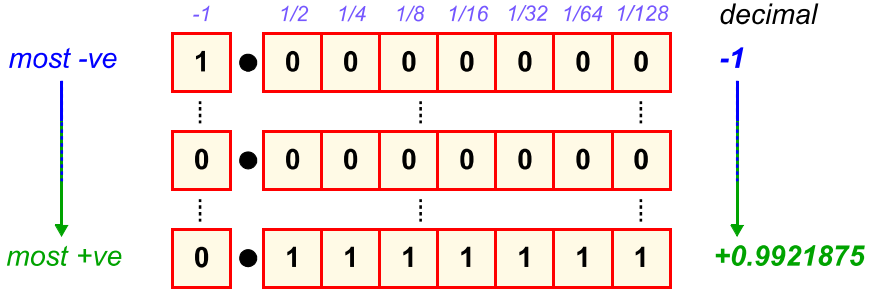
Multiplication & Division via Binary Shifts

- The ***binary point position*** has a ***power-of-2 relationship*** with the ***numerical range*** of the word format, and any number it represents.
- Therefore if we want to ***multiply or divide*** a fixed point number by a ***power-of-two***, this is achieved by simply ***shifting*** the numbers with respect to the binary point!



Normalised Fixed Point Numbers

- Fixed point word formats with ***1 integer bit*** and a number of fractional bits are often adopted.
- Numbers from ***-1*** to ***+1-2^{-b}*** (i.e. just less than 1) can be represented. Some examples:

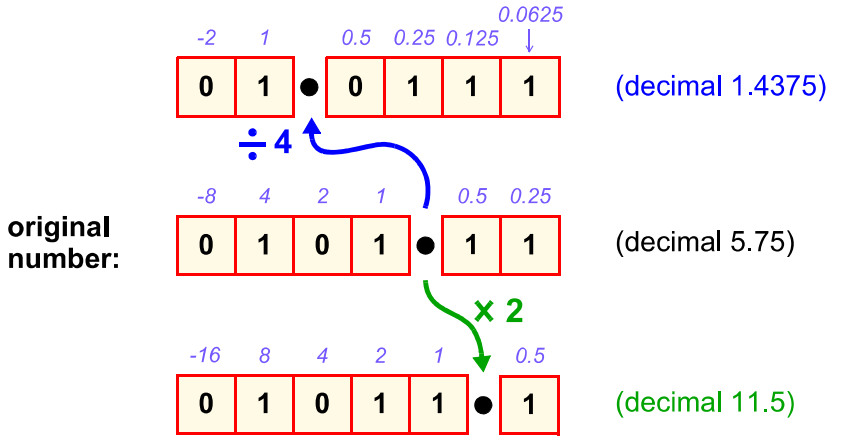


- Limiting the numeric range to ±1 is advantageous because it makes the arithmetic easier to work with... multiplying two normalised numbers together ***cannot*** produce a result greater than 1!

Notes:

Of course, looking at the example in the main slide, we could also consider that the binary point is moved, rather than the bits - it amounts to the same thing! Ultimately the binary point is conceptual - having no effect on the hardware produced - and it falls to the DSP design tool and/or DSP designer to keep track of it.

Reviewing the ***divide-by-4*** and ***multiply-by-2*** examples from the main slide... if we move the binary point, the weightings of the bits comprising the word, and hence the value it represents, change by a power-of-two factor.

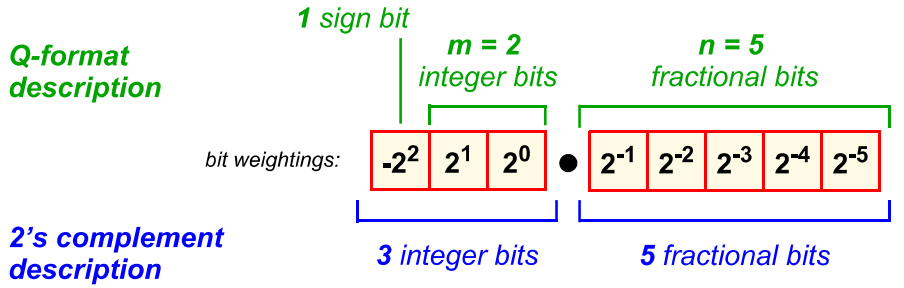


Notes:

The term ***Q-format*** is often used to describe fixed point number formats, usually in the context of DSP processors. However, it is useful to note that Q-format and 2's complement are actually the same thing.

Q-format notation is given in the form $Q_{m.n}$, where m is the number of integer bits, and n is the number of fractional bits. Notably this description *excludes* the MSB of the 2's complement representation, which Q-format considers a sign bit. Therefore the total number of bits in a Q-format number is $1 + m + n$, whereas in 2's complement, the same word format would be described as having m+1 integer bits, and n fractional bits.

For example, a $Q_{2.5}$ number has a sign bit, 2 other integer bits, and 5 fractional bits, and hence can be represented as shown below. In 2's complement, this would be described as having 3 integer bits and 5 fractional bits.

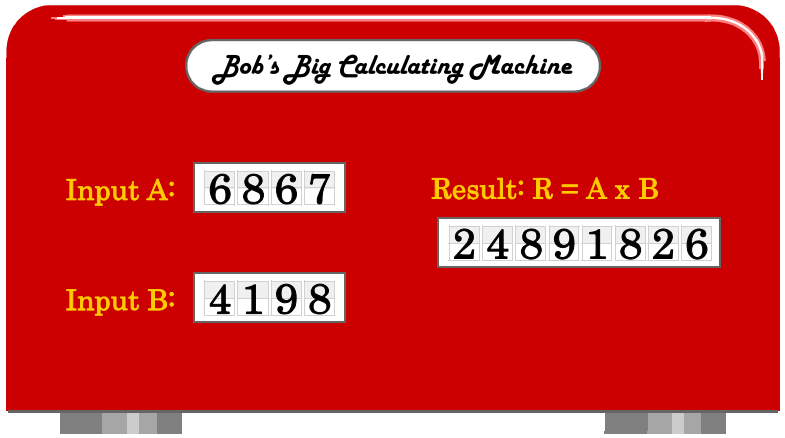


The $Q_{0.15}$ format (often abbreviated to Q_{15}) is used extensively in DSP as it covers the normalised range of numbers from **-1** to **+1 - 2^{-15}**, and is equivalent to a 16 bit 2's complement representation with the binary point at 15, i.e. 1 integer bit and 15 fractional bits.

Motivation to Normalise

4.18

- Usually when we ***multiply*** numbers, we expect them to ***get bigger***!
- Consider this ***4-digit*** decimal multiplying machine... multiplying A x B can produce ***up to 8 significant digits***.



Notes:

The range of the 4 digit inputs, A and B, is -9999 to +9999, so the range of possible results of $R = A \times B$ is -99980001 to +99980001.

However, what happens if we want to use the result as the A input to a further 4 digit multiplication? Clearly, if the result can be up to 8 digits long, there is a good chance that it won't "fit" in the 4 digits allowed for the input.

One solution is to scale down the result, before using it in the next stage of the computation. Scaling down by a factor of 10,000 will ensure that R falls in the range -9999 to +9999. In our example,

$$\frac{R}{10000} = \frac{24891826}{10000} = 2849.1826$$

As there are now fractional digits, these must be truncated to permit a 4 digit representation, so R becomes 2849. Later, after the next stage of the calculation, we must remember to scale the answer back up by a factor of 10000. For example,

$$R_1 = A_1 \times B_1 = 6867 \times 4198 = 24891826$$

$$A_2 = \left\lfloor \frac{28491826}{10000} \right\rfloor = 2849$$

$$R_2 = A_2 \times B_2 = 2849 \times 1627 = 4635323$$

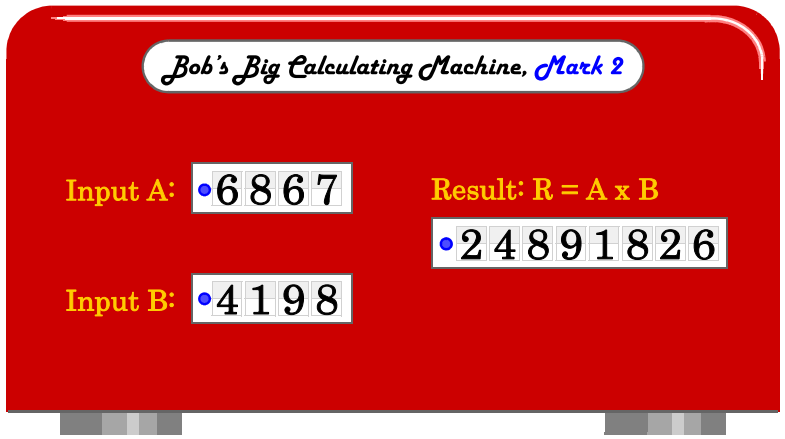
$$\text{(Final result): } F = R_2 \times 10000 = 46353230000$$

This is quite close to the true result of $6867 \times 4198 \times 1627 = 46902612582$, but an error is introduced by truncation, and this cannot be subsequently corrected. Aside from this, the main problem is having to keep track of the scaling, which is inconvenient. Is there a better way? Yes - normalisation.

Normalisation

4.19

- With ***fractional values*** we can ***normalise*** the range to **-1 to +1**. This makes the arithmetic easier to work with.
- The machine has now been improved by inserting a ***decimal point***!



Notes:

In the first example, we had to scale the result and then truncate the fractional part:

`6787 x 4198 = 28491826` **Scale** → `2849.1826` **Tr.** → `2849`

With numbers normalised to the range -1 to + 1, the operation is simpler: no scaling is required, only truncation! Now that a decimal point has been introduced to the calculating machine (i.e. Mark 2), the result displayed by the machine is correct even if the latter 4 bits are dropped, whereas this was not true in Mark 1. The reason is simply that multiplying any two numbers in the range -1 to +1 will give a result which is also in the range -1 to +1.

`0.6787 x 0.4198 = 0.28491826` **Tr.** → `0.2849`

Now the procedure for reducing back to 4 bits is much "easier", and there is no need to keep track of a scaling factor to compensate for later on.

Exactly the same normalisation idea is applied to binary, and the binary point is implicitly used in most DSP systems. Note that in a DSP system, the binary point is all in the eye of the designer. There is no physical connection or wire for the binary point; the *concept* of the binary point just makes things significantly easier in keeping track of wordlength growth, and truncating just by dropping fractional bits.

For example, consider multiplying `00100100 x 01100001 = 0000 1101 1010 0100` (in 2's complement binary, which is equivalent to 36 x 97 = 3492 in decimal). Normalising these binary numbers means dividing the multiplicands through by 128 (retaining one integer bit, with all the rest becoming fractional bits).

The calculation is now `0.0100100 x 0.1100001 = 0.00110110100100`, which is equivalent to 0.28125 x 0.7578125 = 0.213134765625 in decimal. If the wordlength of the result has to be reduced, this can be achieved simply via truncating some of the least significant fractional bits. No scaling is required.

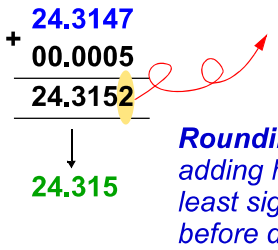
Of course if you prefer to work with integers and are prepared to keep track of the scaling etc yourself, you can do so..... you will get the same answer, and the cost will be the same.

Truncation and Rounding in Decimal

4.20

- ***Truncation*** and ***rounding*** both involve reducing the precision of a number to achieve a shorter number representation.
- Before considering the binary case, let's take a decimal example.
- Assume we have the number **27.3147**, and need to reduce the precision to 3 decimal places.
- You probably already know that the answers are **27.314** with truncation and **27.315** with rounding, but what is the process?

Truncation *simply means "throwing away" the least significant digit(s).*

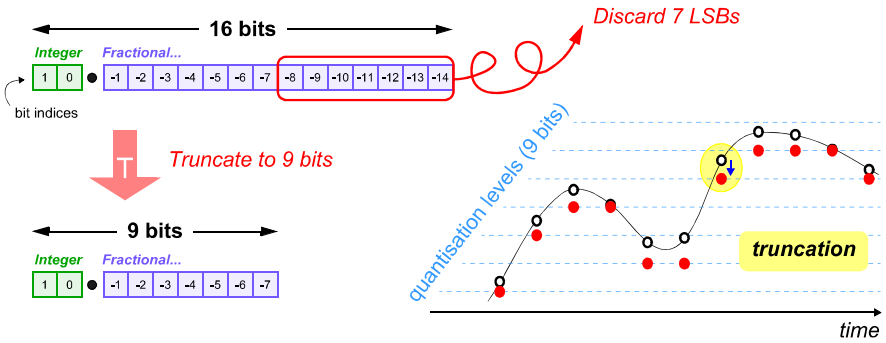


Rounding *involves adding half of the new least significant digit before discarding.*

Binary Truncation

4.21

- Like the decimal case, ***binary truncation*** is the process of simply "removing" bits. This is usually done in a constrained way to convert from a larger to a smaller binary wordlength (thus losing precision).
- Usually truncation is performed on least significant bits (LSBs). This is equivalent to converting to the next lowest quantisation level, according to the new wordlength:

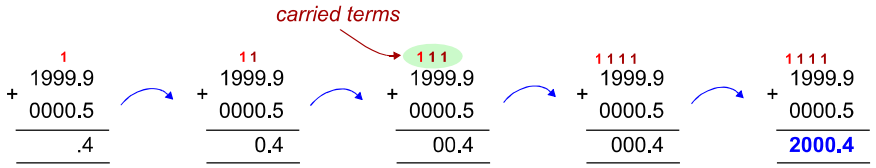


Notes:

Intuitively, if we round a number to the nearest integer then all values greater than or equal to **X.5** are rounded up, and all values less than **X.5** are rounded down (where **X** represents the integer digits).

So, for instance if we had a number in the range **0.5** to **0.999....**, adding **0.5** would result in a number in the range **1.0** to **1.4999...**. Hence the fractional part of the number can then be removed to give the "rounded" result.

Adding a small value like **0.5** is not a trivial operation, however, as the resulting carried terms can propagate all the way to the most significant digits. For example:



The benefit of rounding is that the maximum error is reduced by a factor of 2. For example:

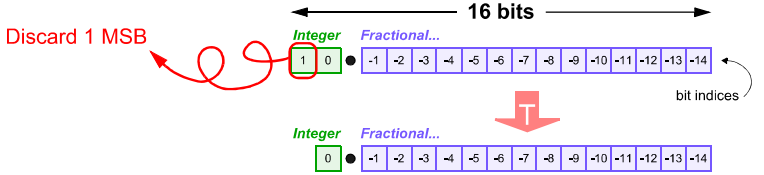
1.9999... is ***truncated*** to **1**, but ***rounded*** to **2**.
(The maximum ***truncation*** error is just less than 1 x the new least significant digit.)

1.4999... is ***truncated*** to **1**, and ***rounded*** to **1**.
(The maximum ***rounding*** error is just less than 0.5 x the new least significant digit.)

Notes:

Consider again the decimal case: truncating 7.8992 to three significant digits is easy - the answer is 7.89. Of course, we truncate the least significant digits resulting in some lost some accuracy (or resolution), but the answer is still representative of the original 5 digit number. If we truncated the most significant digits, leaving '992' (or 0.0992), the result makes little sense in terms of the magnitude of the original number.

In the binary world the concept of truncation of MSB is rare and, as for the decimal example, truncating the MSB is usually catastrophic. However, in some (rare!) instances a sequence of operations may result in a reduction of the overall range of values and therefore merit the removal of MSBs.



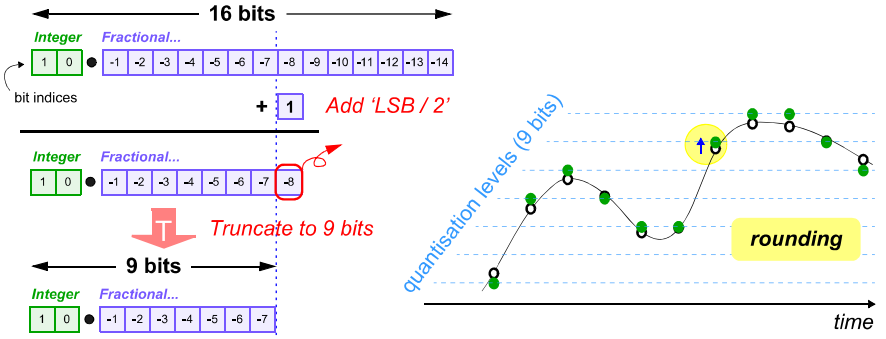
Truncating MSBs can generally only be done when the bits to be truncated are empty. This is shown below for the MSB truncation of the numbers 1.25 and -1.25:

Truncating MSBs is especially problematic when using signed values as the sign bit will be lost.

Binary Rounding

4.22

- Rounding is a more accurate, but more complicated technique that requires an addition followed by a truncation operation.



- This process is equivalent to the technique for decimal rounding, e.g. ***round*** 7.89 to one decimal place is accomplished by ***adding*** 0.05 then ***truncating*** to 7.9. Clearly rounding introduces a smaller error.

- Therefore the process of simple rounding ***requires an add operation***.

