University of Tehran
Electrical and Computer Engineering Department
Computer Aided Design – Fall 96
CA4: Fixed point numbers

Problem Definition

In this CA, we refine the design of the previous CA by replacing the real data type with fixed-point type. This way, all weights and input vectors of a neuron should be defined as 16-bit fixed point numbers. In this format, the integer part has 10 bits and the fractional part has 6 bits. To represent negative numbers, use the most significant bit of the integer part to keep sign (9 bits integer part and one bit sign). Here, you should define the fixed-point type and implement multiplication and addition operations for this type in VHDL.

Note that all datapath components should now be 16 bits.

** Fill weight and input vectors with fixed-point arbitrary numbers that range between -1 to +1
** The number of neuron inputs must be parameterized. Set it to a sample integer value when you are implementing the design.
** Assume one MAC operation (one multiply and one add) can be completed in a single cycle.

Report

You have to submit a report with your code. The report must contain how you have implemented the MAC unit (adder+multiplier). When you describe each unit, you must show in which part of your code (file+line) each required description comes. For example, when you talk about the multiplier, you should clearly describe the architecture of the unit, inputs, output, how the sign is handled, how the overflow is detected, and so on. Then say, for example, the architecture of multiplier comes in file *x.vhd*, line *10*, etc.

Fixed point numbers

Recall that a binary number:

110101_2

represents the value:

$1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0* 2^1 + 1 * 2^0$

$= 32 + 16 + 4 + 1$

$= 53_{10}$

Now, if we divide the number 53 by 2, we know the result should be 26.5. However, how do we represent it if we only had integer representations?

The key to represent fractional numbers, like 26.5 above, is the concept of *binary point*. A binary point is like the decimal point in a decimal system. It acts as a divider between the integer and the fractional part of a number.

In a decimal system, a decimal point denotes the position in a numeral that the coefficient should multiply by $10^0 = 1$. For example, in the numeral 26.5, the coefficient 6 has a weight of $10^0 = 1$. But what happen to the 5 to the right of decimal point? We know from our experience, that it carries a weight of 10^{-1}. We know the numeral "26.5" represents the value "twenty six and a half" because

$2 * 10^1 + 6 * 10^0 + 5 * 10^{-1} = 26.5$

The very same concept of decimal point can be applied to our binary representation, making a "binary point". As in the decimal system, a binary point represents the coefficient of the term $2^0 = 1$. All digits (or bits) to the left of the binary point carries a weight of 2^0, 2^1, 2^2, and so on. Digits (or bits) on the right of binary point carries a weight of 2^{-1}, 2^{-2}, 2^{-3}, and so on. For example, the number:

11010.1_2

represents the value:

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
...	1	1	0	1	0	1	0	...

$= 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0* 2^0 + 1 * 2^{-1}$

$= 16 + 8 + 2 + 0.5$

$= 26.5$

A careful reader should now realize the bit pattern of 53 and 26.5 is exactly the same. The *only* difference, is the position of binary point. In the case of 53_{10}, there is "no" binary point. Alternatively, we can say the binary point is located at the far right, at position 0. (Think in decimal, 53 and 53.0 represents the same number.)

2^5	2^4	2^3	2^2	2^1	2^0	Binary	2^{-1}	2^{-2}	2^{-3}

						Point			
1	1	0	1	0	1	.	0	0	0

In the case of 26.5_{10}, binary point is located one position to the *left* of 53_{10}:

2^5	2^4	2^3	2^2	2^1	2^0	Binary Point	2^{-1}	2^{-2}	2^{-3}
0	1	1	0	1	0	.	1	0	0

Now, recall in class, we discuss shifting an integer to the right by 1 bit position is equivalent to dividing the number by 2. In the case of integer, since we don't have a fractional part, we simply cannot represent digit to the right of a binary point, making this shifting process an *integer division*. However, it is simply a limitation of integer representations of binary number.

In general, mathematically, given a fixed binary point position, shifting the bit pattern of a number to the right by 1 bit *always* divide the number by 2. Similarly, shifting a number to the left by 1 bit multiplies the number by 2.

Fixed Point Number Representation

The shifting process above is the key to understand fixed point number representation. To represent a real number in computers (or any hardware in general), we can define a fixed point number type simply by implicitly *fixing* the binary point to be at some position of a numeral. We will then simply adhere to this implicit convention when we represent numbers.

To define a fixed point type conceptually, all we need are two parameters: width of the number representation, and binary point position within the number

We will use the notation `fixed<w,b>` for the rest of this article, where `w` denotes the number of bits used as a whole (the Width of a number), and `b` denotes the number of the fractional part, or in the other words, the position of binary point counting from the least significant bit (counting from 0).