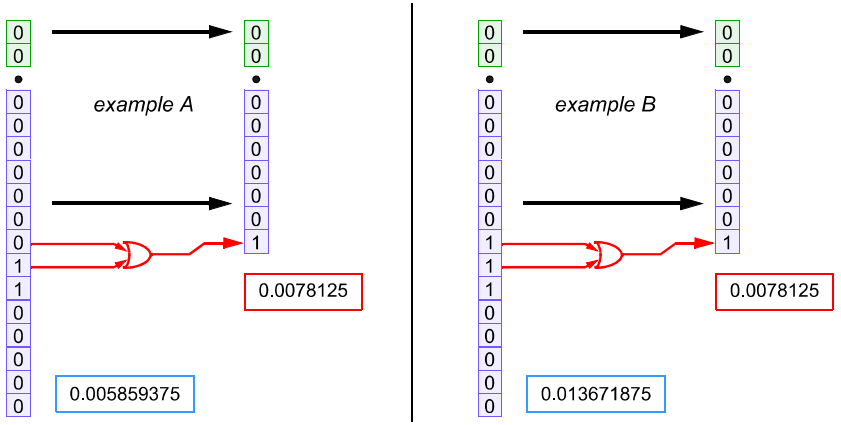
A different approach: Trounding

4.23

- Trounding is a ***compromise*** between truncation and rounding;

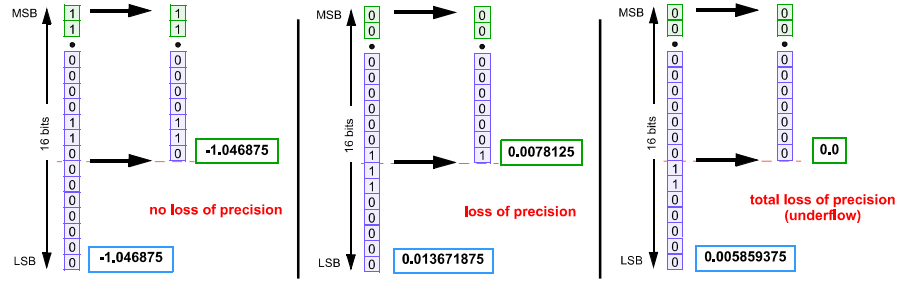
- It preserves information from beyond the LSB like rounding;

- However, unlike rounding it cannot affect any bit beyond the new LSB:

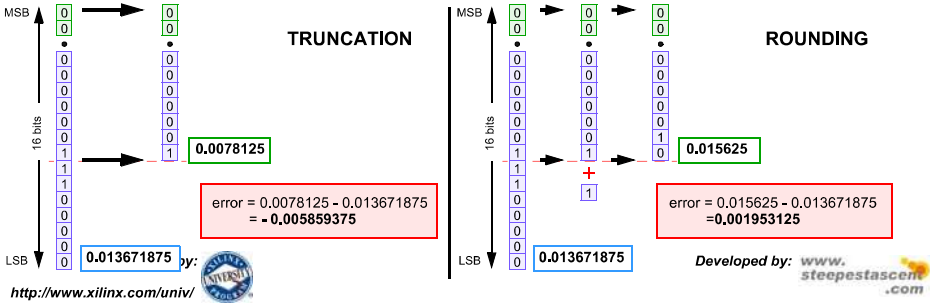


Notes:

Some examples of truncation of LSBs of 16 bit numbers:



The following rounding example is a fairly extreme (but perfectly valid): 0.013671875 is very close to needing to be rounded up (to 0.015625) so truncation results in a significantly larger error than rounding.



Notes:

The simple advantage of "trounding" is that it does not require a full adder and better performance than truncation can be achieved with an OR gate. A very small advantage, but perhaps somewhere this cost saving and performance improvement might just be worth it!

Trounding Explained

- First compare the logical OR operation with addition. Only when both inputs are 1 does trounding differ from rounding:

Input A	Input B	OR	addition
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0 carry 1

- Trounding performs like rounding 75% of the time:
  - 50% of the time, tround=round=truncate
  - 25% of the time, tround=round
  - 25% of the time, tround=truncate
- Trounding has a lower mean quantisation error than truncation, but a higher mean quantisation error than rounding;
- Trounding has a higher quantisation error variance than both rounding and truncation.

Motivation for Floating Point

- The ***range*** of ***2's complement fixed point*** is related to the integer and fractional wordlengths, *n* and *b* respectively, as follows:

$$-2^{n-1} \rightarrow 2^{n-1} - 2^{-b}$$

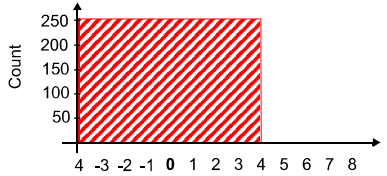
- ***Dynamic range*** is a ratio of largest and smallest representable numbers, as given by

$$20\log\left(2 \times \frac{2^{n-1}}{2^{-b}}\right)$$

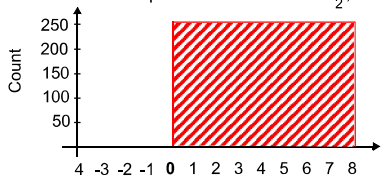
- For example, in ***16 bit*** fixed point, the dynamic range is ***96.3dB***, and for ***32 bit***, the dynamic range is ***192.6dB***.
- What if our application requires a very high dynamic range? Is there a more suitable alternative to fixed point?
  - ***Floating point*** offers high dynamic range (but at a cost!).

Notes:

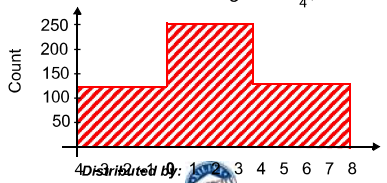
Quantisation Noise pdf: Rounding note range is from -4 to + 4. Mean = 0; Variance = $\frac{q^2}{12}$



Quantisation noise pdf: TruncationMean=$\frac{q}{2}$; Variance=$\frac{q^2}{12}$



Quantisation noise: TroundingMean=$\frac{q}{4}$; Variance=$\frac{7q^2}{48}$ = $1.75\frac{q^2}{12}$

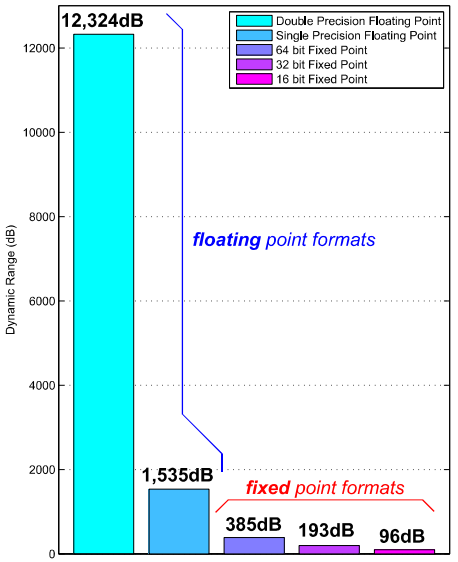


Notes:

As we will see over the coming slides, the concept of floating point is generic, but there is a standardised form of floating point which is in widespread use (as IEEE standard 754). The standard defines two floating point formats (or "precisions") with fixed parameters, and two with flexible parameters. The dynamic ranges of the two fixed precision formats, Single and Double precision, are 1,535dB and 12,324dB respectively.

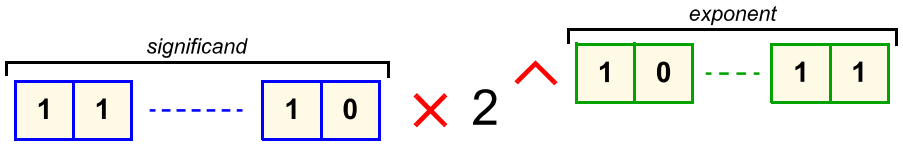
It is not possible to make a general statement on the dynamic ranges of the flexible formats, or indeed proprietary versions of floating point, because this depends not just on the length of the number in bits but also the composition of the number. As will be explained shortly, floating point comprises two sections: a significand and an exponent. A floating point number of length 30 bits with a 20 bit significand and 10 bit exponent has a different dynamic range to one with an 18 bit significand and 12 bit exponent.

To provide a simple comparison between fixed and floating point, the Single precision format (length 32 bits) has a dynamic range of 1,535dB, compared to 192.6dB for a fixed point number of the same length. In fact, to achieve the same dynamic range as Single precision would require a 255 bit fixed point representation!



General Form of Floating Point Numbers

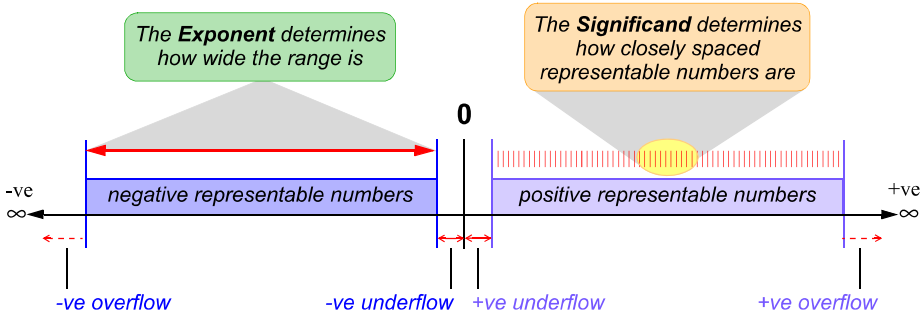
- Floating point numbers comprise two parts: a ***significand*** (sometimes called a "mantissa"), and an ***exponent***.
- The ***significand*** expresses a signed number (in 2's complement form, or sign & magnitude), which is then multiplied by 2, raised to the power of the ***exponent***. This is analogous to ***scientific notation*** in decimal.



- Both the significand and exponent may be positive or negative.
  - The sign of the ***significand*** determines the ***sign*** of the floating point number (or the sign may be conveyed separately).
  - The sign of the ***exponent*** governs the ***magnitude*** of the number. Negative exponents give numbers of magnitude < 1.

Floating Point Range and Precision

- The two sections of the word have different implications for the ***numerical properties*** of the floating point representation.
- The ***significand*** affects how closely spaced representable number are.
- The ***exponent*** defines the range of numbers that can be represented.
- ***Overflow*** and ***underflow*** can occur, as with fixed point arithmetic.



Notes:

Let's consider some binary floating point numbers, alongside the equivalent decimal representations. To keep the examples simple, we'll use a 6 bit significand and a 4 bit exponent, which we assume express the decimal values shown below. Notice that even though only 10 bits are used in total, the dynamic range is large!

The significand and exponent of a floating point number are not necessarily defined using unsigned or 2's complement format, hence the reason for not explicitly showing the binary digits in the diagram below. While the concept of floating point can be applied in a proprietary fashion according to individual requirements, it is more common to use the IEEE 754 Floating Point standard, which in fact defines the representation of the significand and exponent in a particular way (this will be covered in the coming slides).

Floating point number	*Decimal equivalent*
significand = 10 (b b b b b b) x 2 ^ exponent = 3 (a a a a)	$10 \times 2^3 = 80$
significand = -17 (b b b b b b) x 2 ^ exponent = 6 (a a a a)	$-17 \times 2^6 = -1088$
significand = 5 (b b b b b b) x 2 ^ exponent = -7 (a a a a)	$5 \times 2^{-7} = 0.0390625$

Notes:

Floating point arithmetic has finite precision, just as much as fixed point does - neither is capable of representing any arbitrary value exactly, because in both cases quantisation takes place.

Increasing the length of the significand has the effect of introducing more quantisation levels within the defined numeric range (thus increasing precision).

Increasing the length of the exponent means that the range of representable numbers is extended, both towards positive and negative infinity, and also towards zero. This increases the dynamic range.

As a result of these observations, we can confirm that stating the overall length of a floating point format does not fully define its characteristics: the composition of the word, in terms of significand and exponent lengths, is important.

IEEE 754 Standard for Floating Point

4.28

- Floating point has been ***standardised*** as IEEE 754 (and before that, IEEE 854), and this is widely used for developing systems utilising floating point arithmetic.
- The number comprises three parts:
  - ***Significand*** $[b_0 \,.\, b_1\, b_2\, b_3\, ...\, b_{p-2}\, b_{p-1}]$
  - ***Sign bit*** s
  - ***Exponent*** E
- The value, v, represented by the floating point number, X, is given by:

$$v = (-1)^s (2)^E (b_0 \cdot b_1 b_2 b_3 \ldots b_{p-2} b_{p-1})$$

- The ***significand*** conveys magnitude only, and has p bits of precision. This is composed of 1 integer bit and $p-1$ fractional bits.

IEEE 754 Floating Point Word Formats

4.29

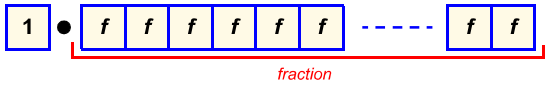
- IEEE 754 defines four precisions as detailed in the table below. The ***Single*** and ***Double*** precision formats are the most commonly used.

Length (bits)	Format			
	Single	Single Extended	Double	Double Extended
Significand	24	≥ 32	53	≥ 64
Exponent	8	≥ 11	11	≥ 15
Total	32	≥ 43	64	≥ 79

- In ***Single*** and ***Double*** formats, the significand is represented using only **23** and **52** bits respectively (one bit is a ***"hidden bit"***, as explained in the notes of Slide 4.28).
- Hence for ***Single*** precision the (stored) word format is:
  23 bit significand + **8** bit exponent + **1** sign bit = **32 bits**
- ... and for ***Double*** precision:
  52 bit significand + **11** bit exponent + **1** sign bit = **64 bits**

Notes:

In IEEE 754, the ***significand*** is normalised, meaning that it takes the following form (note the integer bit is always 1, hence the number expresses a value between 1.0, and just less than 2.0):



For this reason, the integer part of the significand (i.e. the leftmost bit) is implicitly included, but omitted from the word format because it is already known. This implicit integer bit is therefore commonly referred to as the "hidden bit".

The ***sign bit***, s, determines whether the magnitude of the number is multiplied by 1 or -1, i.e.

$s = 0$: $(-1)^s = (-1)^0 = 1$ (expressing positive numbers)

$s = 1$: $(-1)^s = (-1)^1 = -1$ (expressing negative numbers)

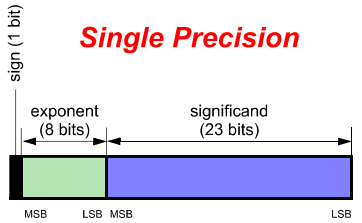
The ***exponent*** is expressed in a biased form in the representation defined by IEEE 754. This means that the sequence of bits stored in the exponent field, e, corresponds to an unsigned integer, whereas the true value of the exponent (the "unbiased exponent", E) may be positive or negative. The *unbiased* exponent is the number to which 2 is raised, as shown by the equation on the main slide.

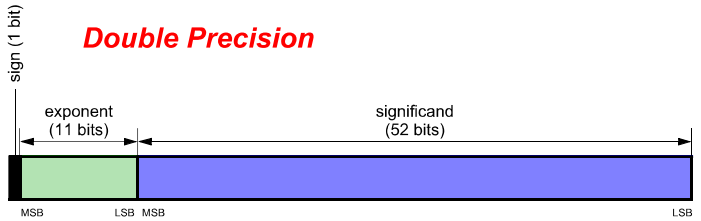
$E = e - bias$

It is worth noting that the origins of IEEE 754 were for computer-based implementation - not FPGAs. There are therefore aspects of the standard which do not map well to FPGA implementation, such as the offset-unsigned representation used for the exponent. 2's complement would arguably be better suited! In fact, floating point cores for FPGAs may internally deviate substantially from the standard, provided that the format conforms to the standard at the input and output, and the functionality implemented is the same.

Notes:

The IEEE 754 standard also defines how these different sections should be arranged to form the integrated floating point word.





The extended formats may vary in their composition; the standard only defines minimum values for the significand and exponent, for both the Single Extended and Double Extended precisions.

Floating Point Exponent

4.30

- Along with the ***bias***, the parameters E_{min} and E_{max} are defined in the IEEE 754 standard for Single and Double precision.

Parameter	Format	
	Single	Double
Exponent bias	+127	+1023
Minimum exponent, E_{min}	-126	-1022
Maximum exponent, E_{max}	+127	+1023

- The ***range*** of the ***unbiased exponent, E,*** is constrained by the parameters E_{min} and E_{max}, e.g. for Single precision $-126 \le E \le 127$.
- Therefore, by applying the bias of +127 according to the table, the range of the ***biased exponent, e,*** is from:

$$E_{min} + bias = -126 + 127 = \mathbf{1} \text{ to } E_{max} + bias = 127 + 127 = \mathbf{254}$$

- Biased exponents **e = 0** and **e = 255** are reserved for special purposes.

Operations in Floating Point

4.31

- The IEEE 754 standard specifies which ***operations*** should be supported by any floating point system based on the standard.
- These operations are:
  - ***Add***, ***subtract***, and ***multiply***
  - ***Divide***, ***square root***, and find the ***remainder***
  - ***Comparisons***
  - ***Rounding*** operations
  - ***Conversion*** between different ***floating point formats***
  - ***Conversion*** between ***integer*** and ***floating point formats***
- Like many standards, however, it defines ***what*** but not ***how***! No guidance is provided on ***implementation*** - this is left to the designer.

Notes:

The exponent bias is related to the length of the exponent in bits, and is designed to shift the unbiased exponent to be (almost) centred at zero. The single precision exponent length is 8, whereas for double precision it is 11. In both cases the bias is given by:

$bias = 2^{n-1} - 1$, where n is the length of the exponent in bits.

The tables below show the special values, ***v***, represented by specific combinations of the biased exponent (**e**) and the fractional part of the significand (**f**), for both Single and Double precision formats.

SINGLE PRECISION	e = 0		e = 255	
	f = 0	f ≠ 0	f = 0	f ≠ 0
Value	$v = (-1)^s 0$ (± zero)	$v = (-1)^s 2^{-126}(0.f)$ (denormalised numbers)	$v = (-1)^s \infty$ (± infinity)	$v = NaN$ (Not a Number)

DOUBLE PRECISION	e = 0		e = 2047	
	f = 0	f ≠ 0	f = 0	f ≠ 0
Value	$v = (-1)^s 0$ (± zero)	$v = (-1)^s 2^{-1022}(0.f)$ (denormalised numbers)	$v = (-1)^s \infty$ (± infinity)	$v = NaN$ (Not a Number)

Denormalised numbers are a special class of floating point numbers where the integer part of the significand (normally assumed to be 1, resulting in significands in the range 1 to 2), is allowed to be 0 (giving significands in the range 0 to 1). This only happens for $E = E_{min}$, meaning that very small values can be represented between 0 and the smallest normalised number. This is sometimes termed "gradual underflow".

Notes:

There are also various rules about how operations involving "special" numbers like infinity should be handled. For example, NaN should be generated as the result of invalid operation such as $\sqrt{-1}$. The IEEE 754 standard document (available via IEEExplore), provides full details.

In terms of FPGA implementation, floating point operators have not yet found widespread use, and this is reflected by the library of components available in System Generator and other similar tools. All of the arithmetic and higher level blocks are based on fixed point. Floating point arithmetic cores are, however, available via tools such as Xilinx Core Generator, and as IP blocks from third party sources.

The main reason for the limited use is that floating point operations are more expensive to implement than fixed point, in terms of the FPGA resources used. They may also be slower to compute, and involve a longer critical path which restricts the maximum clock frequency of the design. Unless a particular application has very demanding requirements in terms of precision or dynamic range, there is no real need for floating point, so fixed point is preferred for its simplicity. This is particularly true if the floating point implementation conforms to the IEEE standard, because it may involve processing longer wordlengths than are strictly required by the application (e.g. the shortest IEEE 754 floating point format is 32 bits long, but your application may only require a tenth of the precision afforded by this format).

One situation where floating point is readily supported on the FPGA is within embedded processors like the MicroBlaze. The MicroBlaze is a "soft processor", i.e. it is implemented using logic fabric resources rather than dedicated silicon. When customising the processor, the designer has the option to include a floating point unit. This is an integrated unit capable of performing all necessary calculations in floating point: however, there is typically only one of these implemented, and operations are scheduled onto it. "Hard" processors like the PowerPC, which are discrete dedicated blocks separate from the main FPGA fabric, can also include floating point support.

If designing with floating point cores implemented on the FPGA fabric, the designer can implement as many of them as he/she wishes to (or has space for!).

Floating Point Multiplication I

- To illustrate why floating point arithmetic is more expensive than fixed point, let's consider a floating point ***multiplier***.
- In floating point, the number has ***three parts***, so the operation is more complex than fixed point, which has ***only one***!
- Floating point multiplication requires the following steps...
  - ***Multiply*** the significands,
  - ***Add*** the exponents,
  - ***Normalise*** the magnitude of the result (if necessary), by adjusting the significand and exponent, and
  - ***Determine the sign*** of the result.
- If performing a fixed point multiplication, only the first of these steps is required. The wordlengths involved may of course vary.

Floating Point Multiplication II

- ***Step 1***, multiply the significands:

$$SIG_R = SIG_A \times SIG_B = 1.375 \times 1.5 = 2.0625$$

- ***Step 2***, add the exponents:

$$E_R = E_A + E_B = 5 + 7 = 12$$

- ***Step 3***, normalise the result by adjusting the significand and exponent (note this is necessary in this case, because the significand $SIG_R > 2$):

$$SIG_{R_{norm}} = \frac{SIG_R}{2} = \frac{2.0625}{2} = 1.03125$$

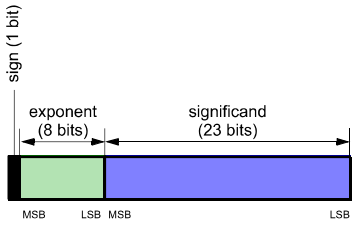
$$E_{R_{norm}} = E_R + 1 = 12 + 1 = 13$$

- ***Step 4***, determine the sign of the result: $S_R = S_A \text{ xor } S_B = 1$

Notes:

To take a numerical example, consider the two floating point numbers, A and B, as given below.

The different parts of the number are arranged as shown in the diagram, i.e. the sign bit comes first, then the exponent (in biased form, from MSB on the left to LSB on the right), followed by the significand, again with MSB on the left and LSB on the right. The table confirms the values of each section of both numbers.



***A* = 1 1000 0100 0110 0000 0000 0000 0000 000**

***B* = 0 1000 0110 1000 0000 0000 0000 0000 000**

	A	*B*
s (sign)	**1**	**0**
E* = e - *bias (unbiased exponent)	132 -127 = **5**	134 - 127 = **7**
SIG (significand)	1.011000....000 = **1.375**	1.1000000.000 = **1.5**

The decimal values of A and B are as calculated below, hence the result $R = A \times B = -44 \times 192 = -8448$:

$$A_{decimal} = (-1)^1 \times 2^5 \times 1.375 = -44$$

$$B_{decimal} = (-1)^0 \times 2^7 \times 1.5 = 192$$

On the next slide, we will calculate the result using floating point arithmetic.

Notes:

Taking all of these parameters into account, the calculated floating point result is as follows:

***R* = 1 1000 1100 0000 1000 0000 0000 0000 000**

	R
s (sign)	**1**
E *(unbiased exponent)*	**13**
e = *E* + *bias* (biased exponent)	13 + 127 = **140**
SIG (significand)	1.000010....000 = **1.03125**

Note that the process of normalisation involves dividing the significand by a factor of 2, and increasing the exponent by 1 (which has the effect of doubling the floating point number). Thus the two cancel out and the values before and after normalisation are numerically identical, as shown below. The purpose of normalisation is to ensure that the significand falls within the allowed range (i.e. between 1 and 2).

$$R_{decimal} = \underbrace{(-1)^1 \times 2^{13} \times 1.03125}_{\textit{normalised}} = \underbrace{(-1)^1 \times 2^{12} \times 2.6025}_{\textit{before normalisation}} = -8448$$

You can also confirm that the result $R = -8448$ is the same as that calculated on the previous notes page.

More Floating Point Arithmetic...

- ***Floating point division*** can be computed in the analogous manner to multiplication...
- 1) ***Divide*** the significands, 2) ***subtract*** the exponents, 3) ***normalise*** as necessary and 4) ***determine the sign*** of the result.

$$D_{significand} = \frac{A_{significand}}{B_{significand}} \qquad D_{exponent} = A_{exponent} - B_{exponent}$$

- ***Addition*** and ***subtraction*** are different, because they require ***two*** normalisation stages to be performed:
  - A first ***normalisation*** before the addition, such that both operands have the same exponent
  - (then the significands are ***added*** and the ***sign determined***)
  - ... and finally a second ***normalisation*** (of the result)

Block Floating Point

- Block floating point is a technique where a block of numbers ***share the same exponent*** (but each have their own ***significand***).
- This means that operations can be performed on numbers from the block ***without having to frequently adjust the exponent /normalise***.
- The shared exponent only needs to be adjusted if the numbers ***grow*** or ***shrink*** too much!
- The value of the shared exponent is defined by the number with the ***largest magnitude***.
- By virtue of the reduced complexity, the ***cost*** of implementing block floating point is therefore ***much lower*** than standard floating point.
- Block floating point is well suited to implementing algorithms like the ***Fast Fourier Transform***, where groups of numbers are processed which may have similar magnitudes.

Notes:

The fact that two normalisations are required in addition and subtraction is significant. In both cases, the significands must be shifted by a variable number of places, and in fact this is not trivial to implement in FPGA hardware (requiring a barrel shifter or similar circuit).

Suppose that *A* and *B* are added, as defined for the multiplication example. We will work in decimal for clarity.

	A	*B*
s (sign)	**1**	**0**
E = e - bias (unbiased exponent)	132 -127 = **5**	134 - 127 = **7**
SIG (significand)	1.011000....000 = **1.375**	1.1000000.000 = **1.5**

Noting that the two exponents are different ($E_A = 5$ and $E_B = 7$), the first step is to equalise the exponents to 7, by altering the significand of *A* appropriately (i.e. dividing by $2^2 = 4$), as shown below:

$$A_{decimal} = \underbrace{(-1)^1 \times 2^5 \times 1.375}_{\text{initial value}} = \underbrace{(-1)^1 \times 2^7 \times 0.34375}_{\text{normalised value}} = -44$$

The adjusted *A* signficand can now be added to the *B* significand, as they both have exponents of 7. In this case, a further normalisation after the addition is *not* required, because the result falls into the range $1 \le R < 2$.

$$R_{decimal} = 2^7 \times (-0.34375 + 1.5) = 2^7 \times 1.15625 = 148$$

The signs of *A* and *B* must also be taken into account, as in the above calculation.

Notes:

Block floating point uses a shared exponent. Each number in the block has the same exponent, but its own significand. In the example below, all of the numbers have an (unbiased) exponent of 10.

This illustrative example is based o Single precision floating point, but block floating point does not need to be based on one of the IEEE 754 standard precisions... the technique is equally applicable where custom lengths are chosen for the exponent and significand.

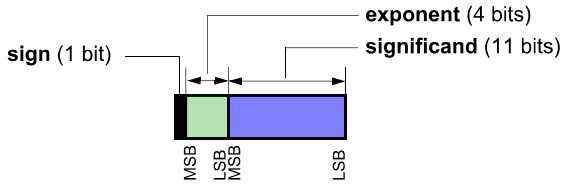
A group of lock floating point numbers

A = 1 1000 1001 0000 1000 0000 0000 0000 000
B = 0 1000 1001 1000 0000 0000 0000 0000 000
C = 0 1000 1001 1110 0000 0000 0000 0000 000
D = 1 1000 1001 0000 0000 0000 0000 0000 000

	A	*B*	*C*	*D*
s (sign)	**1**	**0**	**0**	**1**
E (unbiased exponent)	**10**	**10**	**10**	**10**
e = E + bias (biased exponent)	10 + 127 = **137**	10 + 127 = **137**	10 + 127 = **137**	10 + 127 = **137**
SIG (significand)	1.000010....000 = **1.03125**	1.100000....000 = **1.5**	1.111000....000 = **1.875**	1.000000....000 = **1.0**

Proprietary Formats

- There is no absolute rule that says floating point implementations must conform to the IEEE 754 standard.
- ***More efficient*** implementations could be realised by analysing the requirements of the application in terms of ***precision***, ***dynamic range***, etc., and appropriately choosing the parameters of the word format.
- One possibility is to create a format with a ***short exponent*****, e.g. a** ***4 bit exponent*** and an ***11 bit significand*** (giving 16 bits in total, when the sign bit is included):



- Compared to ***16 bit fixed point***, this format has a much larger ***dynamic range***, at a relatively small cost to ***precision***.

Floating Point for DSP on FPGAs

- Floating point arithmetic is more ***complicated*** and ***expensive*** to implement than fixed point, but it provides a ***much larger dynamic range*** for any given wordlength.
  - To date, the requirements of most DSP applications have favoured ***efficient implementation*** over large dynamic range.
  - Exceptions might be systems like ***radar*** and ***radio telescopes***, and some ***adaptive algorithms***.
- Tool support for floating point is ***not as comprehensive*** as fixed point.
- ***Floating point units (FPUs)*** are commonly featured as optional units of DSP processors, including FPGA-based processors. These units are expensive and typically there is only one of them; floating point operations are ***scheduled*** onto the FPU.
- However with FPGAs, there is the possibility of implementing ***as many floating point operators as required*** - the FPGA is a blank canvas!

Notes:

To confirm the differences between the 16 bit fixed and floating point word formats, let's take a look at some of the key parameters. Note that the same general form has been applied as Single and Double precision, i.e. the exponents have been limited in order to support special values, and it has been assumed that the denormalised numbers are not used. Clearly the dynamic range of the floating point format is greater than fixed point..

	16-bit fixed point	**16-bit floating point**
	2 integer bits, 14 fractional bits	4 bit exponent, 11 bit significand
Smallest magnitude	$2^{-b} = 2^{-14}$	1×2^{-6}
Largest magnitude	2	$2 \times 2^{7} = 2^{8}$
Precision	$2^{-14} = 6.103515625 \times 10^{-5}$	$2^{-11} = 4.8828125 \times 10^{-4}$
Range (decimal)	-2 to +1.99993896484375	-256 to 255.9375
Dynamic range (dB)	$20\log\left(2 \times \frac{2}{2^{-14}}\right) = 96.3\text{dB}$	$20\log\left(2 \times \frac{256}{2^{-11}}\right) = 120.4\text{dB}$

Notes:

One important consideration when implementing any DSP system is the rate at which samples can be processed.

FPUs on FPGA can be fast - in the order of 100 or 200 MFLOPS (Million Floating Point Operations Per Second). However, if there is only one FPU implemented, an algorithm requiring several floating point operations per input sample must re-use this FPU multiple times. This reduces the achievable sampling rate by a potentially large factor.

A more direct alternative is simply to implement a floating point operator (add, multiply, etc.) every time one is required by the algorithm - just like the fixed point approach. Of course, implementing floating point is costly, so it may be desirable to reduce the amount of resources required by serialising the algorithm.

There may of course be appropriate compromises between the two extremes, for example instantiating several floating point operators and time-sharing them according to some other mechanism.

Conclusions

- This chapter has covered the representation of numbers in binary, properties of these numbers, characteristics and simple operations.
- In particular, we have reviewed:
  - Number representation formats, including ***unsigned*** and ***2's complement***, ***fixed point*** and ***floating point***.
  - Properties of numbers, including ***range*** and ***precision***.
  - ***Truncation*** and ***rounding***, ***saturation*** and ***wraparound***, and the implications for numerical integrity / implementation cost;
  - Working with ***normalised*** fixed point numbers
  - ***Floating point*** representations and operations.

Notes:

FPGA Arithmetic II: Operations and Circuits

Introduction

5.1

- The implementation of ***arithmetic*** operations in FPGA hardware is an integral and important aspect of DSP design.
- The following key issues are presented in this chapter:
  - ***Structures for arithmetic operations:***
    Addition/Subtraction,
    Multiplication (and the various options available!)
    Division
    Square Root;
  - ***Circuitry for rounding and saturation***
    The costs involved with choosing these options
  - ***Complex arithmetic*** operations:
    Complex addition and multiplication
  - ***Mapping to Xilinx FPGA hardware***
    ... including special resources for high speed arithmetic

THIS SLIDE IS BLANK

Notes:

This section of the course will introduce the following concepts:

Addition - hardware structures for addition, differences between unsigned and 2's complement circuits, saturation and rounding logic, Xilinx-specific FPGA structures for addition.

Multiplication - hardware structures for multiplication, constant multiplication, shift-and-add multiplication, Xilinx-specific FPGA structures for multiplication (DSP48x slices), multiplier implementation options.

Division.

Square root.

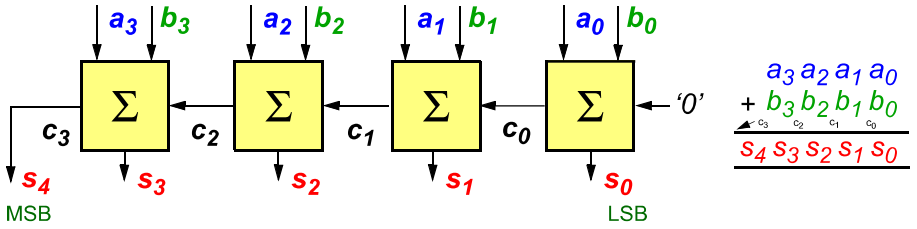
Complex addition.

Complex multiplication.

Multi-bit Adder Structure (Unsigned)

5.2

- A ***full adder*** circuit is capable of adding together two 1-bit numbers.
- A cascade of ***full adder*** circuits can be used to add two multi-bit numbers, ***A*** and ***B***. The following diagram is for 4 bit input words:



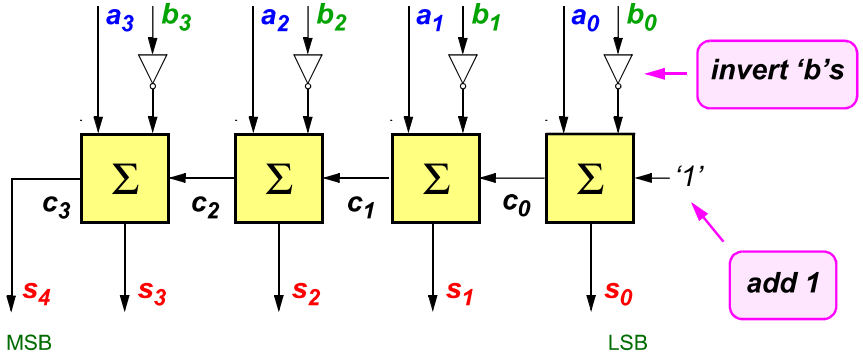
- This chain can be ***extended*** to any number of bits. Note that the last carry output (c_3) forms the most significant bit of the output sum, ***S***.
- Without this most significant carry out, ***overflow*** could occur.
- The above structure is easily adapted into a ***multi-bit subtractor***.