For example, `fixed<8,3>` denotes an 8-bit fixed point number, of which 3 right most bits are fractional. Therefore, the bit pattern:

0	0	0	1	0	.	1	1	0

represents a real number:

00010.110_2

$= 1 * 2^{1} + 1 * 2^{-1} + 1 * 2^{-1}$

$= 2 + 0.5 + 0.25$

$= 2.75$

Note that on a computer, a bit pattern can represents anything. Therefore the same bit pattern, if we "cast" it to another type, such as a `fixed<8,5>` type, will represents the number:

000.10110_2

$= 1 * 2^{-1} + 1 * 2^{-3} + 1 * 2^{-4}$

$= 0.5 + 0.125 + 0.0625$

$= 0.6875$

If we treat this bit pattern as integer, it represents the number:

10110_2

$= 1 * 2^{4} + 1 * 2^{2} + 1 * 2^{1}$

$= 16 + 4 + 2$

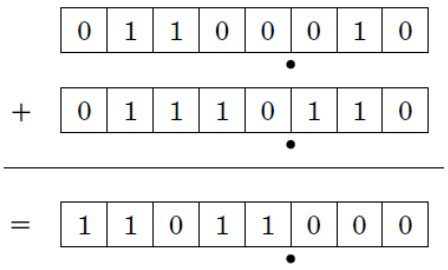
$= 22$

This representation has limited precision and we have to round a number to the nearest representable number. For example in `fixed<8,3>`, 2.75 and 2.76 and 2.70 all have the same representation 00010.1102.

Fixed Point Addition/subtraction

Fixed point arithmetic works just like ordinary binary arithmetic. The hardware elements are unchanged. You only have to care where your decimal point (.) is placed after the calculation (add, Multiply,...).

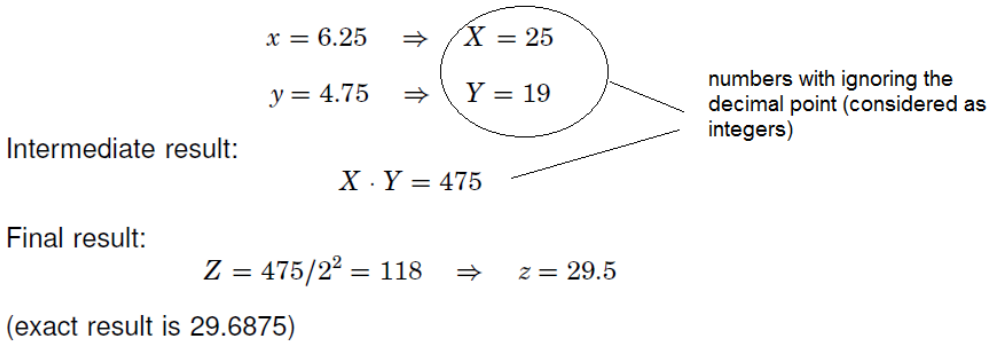
Two fixed-point numbers in the same `fixed<w,b>` format can be added or subtracted directly. The result will have the same number of fractional bits, but we should take care of overflow as the operation may need w+1bits.Example: 12.25+14.75=27

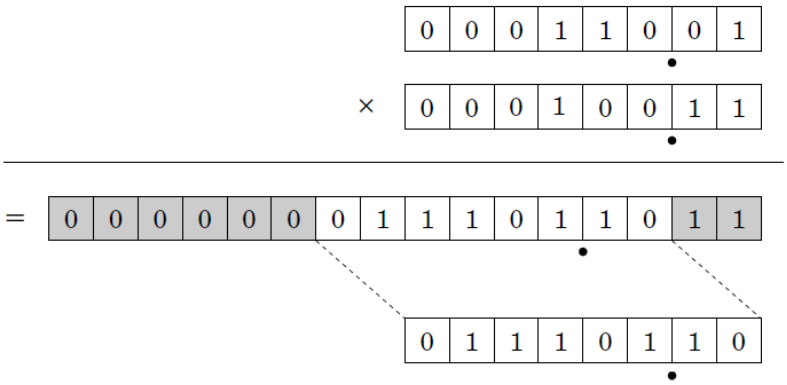


Fixed Point Multiplication

Two fixed-point numbers in the same `fixed<w,b>` format can be multiplied by considering them as integers and performing the same multiplication as an integer. The results should then be divided by 2^b and the decimal point be inserted at its position (b).

This way, double word length is needed for the intermediate result. After dividing by 2^b , we remove some MSBs to get w bit numbers. Division by 2^b is implemented as a right-shift by b bits. Example: 6.25×4.75=29.5





Negative numbers

For negative numbers we have to add another bit to keep the sign. You can keep negative numbers as two's complement. Sign and magnitude is another simple way of representing negative integers. The far left bit is called the sign bit, and is made a zero for positive numbers, and a one for negative numbers. The other bits are a standard binary representation of the absolute value of the number. In this case, you can perform the operations as usual but use the sign bits of the operands to determine the sign of the results.