Subtractor (Unsigned)

5.3

- Subtraction is very readily derived from addition: **S = A + (-B)**.
- To form -B, all we need to do is ***invert each bit of the B input***, and then ***add 1*** (which happens via the carry). A and -B are then added together.

4-bit subtractor:

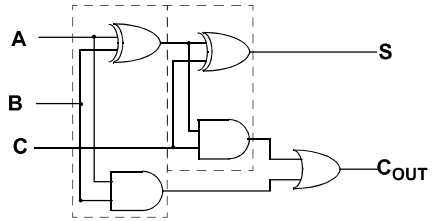


Notes:

The truth table for the full adder is:

A	B	C_{IN}	S	C_{OUT}	
0	0	0	0	0	0+0+0 = 0
0	0	1	1	0	0+0+1 = 1
0	1	0	1	0	0+1+0 = 1
0	1	1	0	1	0+1+1 = 2
1	0	0	1	0	1+0+0 = 1
1	0	1	0	1	1+0+1 = 2
1	1	0	0	1	1+1+0 = 2
1	1	1	1	1	1+1+1 = 3

Full adder circuitry can be therefore produced with gates:

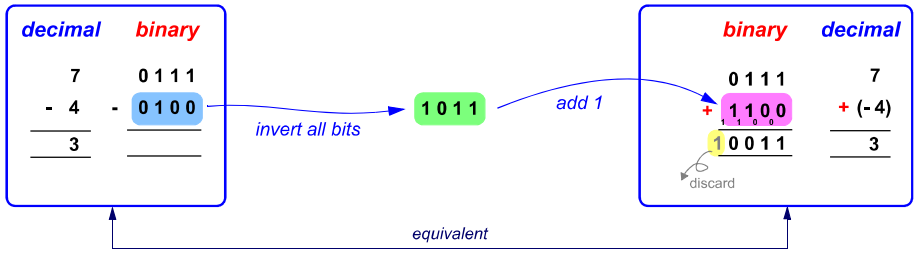


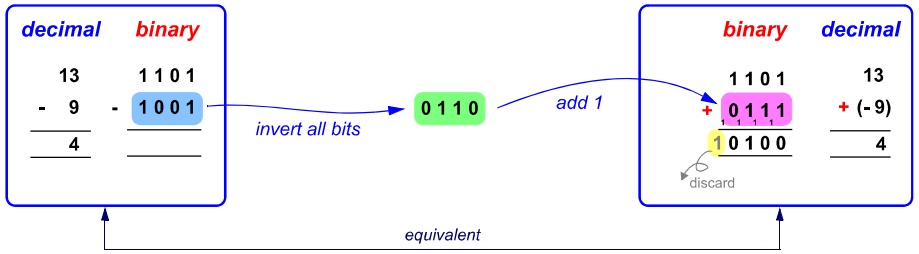
$$S_{out} = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$
$$= A \oplus B \oplus C$$
$$C_{out} = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$
$$= AB + AC + BC = C(A \oplus B)$$

The longest propagation delay path in the above full adder is "two gates".

Notes:

Let's consider a couple of numerical examples, sticking with our simple 4-bit words...



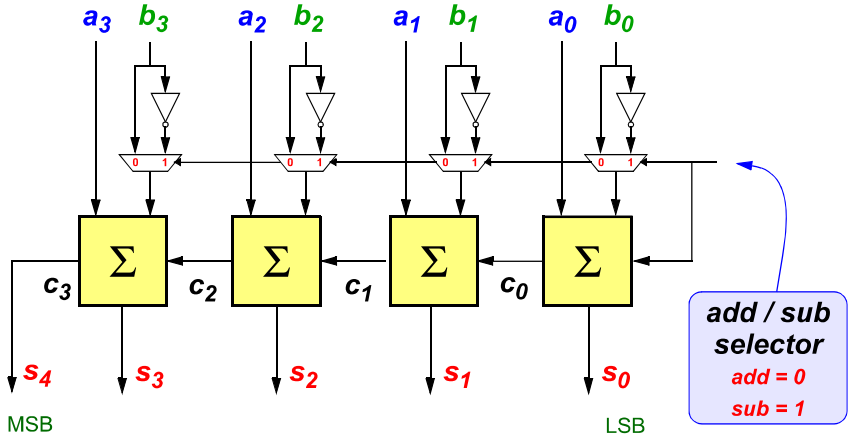


Combined Adder / Subtractor

5.4

- Sometimes we need a ***combined*** adder / subtractor with the ability to ***switch between modes***...

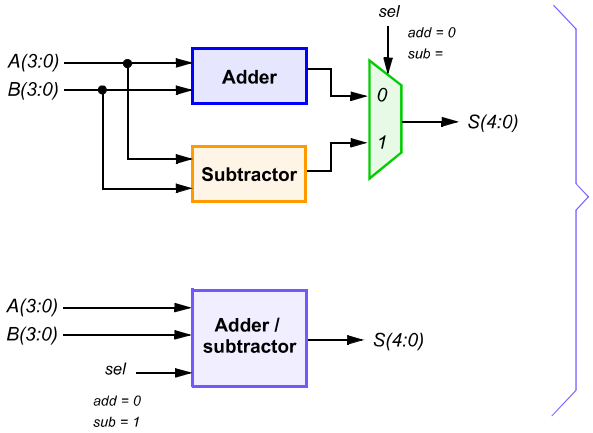
4-bit adder/subtractor:



Notes:

This sort of circuit is useful if you are implementing an algorithm which dynamically chooses between an addition and a subtraction operation. A good example is the CORDIC algorithm (see the CORDIC chapter for further details).

The alternative would be to implement both an adder and a subtractor, and choose between the results using a multiplexer - however, this would be more costly to implement.



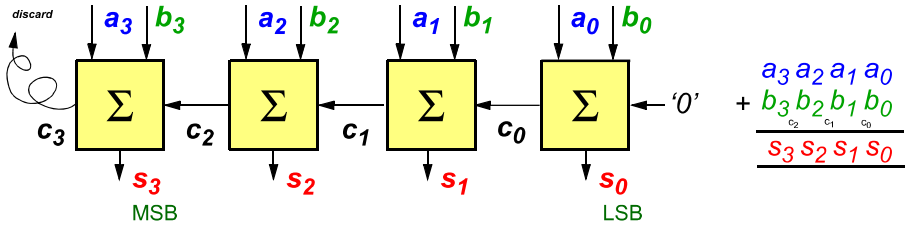
Two possible implementations of a dynamic adder / subtractor

sel = 0	sel = 1
S = A + B	S = A - B

2's Complement Addition / Subtraction

5.5

- When adding or subtracting ***2's complement*** numbers, the final ***carry out (c_3)*** must be ***discarded***, resulting in the circuit shown below:



$$\begin{array}{r} a_3\,a_2\,a_1\,a_0 \\ +\ b_3\,b_2\,b_1\,b_0 \\ \hline s_3\,s_2\,s_1\,s_0 \end{array}$$

(c_2, c_1, c_0 are the internal carries)

- Notice here that ***two 4-bit numbers*** (range **-8** → **+7**) are being added together, to produce a result which is ***also represented in 4 bits***.
- The answer could therefore be in the range **-16** → **+14**, but with 4 bits we can only represent **-8** → **+7**, so ***overflow*** is likely to occur!!
- Overflow has the usual consequence of ***wraparound*** or ***saturation***.

Notes:

Suppose that we wanted to add the two 4-bit numbers A and B. Clearly, if the two numbers have opposite signs, the magnitude of the result will be smaller than either of the inputs (e.g. 5 + -2 = 3). Overflow is possible only when the numbers have the same signs (e.g. 5 + 5 = 10, or -6 + -7 = -13).

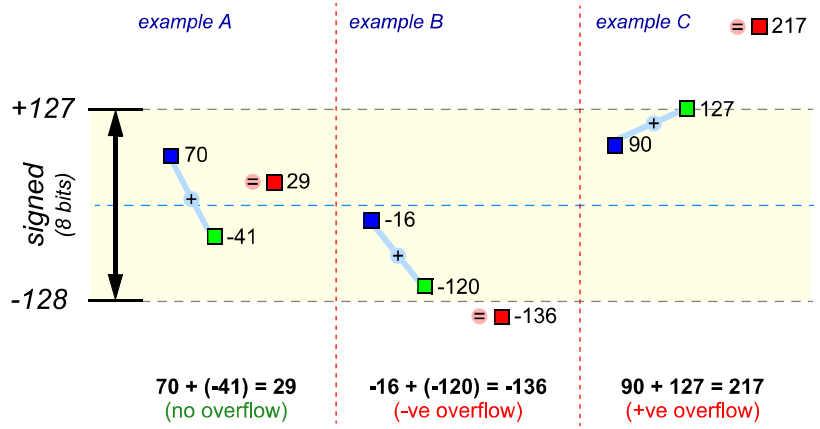
Assuming that the signs are the same, then randomly chosen numbers in the range 0 to +7 will sum to value greater than 7 about half of the time. Similarly, adding two numbers in the range -8 to -1 will produce an overflow about half of the time.

Should overflow occur, this will manifest as wraparound, unless saturation circuitry has been incorporated.

2's Complement Adder Overflow

5.6

- In the example below, pairs of ***8-bit numbers*** are added - note there is a ***good chance*** that the result ***cannot be represented in 8 bits***!

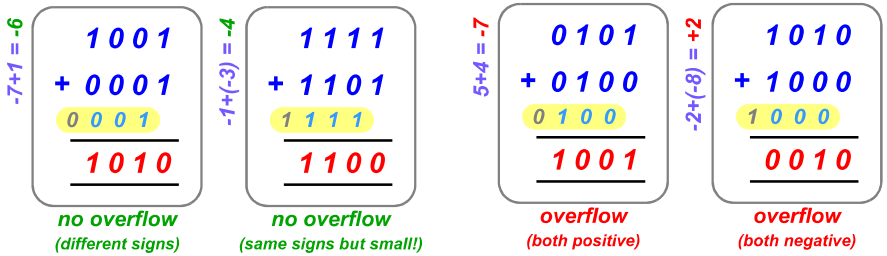


- Overflow can only occur when adding ***numbers of the same sign***.

Saturation Logic

5.7

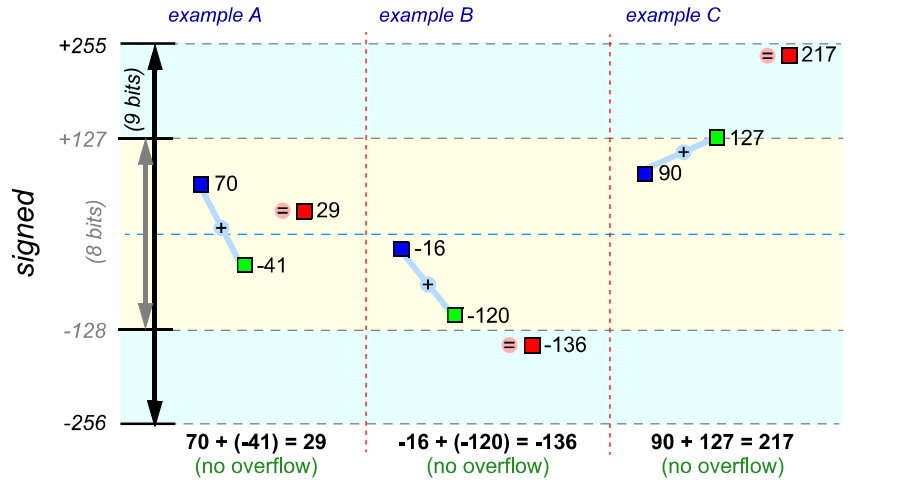
- 2's complement overflow can be ***detected*** by comparing the MSBs of the two inputs, and the output.
  - If the ***input MSBs*** have ***different signs*** (e.g. 1 and 0 for -ve and +ve), overflow cannot occur.
  - If the two ***input MSBs*** have ***the same sign*** (e.g. 0 and 0 for +ve), and the ***MSB of the output*** has a ***different sign*** (e.g. 1 for -ve), the calculation has ***overflowed***.
- Compare the input and output MSBs in the 4-bit examples below:



Notes:

In order to ensure that overflow does not occur, the output must be represented using 9 bits, or in the general case, 1 bit more than the inputs (if the inputs have different lengths, the longer of the two inputs).

For example, the maximum possible result from an 8-bit addition is 127 + 127 = 254, which requires 9 bits to represent. Similarly, the minimum possible result is -128 + (-128) = -256, which also requires 9 bits.



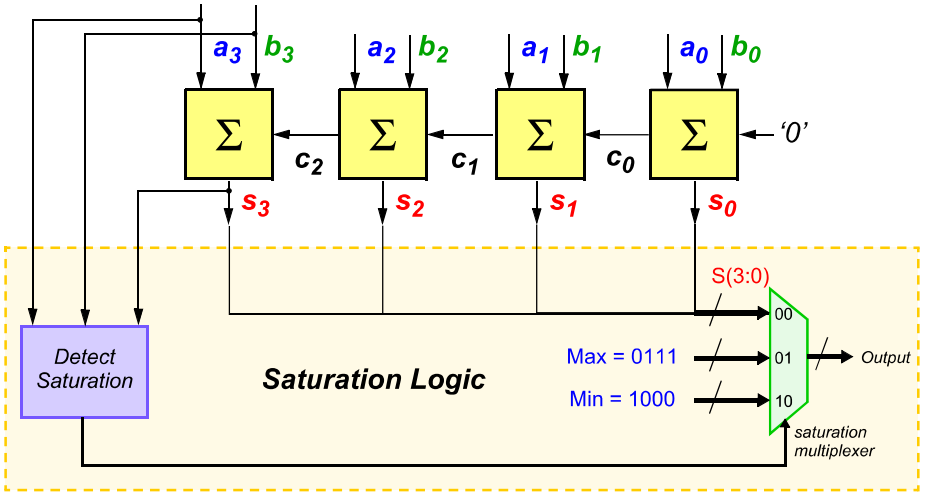
Distributed by: http://www.xilinx.com/univ/

Developed by: www.steepestascent.com

Notes:

The saturation circuit has the job of inspecting these three bits of interest, and switching in the most positive or most negative value when appropriate. This takes the form of an additional circuit to the main adder, hence ***there is a cost associated with implementing saturation!***

The illustration below is for the four bit case: notice that there are two new elements in the circuit.



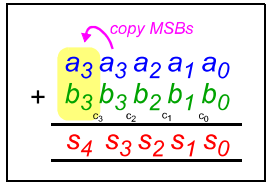
Distributed by: http://www.xilinx.com/univ/

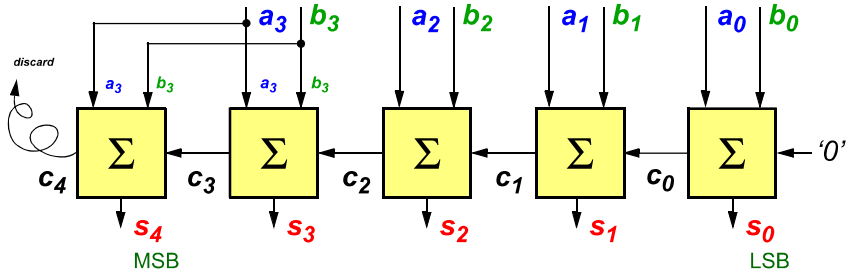
Developed by: www.steepestascent.com

2's Complement Sign Extension

5.8

- 2's complement addition / subtraction produces a result with ***the same wordlength as the inputs***, risking ***overflow***.
- To avoid this, the technique of ***sign extension*** is used to ***increase the wordlengths*** of both input operands, before adding them.
- This simply involves ***copying the MSBs once***, hence one extra full adder is required.

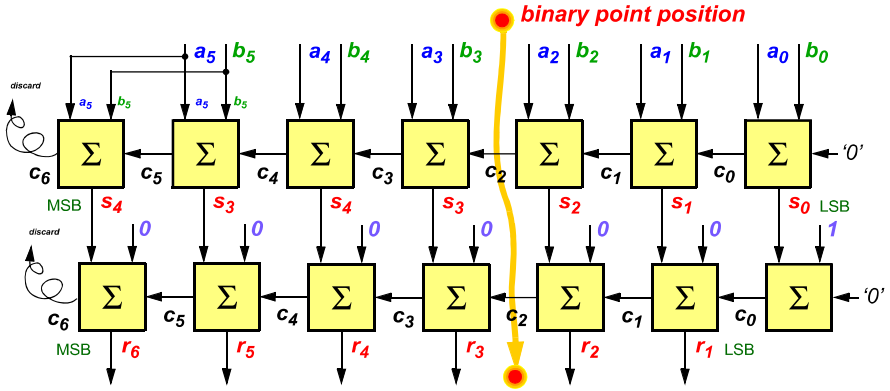




Rounding Logic

5.9

- Rounding is ***more accurate*** than truncation, but requires an ***extra addition stage***.
- Here we add two 6-bit numbers (3 integer and 3 fractional bits, [3:3]), then round the answer to [4:2] ... rounding ***doubles the cost*** in this case! The bottom row of adders is ***entirely due to rounding***.



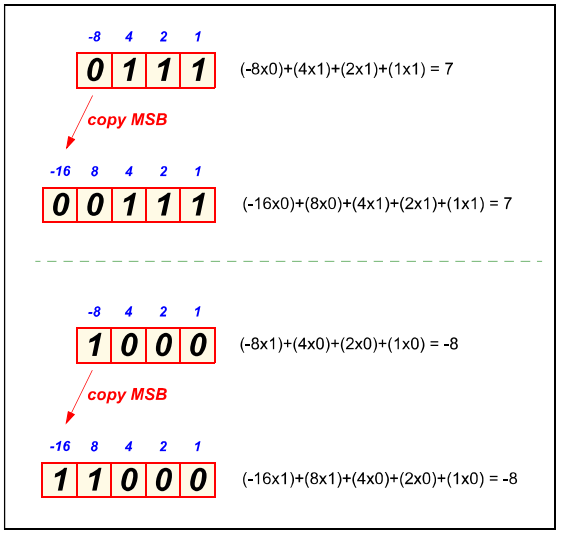
Notes:

For example, suppose you started with the four bit inputs **0111** (+7) and **1000** (-8). These are the largest and smallest numbers that can be represented in 4-bit 2's complement, i.e. the range is **-8 to +7**.

Sign extension involves copying the MSBs to form new MSBs, hence the words become **00111** (still +7), and **11000** (still -8). However, because the words are now five bits long, the range has been increased to **-16 to +15**.

In 2's complement addition, where we add two *n*-bit numbers to produce an *n*-bit result, this preceding step of sign extending the inputs by 1 bit ensures that the result actually has ***n*+1** bits. Therefore, overflow cannot occur.

More generally, sign extension can be used to increase the length of a 2's complement word by an arbitrary number of bits, while preserving the value represented.



Often overflow is "designed out", by choosing long enough numeric wordlengths to ensure it never happens. Wordlengths might be chosen assuming worst case arithmetic growth, or perhaps by performing simulation and analysis using representative input signals.

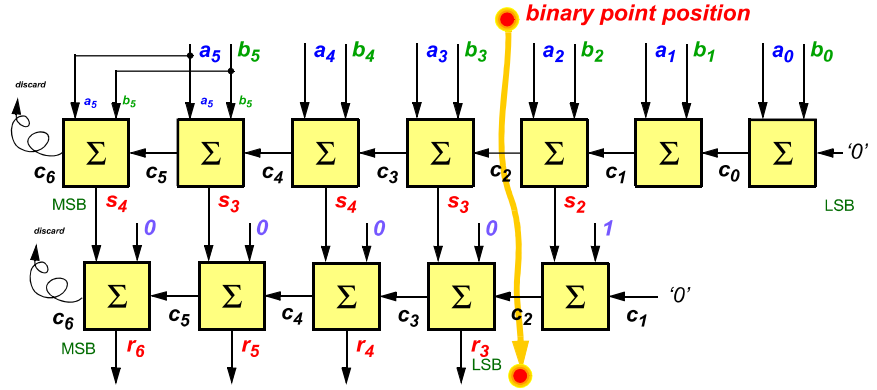
Distributed by: *http://www.xilinx.com/univ/*

Developed by: www.steepestascent.com

Notes:

Rounding does not necessarily double the cost of an addition (as it does in the main slide); the extra cost involved depends on the degree of rounding taking place. When we round, we have to add half of the least significant bit of the new number.

As an example, suppose we wanted to perform the same addition as in the main slide, but this time we wanted to round to [4:0], i.e. 4 integer bits and 0 fractional bits. In this case, the rightmost adder would be at the first fractional bit, thus requiring 2 fewer adders than the original example where we retained 2 bits after the binary point. The amended structure is shown below.



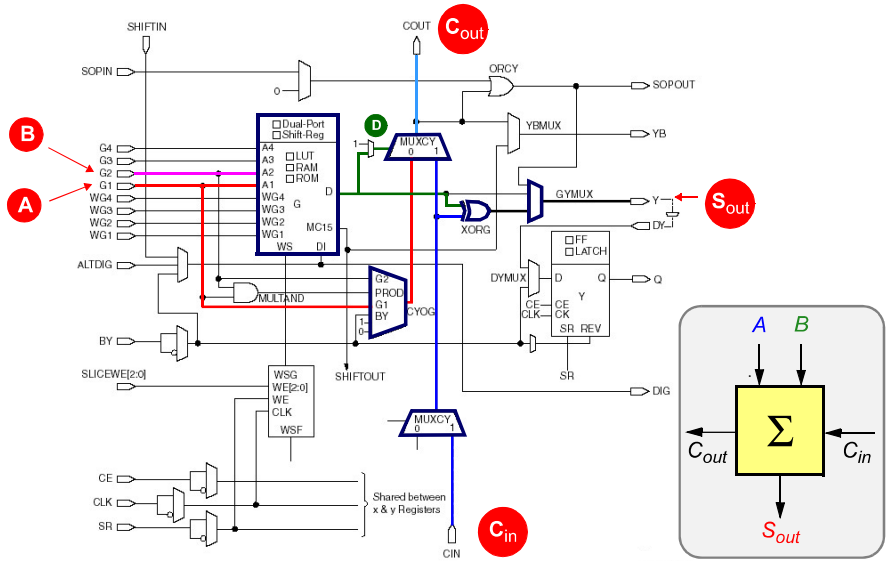
Distributed by: *http://www.xilinx.com/univ/*

Developed by: www.steepestascent.com

One Bit Addition in Slice Logic

5.10

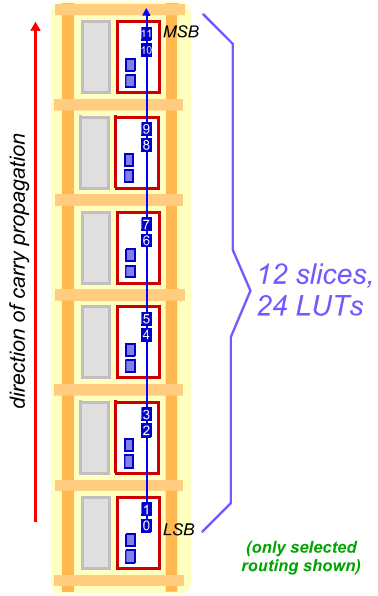
- This shows a 1-bit addition implemented on a single 4-input LUT.



Building Adders from Cascaded Slices

5.11

- Multi-bit adders are produced by ***cascading*** LUTs together.
- Depending on the device family and generation, slices have ***2 or 4 LUTs***.
- Therefore each slice is capable of performing ***2 or 4 bits*** of addition.
- The Xilinx tools will automatically cascade the carry bits through adjacent slices in a ***vertical column***.
- In the example on the right, ***12 slices*** have been cascaded together. Assuming each slice contains ***2 LUTs***, this adder can add ***24 bit*** *unsigned numbers**, or ***23 bit*** *signed numbers**.



Notes:

Picture of Xilinx-II Pro slice (upper half) taken from "Virtex-II Pro Platform FPGAs: Introduction and Overview", DS083-1 (v2.4) January 20, 2003 (see http://www.xilinx.com). Although quite an old device (in FPGA terms!) this Virtex-II Pro schematic shows a typical view for a 4-input LUT FPGA.

Here is a truth table for the Lookup Table (LUT) programmed with a two-input XOR function:

G1 (A)	G2 (B)	D
0	0	0
0	1	1
1	0	1
1	1	0

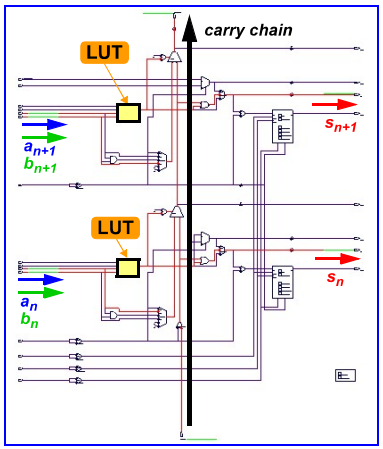
$S_{out} = C_{in}$ xor D , $C_{out} = DA + C_{in}\,\overline{D}$ (multiplex operation). The result is the FULL ADDER implementation:

G1 (A)	G2 (B)	C_{in}	D	S_{out}	C_{out}
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	1	0
0	1	1	1	0	1
1	0	0	1	1	0
1	0	1	1	0	1
1	1	0	0	0	1
1	1	1	0	1	1

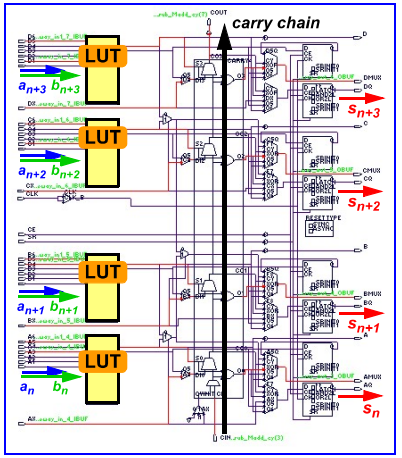
Notes:

* Additionally, one half-slice may be used to propagate the final carry out from the last full adder.

Note that different FPGA families have different slice configurations. Two further examples are given below.



Spartan 3A DSP Slice, showing two LUTs and the carry signal propagating from the bottom to the top. (Implements 2 bit add.)



Spartan 6 Slice, showing four LUTs and the carry signal propagating from the bottom to the top. (Implements 4 bit add.)

Multiplication in binary

5.12

- Multiplying in binary follows the same form as in decimal:

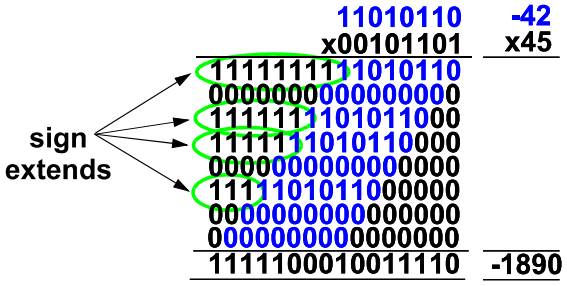
```
          11010110  A7...A0
        x00101101  B7...B0
        ----------
          11010110
         000000000
        1101011000
       11010110000
      000000000000
     1101011000000
    00000000000000
   000000000000000
   ----------------
   0010010110011110  P15...P0
```

- Note that the product *P* is composed purely of selecting, shifting and adding *A*. The *i*th column of *B* indicates whether or not a shifted version of *A* is to be selected or not in the *i*th row of the sum.
- So we can perform multiplication using just full adders and a little logic for selection, in a layout which performs the shifting.

2's Complement Multiplication

5.13

- For one negative and one positive operand just remember to sign extend the negative operand.



Notes:

Multiplication in decimal

Starting with an example in decimal:

```
  214
 x45
----
 1070
+8560
----
 9630
```

Note that we do $214 \times 5 = 1070$ and then add to it the result of $214 \times 4 = 856$ *right-shifted by one column.*

For each additional column in the second operand, we shift the multiplication of that column with the first operand by another place.

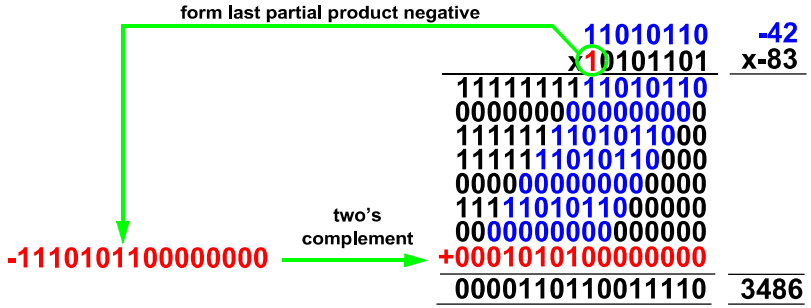
```
  zzz
 xaaaa
-----
 bbbb
+cccc0
+dddd00
+eeee000   etc...
```


Notes:

2's complement multiplication (II)

For both operands negative, subtract the last partial product.

We use the trick of inverting (negating and adding 1) the last partial product and adding it rather than subtracting.



Of course, if both operands are positive, just use the unsigned technique!

The difference between signed and unsigned multiplies results in different hardware being necessary. DSP processors typically have separate unsigned and signed multiply instructions.

Fixed Point multiplication

- Fixed point multiplication is no more awkward than integer multiplication:

```
          11010.110                    26.750
        x00101.101                   x5.625
        -----------                  -------
          11.010110                  0.133750
         000.000000                  0.535000
        1101.011000                 16.050000
       11010.110000                133.750000
      000000.000000                150.468750
     1101011.000000
    00000000.000000
   00000000.000000
   -----------------
   0010010110.011110
```

- Again we just need to remember to interpret the position of the binary point correctly.

Notes:

Multiplier Implementation Options

- ***Distributed*** multipliers
- ***Constant*** multipliers
  - Using the logic fabric (LUTs)
  - Using block RAM
- ***Shift-and-add*** "multipliers"
- High speed ***embedded*** multipliers
  - 18 x 18 bit multipliers
- High speed integrated ***arithmetic slices*** (DSP48s)
  - Multiply, accumulate
  - Add, multiply, accumulate

Notes:

Over the next few slides we will see that multipliers can be implemented in a variety of different ways. As multipliers are used extensively in DSP, implementing them efficiently is a priority consideration.

The most basic multiplier is a 2-input version which is implemented using the logic fabric, i.e. the lookup tables within the slices of the device. This type is referred to as a *distributed* multiplier, because the implementation is distributed over the resouces in several slices.

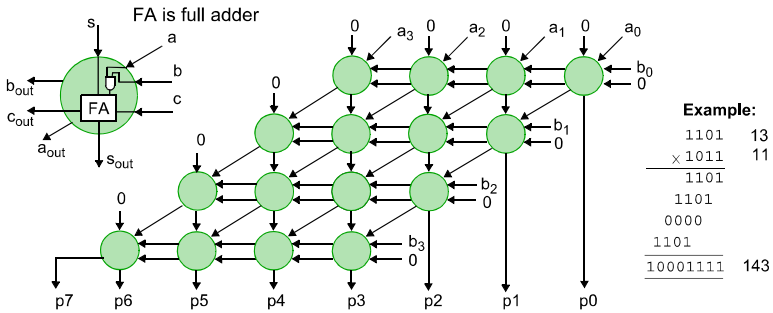
In the case of multiplication with a constant, which is commonly required in DSP, the knowledge of one multiplicand can be exploited to create a cheaper hardware implementation than a conventional 2-input multiplier. Two approaches that will be discussed in the coming pages are ROM-based constant multipliers, and "shift-and-add" multipliers which sum the outputs from binary shift operations.

The FPGA companies are well aware that DSP engineers desire fast and efficient multipliers, and as a result, they began incorporating *embedded multipliers* into their devices in the year 2000. Since then the sophistication of these components has increased, and they have been extended to feature fast adders and in many cases longer wordlengths, too. We can now think of them as embedded arithmetic slices, rather than simply multipliers.

Distributed Multipliers

5.16

- This figure shows a four-bit multiplication:



Example:

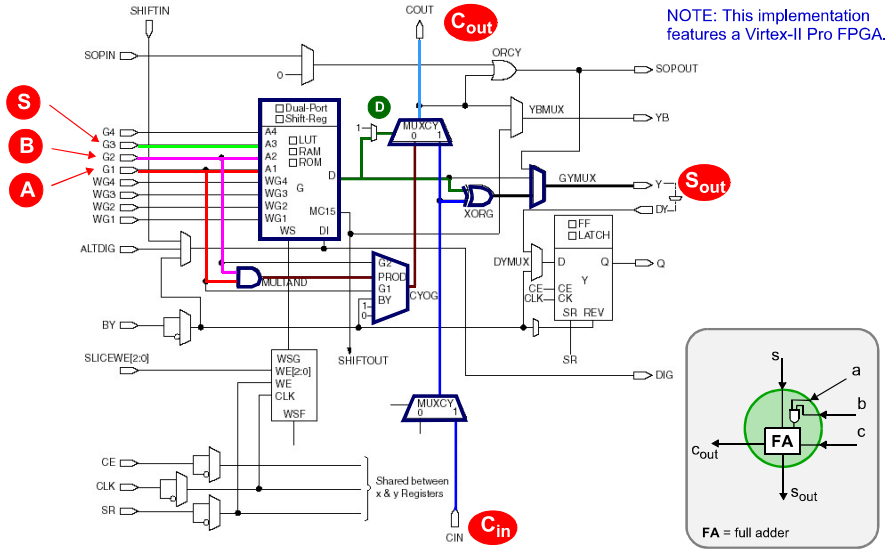
1101	13
× 1011	11
1101	
1101	
0000	
1101	
10001111	143

- The *AND* gate connected to *a* and *b* performs the selection for each bit. The diagonal structure of the multiplier implicitly inserts zeros in the appropriate columns and shifts the *a* operands to the right.
- Note that this structure does not work for signed two's complement!

Distributed Multiplier Cell

5.17

- Half of a slice with two 4-input LUTs can implement one multiplier cell...



Notes:

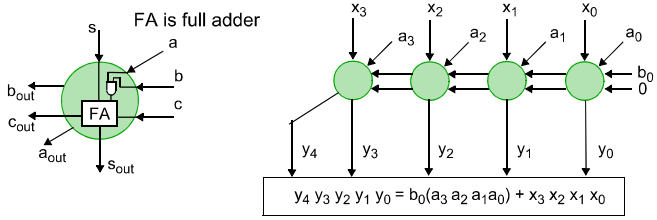
Note the function of the simple AND gate.

The operation of multiplying 1's and 0's is the same AND 1's and 0's

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

Z = A x B (where x = multiply) or in Boolean algebra Z = A $_{and}$ B = AB

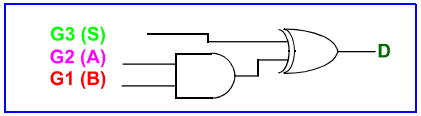
Hence the AND gate is the bit multiplier. The function of one partial product stage of the multiplier is as shown below.



Notes:

Picture of Xilinx-II Pro slice (upper half) taken from "Virtex-II Pro Platform FPGAs: Introduction and Overview", DS083-1 (v2.4) January 20, 2003. http://www.xilinx.com

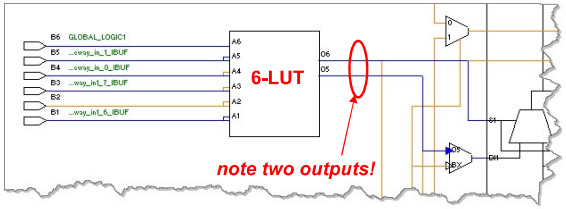
LUT implements the XOR of two ANDs:



The MULTAND unit is required as the intermediate product G1G2 cannot be obtained from within the LUT, but is required as an input to MUXCY. The two AND gates perform a one-bit multiply each, and the result is added by the XOR plus the external logic (MUXCY, XORG):

$S_{out} = C_{IN}$ xor D, $C_{OUT} = DAB + \overline{C_{IN}}D$

Hence this structure will implement one cell of the multiplier. Note that the 6-input LUTs featured on more recent FPGAs can actually implement two logic functions at the same time, and therefore a multiplier cell can be implemented using only the logic resources of 1 LUT, and the MULTAND is no longer required.

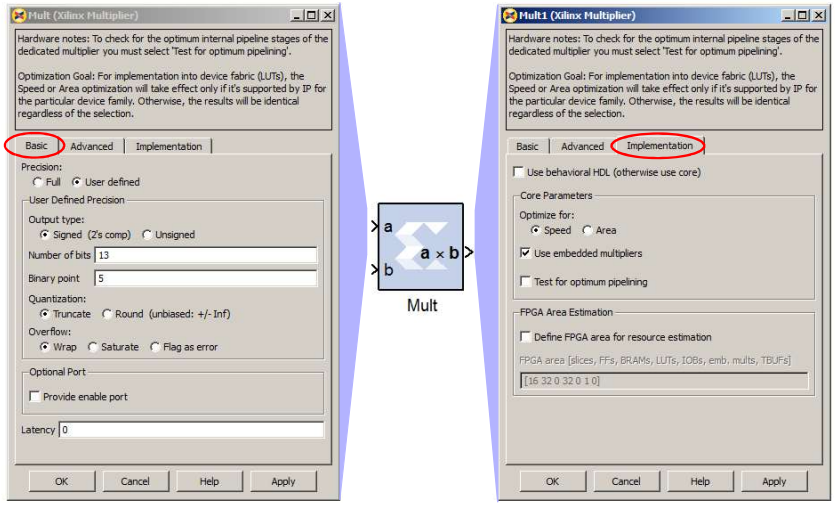


Note that whereas the signal flow graph of the distributed multiplier shows signals propagating from the top and right of the diagram to the bottom, the internal structure of the FPGA slice logic results in a different configuration when implemented on a device (i.e. a set of slice columns).

Multipliers in System Generator

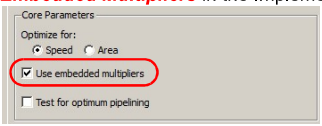
5.18

- The multiplier block in System Generator gives the DSP designer control over several aspects of its implementation.



Notes:

Note in particular the option to ***Use Embedded Multipliers*** in the Implementation tab:



If we UNCHECK this option, the multiplier will be created in a distributed implementation.

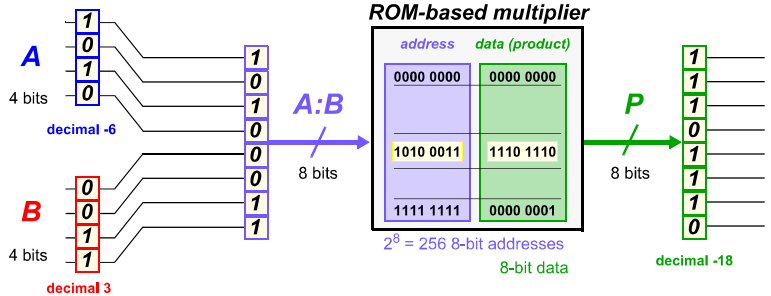
If we CHECK this option, the multiplier will be created using one of the embedded multipliers available on the FPGA. In later generations of FPGAs, this will actually be a multiplier from one of the DSP48 integrated arithmetic slices.

Note that in distributed multipliers, the total cost relates directly to the wordlengths specified. As will be discussed later, this is not strictly the case with embedded multipliers, because they have fixed wordlengths (18 x 18 or 18 x 25 bits).

ROM-based Multipliers

5.19

- Just as logical functions such as XOR can be stored in a LUT as shown for addition, we can use storage-based methods to do other operations.
- By using a ROM, we can store the result of every possible multiplication of two operands.
- The two operands ***A*** and ***B*** are concatenated to form the ***address*** with which to access the ROM. The value stored at that address is the multiplication ***result, P***:



Notes:

There is one serious problem with this technique: as the operand size grows, the ROM size grows exponentially. For two N bit input operands, there are 2^{2N} possible results, and hence the ROM has 2^{2N} entries. The output result is $2N$ bits long, and in total $2N \times 2^{2N}$ bits of storage are required.

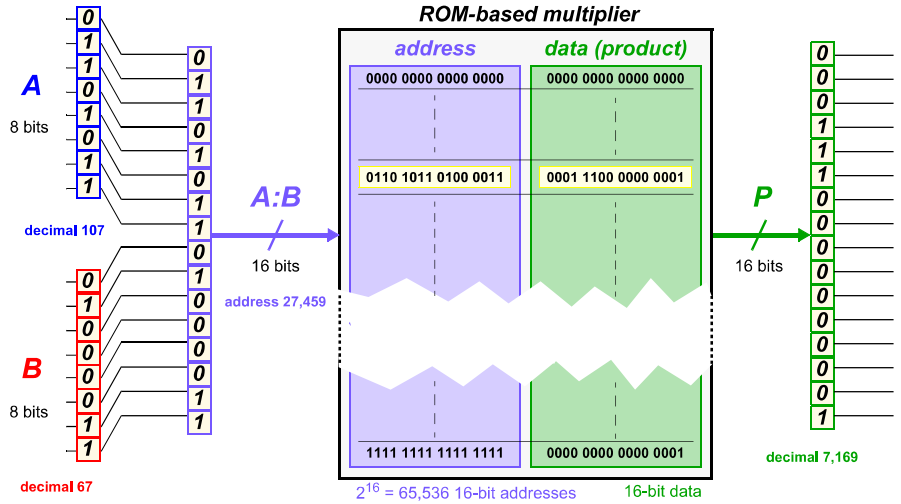
For example, with 8 bits operands (a fairly reasonable) size, 1Mbit of storage is required - a large quantity. For bigger operands e.g. 16 bits, a huge quantity of storage is required. 16 bit operands require 128Gbits of storage and hence a ROM-based multiplier is clearly not a realistic implementation choice!

Input Wordlength (N)	Output Wordlength ($2N$)	No. of ROM entries (2^{2N})	Total ROM Storage ($2N \times 2^{2N}$)
4	8	2^8 = 256	2 Kbits
6	12	2^{12} = 4,096	48 Kbits
8	16	2^{16} = 65,536	1 Mbit
10	20	2^{20} = 1,048,576	20 Mbits
12	24	2^{24} = 16,777,216	384 Mbits
14	28	2^{28} = 268,435,456	7 Gbits
16	32	2^{32} = 4,294,967,296	128 Gbits
18	36	2^{36} = 68,719,476,736	2.25 Tbits
20	40	2^{40} = 1,099,511,627,776	40 Tbits

Input Wordlength and ROM Addresses

5.20

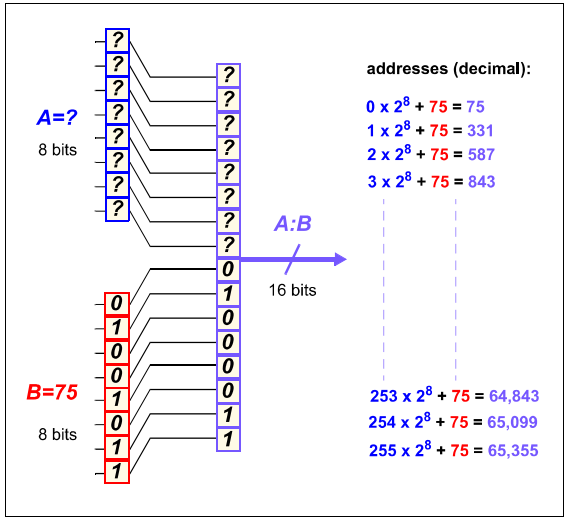
- Consider a ROM multiplier with 8-bit inputs: ***65,536*** 8-bit locations are required to store all possible outputs... so ***1Mbit*** of storage is needed!



Notes:

For example, if the *B* input was the constant value 75, the possible input words would be composed of 256 possible combinations of the upper 8-bits of the address, concatenated with the 8-bit binary word **0100 1011**, as shown in the diagram. The result is that only 256 of the 65,536 memory locations are actually accessed.

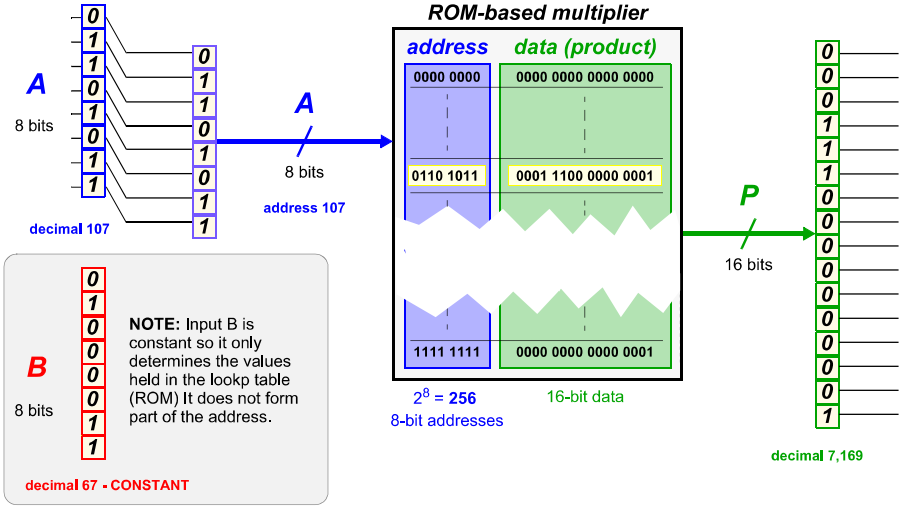
Therefore, when one of the inputs to the ROM-based multiplier is fixed, the size of the required ROM can be reduced to 256 locations of 16-bit data (note that the precision of the stored output words remains 8 bits + 8 bits). The total memory required is thus 256 x 16 = 4kbits.



ROM-based Constant Multipliers I

5.21

- If one input is constant, the ROM required for the multiplier is much smaller: in our 8 × 8 example, only **2^8 = 256** entries are needed!



Notes:

Depending on the value of the constant, it may also be possible to reduce the length of the stored results. For instance, if the value of *B* is (decimal) 10, the maximum output product generated by the multiplication of *B* with any 8-bit input *A* will be:

$-128 \times 10 = -1280$

As -1280 can be represented with 12 bits, that represents a further saving of 4 bits storage x 256 memory locations = 1kbit.

The total storage requirement for this example constant coefficient multiplier would therefore be 3 kbits... significantly smaller than the 1Mbit needed for a 16-bit multiplier where both operands are unknown!

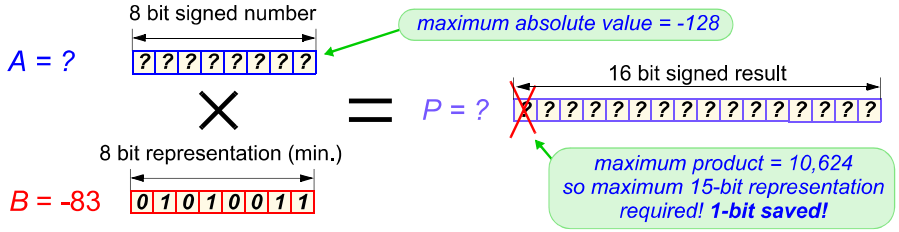
ROM-based Constant Multipliers II

5.22

- ROM-based multipliers with a constant input require ***fewer addresses***.
- The storage required for output words may also be reduced, if the ***maximum result*** does not require the full numerical range of:

$$-2^{2N-1} \le \text{result} \le 2^{2N-1} - 1$$

- The maximum product and output wordlength can be calculated for the ***particular constant value***, and the multiplier optimised accordingly...

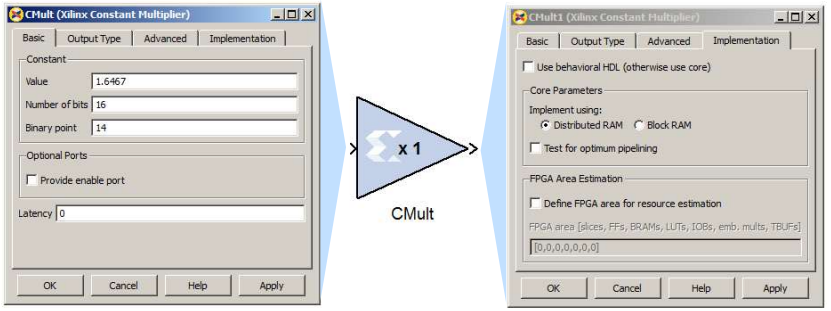


- ***Additional optimisations*** allow cost to be reduced further.

Notes:

Constant multipliers can be implemented using the LUTs within the logic fabric ("distributed ROM"), or with one or more of the Block RAMs available on most devices. The selection is influenced by the other demands placed on these resources by the rest of the system being designed.

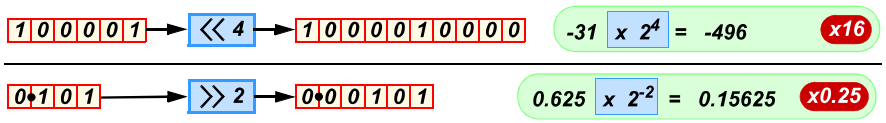
In System Generator, the designer can specify the implementation style via the Constant Multiplier dialogue box, along with the constant value, the output wordlength, and other parameters.



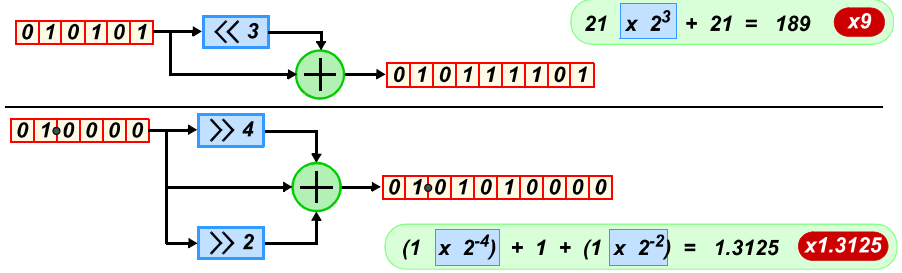
Multiplication by Shift and Add

5.23

- Multiplication by a ***power-of-2*** can be achieved simply by ***shifting*** the number to the right or left by the appropriate number of binary places.



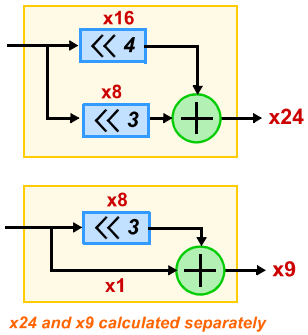
- Extending this a little, multiplications by other numbers can be performed at low cost by creating partial products from ***shifts***, and then ***adding*** them together.



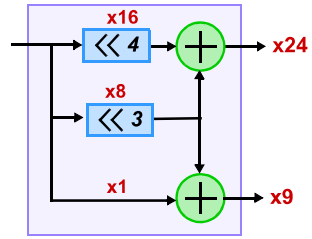
Notes:

Shift operations are effectively "free" in terms of logic resources, as they are implemented only using routing. Therefore multiplications by power-of-two numbers are very cheap! By recognising that multiplications by other numbers can be achieved by summing *partial products* of power-of-two shifts, any arbitrary multiplication can be decomposed into a series of shifts and add operations. The "closer" the desired multiplication is to a power-of-two, i.e. the fewer partial products that are required, the fewer adders are required, and hence the lower the cost of the multiplier.

This type of multiplier is suitable only for constant multiplications, because there is only one input, and the result is achieved using the configuration of the hardware.



x24 and x9 calculated separately



combined - fewer partial products

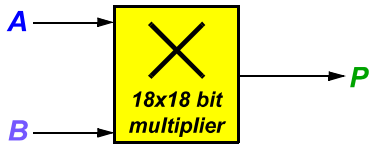
The technique can be particularly powerful when applied to parallel multiplications of the same input. The partial product terms common to several multiplications can be shared and thus the overall effort reduced. Transpose form filters are very suitable for optimisation in this way.

Taking the above simple example of two concurrent multiplications, one x9 and the other x24, it is clear that the shift right by three places can be shared as x8 is common to both operations.

Embedded multipliers

5.24

- The Xilinx Virtex-II and Virtex-II Pro series were the first to provide “on-chip” multipliers in early 2000s.
- These are in hardware on the FPGA ASIC, not actually in the user FPGA “slice-logic-area”. Therefore are permanently available, and they use no slices. They also consume less power than a slice-based equivalent and can be clocked at the maximum rate of the device.

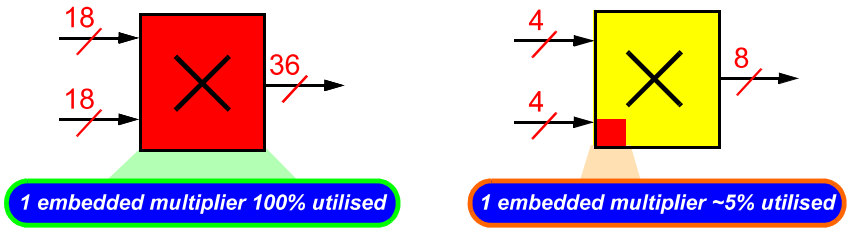


- ***A*** and ***B*** are 18-bit input operands, and ***P*** is the 36-bit product, i.e. $P = A \times B$.
- Depending upon the actual FPGA, between 12 and more than 2000 (Virtex 6 top of range) of these dedicated multipliers are available.

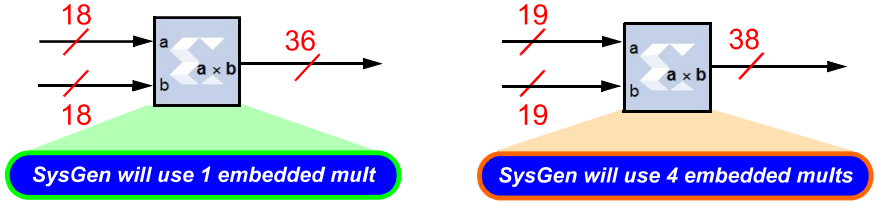
Embedded Multiplier Efficiency

5.25

- It can be easy to utilise on chip embedded multipliers ***inefficiently*** through choice of wordlengths...

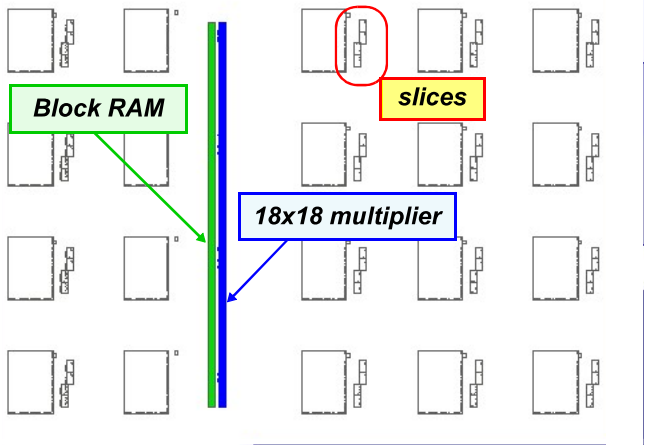


- When using multipliers in System Generator....be careful



Notes:

Looking at a device floorplan, you can clearly see the embedded multipliers, which are located next to Block RAMs on the FPGA in order to support high speed data fetching/writing and computation.



Information on dedicated multipliers taken from “Virtex-II Pro Platform FPGAs: Introduction and Overview”, DS083-1 (v2.4) January 20, 2003. http://www.xilinx.com.

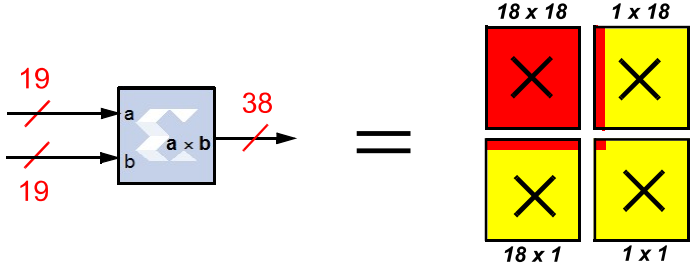
Notes:

If you specify the use of embedded multipliers for a particular multiplier in System Generator, the tool will do exactly as you have asked, and implement it entirely using embedded multipliers. However, depending on the wordlengths involved, this may lead to an inefficient implementation.

The wordlengths of the embedded multipliers are fixed at 18 x 18 bits, and it makes sense to use them as fully as possible.

It is relatively easy to see that a 4 x 4 bit multiply will greatly under use the capabilities of the multiplier, and this particular multiply operation might be better mapped to a distributed implementation, which would leave the embedded multiplier free for use somewhere else. Of course, these decisions are made in the context of some larger design with its own particular needs for the various resources available on the FPGA being targeted.

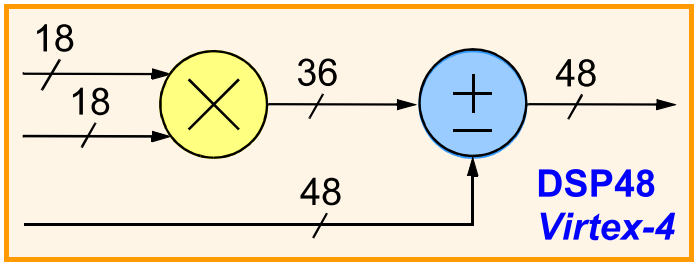
Perhaps less obviously, mapping a multiplication to embedded multipliers where the input operands are slightly longer than 18 bits is also inefficient. This may result in, for example, the following implementation for a requested 19 x 19 bits multiplier, where 4 embedded multipliers are used instead of the expected 1!



High Speed Arithmetic Slices (DSP48s)

5.26

- As much DSP involves the Multiply-Accumulate (MAC) operation, soon after embedded multipliers came DSP48 slices (on the Virtex-4).
- These feature an 18 x 18 bit adder followed by a 48 bit accumulator.

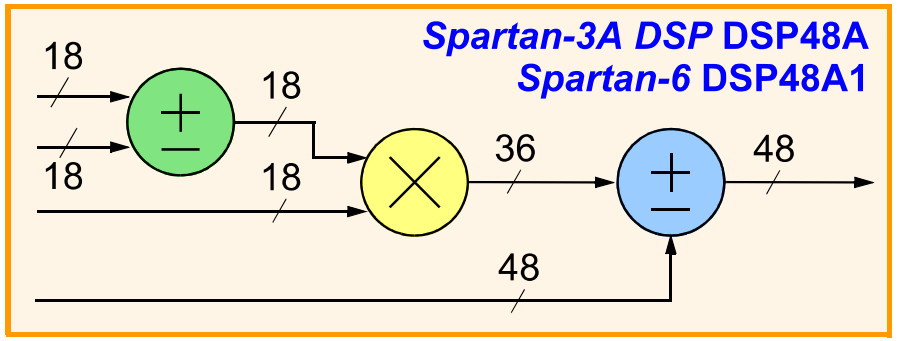


- Like the embedded multipliers, these are low power and fast.
- The ability to cascade slices together also means that whole filters can be constructed without having to use any slices.

DSP48s with Pre-Adders

5.27

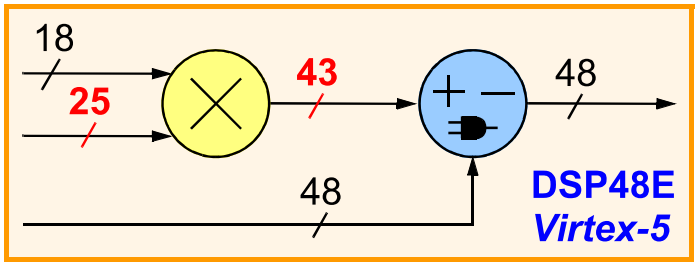
- The ***Spartan-3A DSP*** series and subsequent ***Spartan-6*** feature a version of the DSP48 with a pre-adder, prior to the multiplier.
- This feature is especially useful for DSP structures like ***symmetric filters***, because it allows the total number of multiplications to be reduced.



Notes:

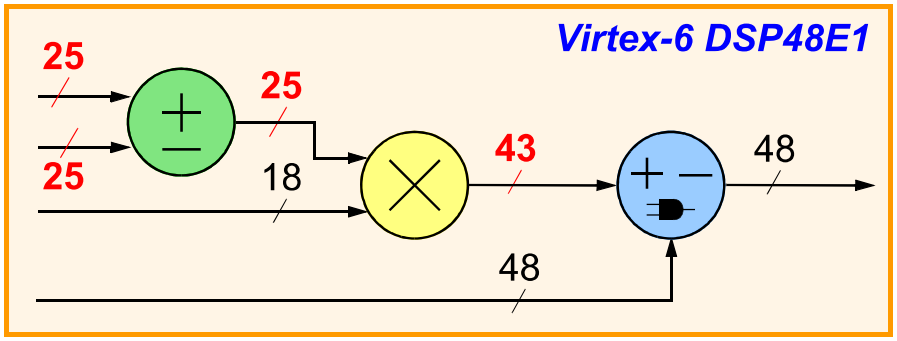
The next series of FPGAs (the Virtex-5) enhanced the capabilities of the DSP slice with the DSP48E.

The major improvements of this slice are logic capabilities within the adder/subtractor unit, and an extended wordlength of one input to 25 bits. The maximum clock frequency also increased in line with the speed of the device.



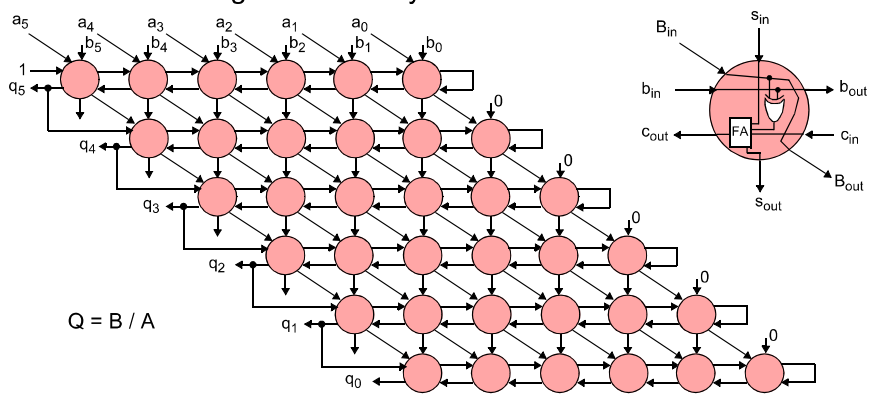
Notes:

The Virtex-6 offers a combination of the benefits of the Virtex-5 (the longer wordlength and arithmetic unit), together with the pre-adder from the Spartan series. This results in a very computationally powerful device, especially as it can be clocked at 600MHz, and the largest chips have 2000+ of them!



Division (i)

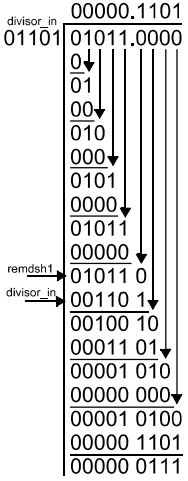
- Divisions are sometimes required in DSP, although not very often.
- 6 bit non-restoring division array:



- Note that each cell can perform either addition **or** subtraction as shown in an earlier slide ⇒ either $S_{in}+B_{in}$ or $S_{in}-B_{in}$ can be selected.

Division (ii)

- There is an alternative way to compute division using another paper and pencil technique.



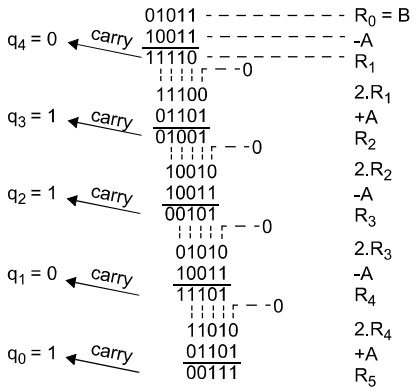
VHDL Design

```
55 architecture divider_section_no_pipeline_arch of divider_section_no_pipeline is
56
57 signal remdsh1 : unsigned(WIDTHDIVISOR downto 0);
58
59 begin
60
61         process(divisor_in,dividend_in,remdsh1,neg_flag_in)
62
63         variable temp   : unsigned(WIDTHDIVISOR downto 0);
64
65         begin
66
67         if std_logic_vector(divisor_in) <= std_logic_vector(remdsh1) then
68
69               temp       := remdsh1(WIDTHDIVISOR downto 0) - divisor_in(WIDTHDIVISOR downto 0);
70               remd_out     <= temp(WIDTHDIVISOR-1 downto 0);
71               dividend_out   <= dividend_in(WIDTHDIVIDEND-2 downto 0) & '1';
72               divisor_out    <= divisor_in;
73
74         else
75
76              remd_out    <= remdsh1(WIDTHDIVISOR-1 downto 0);
77              dividend_out  <= dividend_in(WIDTHDIVIDEND-2 downto 0) & '0';
78              divisor_out   <= divisor_in;
79
80         end if;
81
82         end process;
83
84 -- Shifted remainder.
85 remdsh1 <= remd_in & dividend_in(WIDTHDIVIDEND-1);
86
87 end divider_section_no_pipeline_arch;
```


Notes:

A Direct method of computing division exists. This “paper and pencil” method may look familiar as it is often taught in school. A binary example is given below. Note that each stage computes an addition or subtraction of the divisor A. The quotient is made up of the carry bits from each addition/subtraction. If the quotient bit is a 0, the next computation is an addition, and if it is a 1, the divisor is subtracted. It is not difficult to map this example into the structure shown on the slide.

Example: B = 01011 (11), A = 01101 (13) ⇒ -A = 10011. Compute Q = B / A.



Q = B / A = 01101 x 2^{-4} = 0.8125

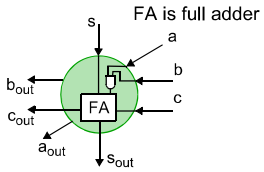
Notes:

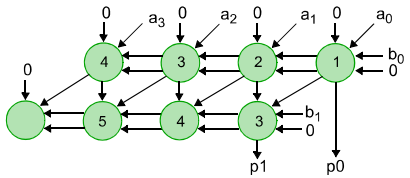
Distributed by: http://www.xilinx.com/univ/
Developed by: www.steepestascent.com

The Problem With Division

5.30

- An important aspect of division is to note that the ***quotient*** is generated ***MSB first*** - unlike multiplication or addition/subtraction!
- This has implications for the rest of the system.
- It is unlikely that the quotient can be passed on to the next stage until ***all the bits*** are computed - hence slowing down the system!
- Also, an *N* by *N* array has another problem - ***ripple through adders***.
- Note that we must wait for ***N full adder delays*** before the next row can begin its calculations.
- Unlike multiplication there is no way around this, and as result division is always ***slower*** than multiplication, even when performed on a parallel array...
    - an *N* by *N* multiply will run faster than an *N* by *N* divide!

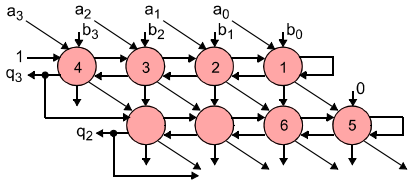


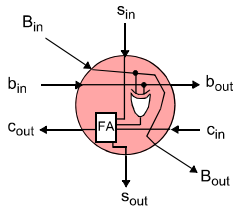


Notes:

By looking at the top two rows of a 4 x 4 division array we can see that the first bit to get generated is the MSB of the quotient. This is unlike the multiplication array that can also be seen below, where the LSB is generated first. This is a problem when using division as most operations require the LSBs to start a computation and hence the whole solution will have to be generated before the next stage can begin.

Another problem for division is the fact that it takes N full adder delays before the next row can start. In the examples below, the order in which the cells can start has been shown. So for the multiplier, the first cell on the second row is the 3rd cell to start working. However, for the divider, the first cell on the second row is only the 5th cell to start working because it has to wait for the 4 cells on the first row to finish.





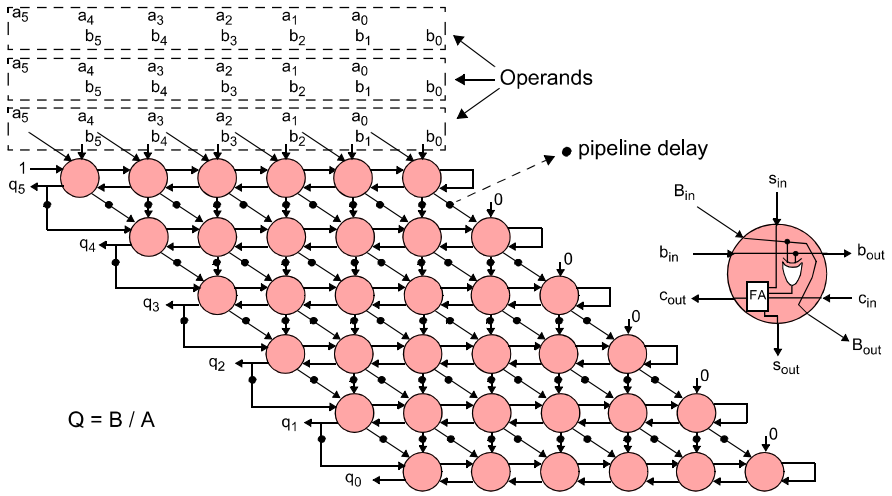
Notes:

Pipelining The Division Array

5.31

- The division array shown earlier can be pipelined to increase

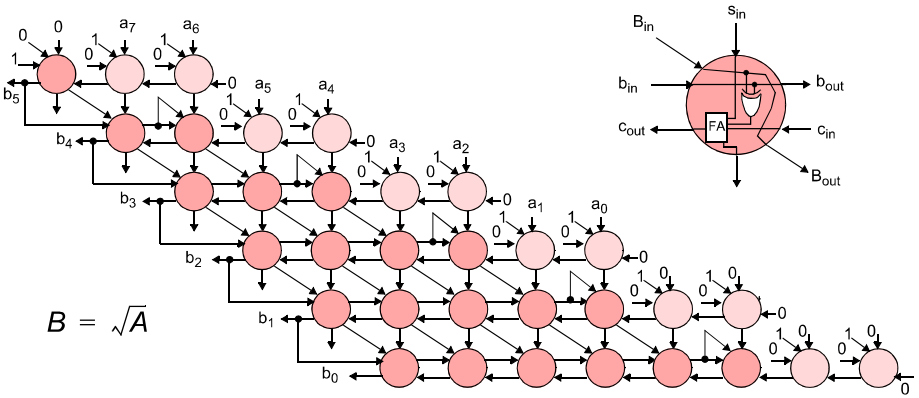
throughput.



Square Root (i)

5.32

- 6 bit non-restoring square root array.

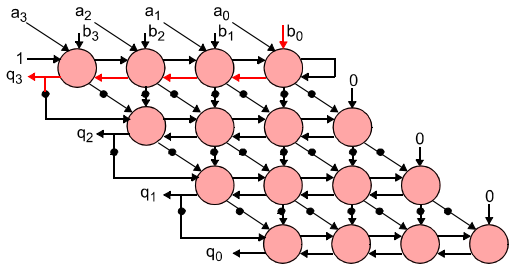


- The square root is found (with divides) in DSP in algorithms such as QR algorithms, vector magnitude calculations and communications constellation rotation.

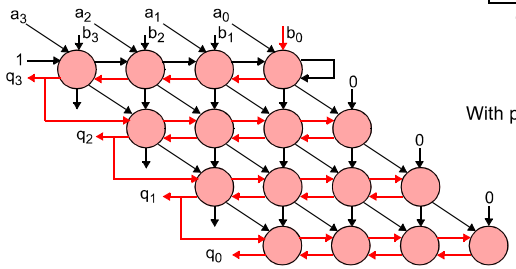
Notes:

To increase the throughput, the critical path can be broken down by implementing pipeline delays at appropriate points. If pipelining is not used, the delay (critical path) from new data arriving to registering the full quotient is N^2 full adders. This delay represents the maximum rate that new data can enter the array. However, by pipelining the array, the critical path is broken down to just N full adders and thus the rate at which new data can arrive is increased dramatically.

The longest path from register to register is the Critical Path.



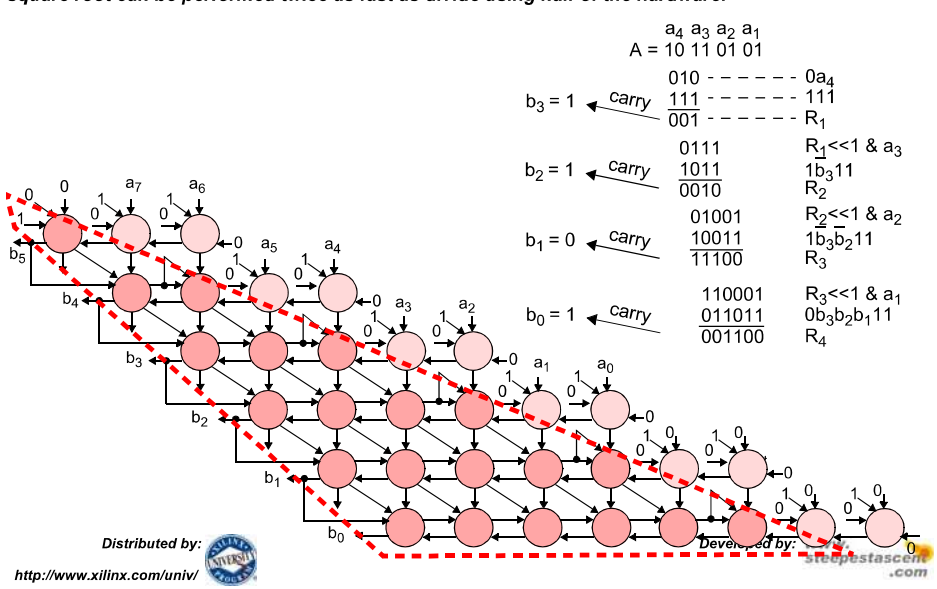
With pipelining the critical path is only N full adders.



Without pipelining the critical path is through N^2 full adders.

Notes:

Looking carefully at the non-restoring square root array, we can note that this array is essentially "half" of the division array! If the division array above is cut diagonally from the left we can see the cells that are needed for the square root array. The 2 extra cells on the right hand side are standard cells which can be simplified. ***So square root can be performed twice as fast as divide using half of the hardware!***



Square Root - An Alternative Approach

- Unfortunately the square root algorithm suffers from the same problems as division, although not to the same extent.
- These are:
  - The result is generated MSB first.
  - Each row has to wait longer and longer for the data it needs from the previous row.
- A solution is to use memory to store the pre-computed square root values. The input is then used as an address to look up the answer.
- This can be fast but if the input wordlength is large this approach quickly becomes unfeasible.
- Another approach is to use memory to look up a partial solution and then use an iterative approach like the Newton-Raphson algorithm to find the final solution.

Notes:

The Newton-Raphson equation can be used to find the square root of a number. It is an iterative technique which can achieve accurate results with relatively few iterations. However, there are two parameters that make it less than ideal for DSP.

- An initial guess is required to start the algorithm and the accuracy of this guess effects the accuracy of the solution after n iterations.
- The number of iterations n to achieve a desired accuracy are unknown.

The iterative algorithm is:

$$x_{n+1} = \left(x_{n+1} + \frac{\text{Input}}{x_n}\right) / 2$$

where x_n is the initial estimate of the square root.

One approach that uses this algorithm is to take the first b MSB bits of the input and use them to address memory containing values for the initial guess x_n. This value is then fed into the Newton-Raphson algorithm for n iterations.

Square Root and Divide - Pythagoras!

- The main appearance of square roots and divides is in advanced adaptive algorithms such as QR using givens rotations.
- For these techniques we often find equations of the form:

$$\cos\theta = \frac{x}{\sqrt{x^2 + y^2}} \quad \text{and} \quad \sin\theta = \frac{y}{\sqrt{x^2 + y^2}}$$

- So in fact we actually have to perform two squares, a divide and a square root. (Note that squaring is “simpler” than multiply!)
- There are a number of iterative techniques that can be used to calculate square root. (However these routines invariably require multiplies and divides and do not converge in a fixed time.)
- There seems to be some misinformation out there about square roots: ***For FPGA implementation square roots are easier and cheaper to implement than divides....!***

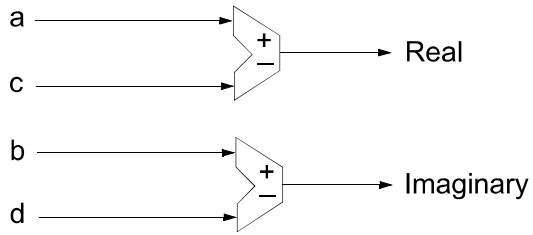
Notes:

Complex Addition/Subtraction

- Complex Addition and Subtraction obey the following:

$$(a+jb)+(c+jd) = (a+c)+j(b+d)$$
$$(a+jb)-(c+jd) = (a-c)+j(b-d)$$

- Thus 2 additions/subtractions are required:



Notes:

For more information on Complex Arithmetic the following text may be useful:

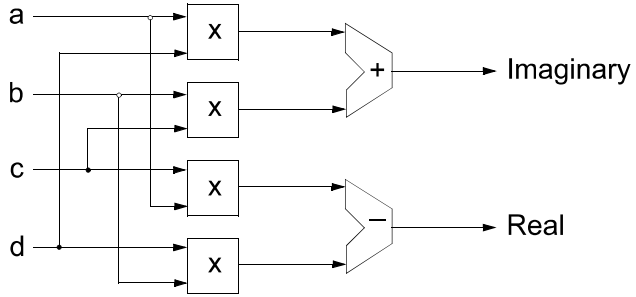
[1] Press, Teukolsky, Vetterling, Flannery. Numerical Recipes in C, Cambridge University Press, 1992

Complex Multiplication

- Complex Multiplication requires more operations:

$$(a+jb)\times(c+jd) = (ac-bd)+j(bc+ad)$$

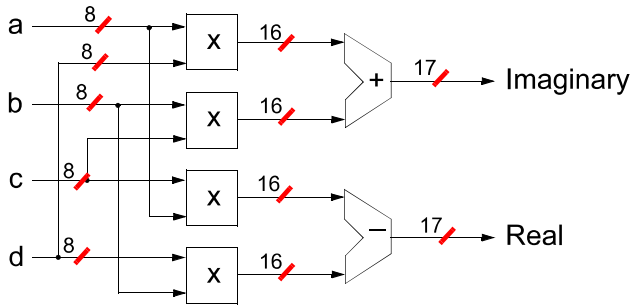
- Thus, 4 multiplications and 2 additions are required:



Notes:

The total number of operations that must be performed for a complex multiplication is 6. But 4 of these operations are multiplies. Generally multiplies are more costly in terms of speed and/or area than additions. Thus, if we can reduce the number of multiplies at the expense of a few more additions, this can be beneficial.

Note the wordlength growth that can occur (using an 8 bit example below):



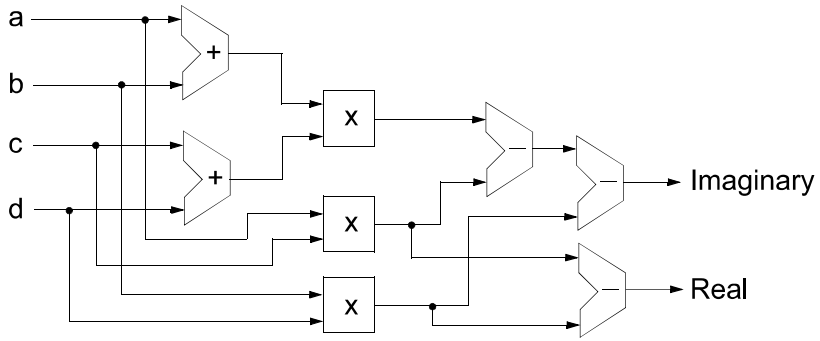
Note that from version 11 of the System Generator tools a complex multiplier block is now available.

Alternative Complex Multiplication

- The multiplication of two complex numbers can also be written as:

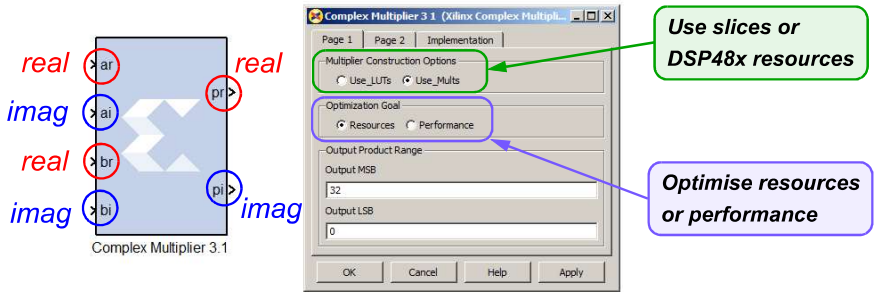
$$(a+jb)\times(c+jd) = (ac-bd)+j[(a+b)\times(c+d)-ac-bd]$$

- Which comprises of 3 multiplications and 5 additions:



Support for Complex Multiplication

- System Generator features a ***Complex Multiplier block***, which saves the effort of building a complex multiplier from first principles!

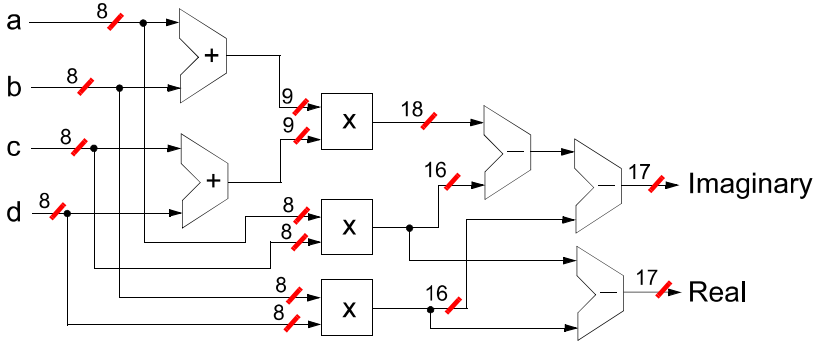


- Depending on whether ***resources*** or ***speed*** are being optimised, the Complex Multiplier block will be implemented using ***3*** or ***4*** multipliers.
- ***Input wordlength*** also influences the number of DSP48xs required.

Notes:

With some algebraic manipulation a complex multiplication can be expressed in terms of 8 operations as opposed to 6. However, even though this form has 2 more operations than the previous one, there is 1 less multiplier. We have effectively substituted a multiplier for 3 additions. This procedure offers an alternative architecture which may be faster in systems where multiplication takes considerably longer than addition.

Note however the implementation cost of the 3 multiply version is not necessarily lower given that one of the multipliers is a 9 bit multiplier and there are of course 5 adds.



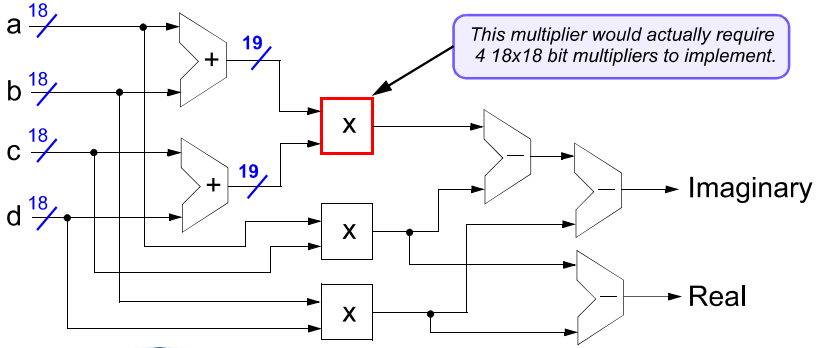
So which would be cheaper in a hardware implementation?

Notes:

When optimising performance, the 4-multiplier option is preferable because it has a shorter critical path than the 3-multiplier architecture, and therefore can be clocked faster.

When optimising resources, usually the 3-multiplier option is chosen. However, if the wordlength grows beyond 18 bits at the input to one of the multipliers, then this will result in a more costly implementation, i.e. the use of 2 or 4 DSP48x slices. (Note that some DSP48x slices have 25 x 18 multipliers, while others have 18 x 18, hence the difference.) Therefore, the 4-multiplier architecture is chosen to optimise resources in some cases.

By referring to the diagram from Slide 5.37, it may be seen that the top multiplier requires more than 1 DSP48x slice, for a, b, c, and d wordlengths of 18 bits. In this case, the 3-multiplier architecture would require 6 DSP48s, for example, instead of 4 DSP48s for the 4-multiplier alternative.

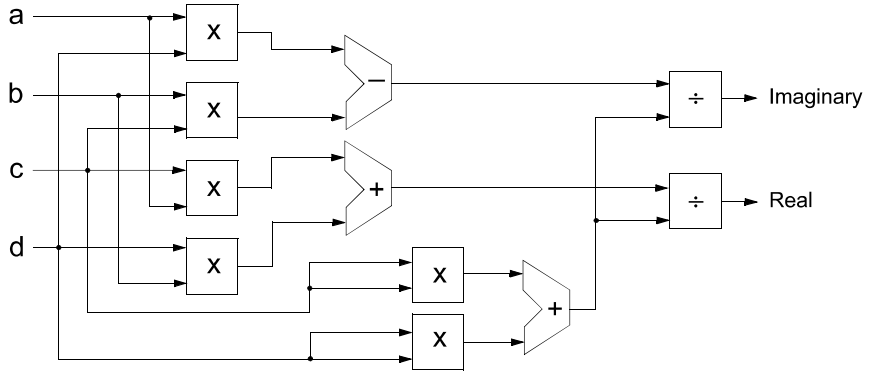


Complex Division

- Division of complex numbers uses more hardware than multiplication:

$$\frac{a+jb}{c+jd} = \frac{(ac+bd)+j(bc-ad)}{c^2+d^2}$$

- Hence, 6 multiplications, 2 divisions and 3 additions are required:



Notes:

Clearly the division of complex numbers is even more expensive than multiplication in terms of the amount of hardware required to carry out the operation. This process requires 6 multiplies, which we already know to be slow, and divides. Dividers however are even slower than multipliers so it is clear that the division of complex numbers is an expensive operation, both in terms of area and speed.

Conclusions

- An overview of the principles of arithmetic for DSP has been given in this section. We have reviewed:
  - The implementation of the following arithmetic operators from first principles:
    adders, ***multipliers***, ***dividers*** and ***square roots***.
  - The costs involved in implementing ***rounding*** and ***saturation***.
  - ***Resources*** used on the FPGA;
  - The various options available for implementing ***multipliers***;
  - The special case of ***constant multiplication***;
  - Multiplication using a ***shift-and-add*** technique
  - ***Complex arithmetic*** operations and their implementations.

Notes: