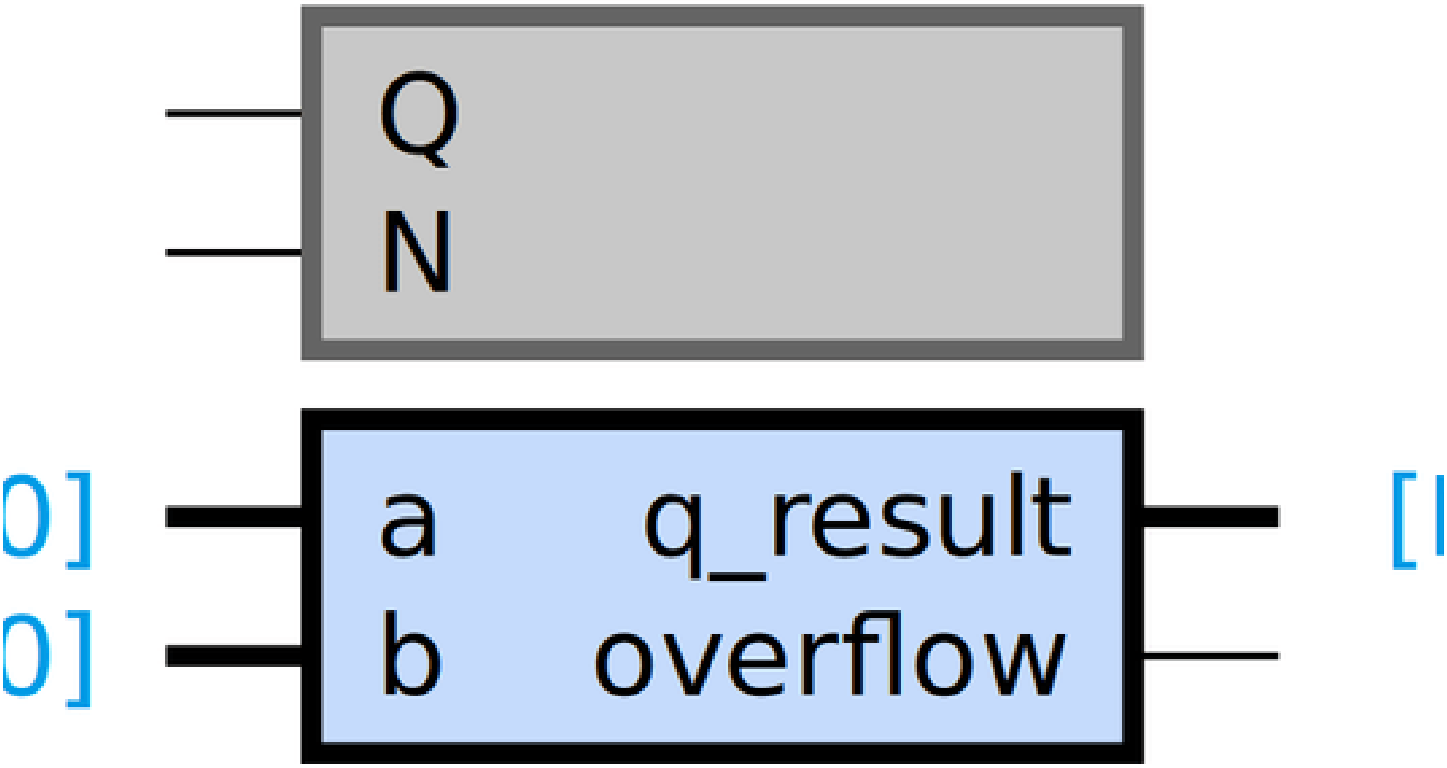
nt arithmetic to your design



ere we implement an FPGA based Convolutional Neural Network accelerator. However, the content of this article is useful for understanding the 'Fixed-po
u can understand the general principle and use the code even if you aren't interested in the entire series of articles.

n

ithmetic to your design

mating the Verification Process

neural network in hardware.

ber representation system?

n real life inputs and data, we need it to be able to interpret the various formats humans use to interpret this data. Real world data has everything from sig :s. However, when we work with hardware, all we see is registers and wires which are simply variables capable of holding a sequence of bits. All we get to · parameter involved in defining these variables. As the author of this excellently written whitepaper on fixed point arithmetic puts it

› meaning inherent in a binary word, although most people are tempted to think of them (at first glance, anyway) as positive integers. **However, the meani ion**, i.e., on the representation set and the mapping we choose to use."

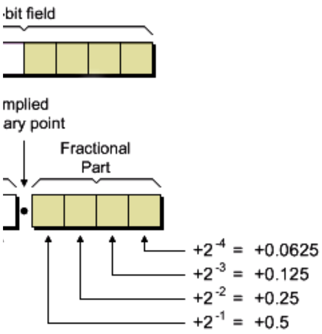
ı-point are simply that, they're representation sets and mapping schemes that have been widely used and standardized by the entire community. You coul ıse it to interpret these binary words.

›ut in the open that can teach you about the basics of representing numbers on computers and the differences between fixed-point and floating-point nur

t of fixed-point of floating-point numbers. It just recognizes fields with certain bit widths. This means that we need a layer above the hardware to help us (ıt the same time help us interpret the results that the hardware is giving us by converting the raw binary words into the chosen format.

ıg python mainly because of its simplicity and code readability.

/hich the position of the decimal point remains fixed independent of the value that number is representing. This is what makes fixed point numbers easier son to the floating point numbers. Fixed point arithmetic also uses much less resources in comparison to floating point arithmetic. Of course all of this cc s of precision for a particular bit-width than fixed-point arithmetic. We take a hit in terms of the quantization noise when we use numbers in which the bin sented by that number.



ting a fixed-point number

ıplement fixed-point representation** wherein, as show in in the above image, the first bit represents the sign bit and if the number is negative (sign bit = 1) nindful of this when dealing with negative numbers.

›erly denote the various parameters of a fixed point number. The most popular one is probably the **A(a,b)** format wherein **a** is the number of bits used to re ; used to represent the fractional portion of the number. Which means that the total number of bits used to store an A(a,b) fixed point number is **N = a + b**

ɪ based on the *range* and *precision* that you need for the problem at hand. For example, if all or most of the numbers you're trying to represent have a very n use less bits to represent the integral portion and use more bits to represent the fractional portion thus achieving greater precision in your arithmetic. T ›lication where your hardware will be used.

xed-point numbers, go through this document.

---

he fixed point arithmetic in Verilog, let's create a golden model i.e a model of the target functionality that gives use the correct target outputs which we ca and fix any bugs based on the wrong outputs.

w us to convert numbers between the standard floating-point representation that python uses and the fixed point representation that we want our hardwa

at converts a number in the standard *float* format to a user specified format. The variable *integer_precision* represents the number of bits to be used to re ıt number and the variable *fraction_precision* denotes the number of bits used for the fractional part (after the decimal point).

ıble hereon after, it represents the 'float' data type in python. Apparently, it is stored in the form of a 32-bit floating point number by the CPU under the h

```
ger_precision,fraction_precision):
                                              #sign bit is 1 for negative numbers in 2's complement representation




(integer_precision) + 'b'
ıat(int(num),precision)
num - int(num)

ion_precision):
ıart_f*2
f = d -int(d)
append(int(d))
.join(str(e) for e in fractional_part)

ın_bit) + twos_comp(integral_part + fraction_string,integer_precision,fraction_precision)

ın_bit) + integral_part+fraction_string
```

ed by this code is another function that we use to generate the 2's compliment of any binary number. Here is the code:

```
r_precision,fraction_precision):
r(1-int(x))for x in val)
nteger_precision+fraction_precision) + 'b'
((int(flipped,2)+1),length)
```

ıat we've built:

```

110110110', 3, 12))

LOAT NUMBER TO FIXED-POINT
3,12))

LOAT NUMBER TO FIXED-POINT
,3,12))
```

ıta from our fixed-point representation to a human readable *float* variable:

```
r_precision,fraction_precision):       #s = input binary string
```

```
n - 1

 twos_comp((s[1:]),integer_precision,fraction_precision)

 s[1:]
recision + fraction_precision -1):
complemented[j])*(2**i)




er
```

working:

```
01001001110',3,12))

10111111011',3,12))
```

tion, let's see what happens when we nest these functions together!

```
to_fp(-2.729,3,12),3,12))
```

How come the value is different when converted back and forth between fixed-point and *float* representation?

sion of the fixed-point representation in action. When we store a *float* variable in python, the CPU actually stores it in a 32-bit floating point number in its n
e loose a teeny tiny bit of precision as the fixed-point representation does not have enough bits to achieve the same precision as the 32-bit floating point

---

r fixed-point arithmetic

nodel ready, let's begin coding in Verilog!

odified versions of the fixed-point arithmetic library from Opencores.

```
sign-bit + 3 integer-bits + 12 fractional-bits = 16 total-bits
```

```
|III|FFFFFFFFFFFF|
,F) format would be A(3,12)

negative number in it's 2's complement form by default, all we
e two numbers together (note that to subtract a binary number
its two's complement)



 your system (the software/testbench feeding this hadrware with
y negative numbers in their 2's complement form,(some people
nitude as it is and make the sign bit '1' to represent negatives)
a look at the fixed point arithmetic modules at opencores linked
```

```
sign-bit + 3 integer-bits + 12 fractional-bits = 16 total-bits
|III|FFFFFFFFFFFF|
,F) format would be A(3,12)

es





:0] a,
:0] b,
:0] q_result,    //output quantized to same number of bits as the input
flow             //signal to indicate output greater than the range of our format


ssumption, here, is that both fixed-point values are of the same length (N,Q)
 the results will be of length N+N = 2N bits
fies the hand-back of results, as the binimal point
n the same location

sult;       //  Multiplication by 2 values of N bits requires a
            //  register that is N+N = 2N deep
plicand;
iplier;
mp, b_2cmp;
tized_result,quantized_result_2cmp;

-1],{(N-1){1'b1}} - a[N-2:0]+ 1'b1};  //2's complement of a
-1],{(N-1){1'b1}} - b[N-2:0]+ 1'b1};  //2's complement of b

= (a[N-1]) ? a_2cmp : a;
= (b[N-1]) ? b_2cmp : b;

 = a[N-1]^b[N-1];                      //Sign bit of output would be XOR or input sign bits
ltiplicand[N-2:0] * multiplier[N-2:0]; //We remove the sign bit for multiplication
ult = f_result[N-2+Q:Q];               //Quantization of output to required number of bits
ult_2cmp = {(N-1){1'b1}} - quantized_result[N-2:0] + 1'b1;  //2's complement of quantized_result
0] = (q_result[N-1]) ? quantized_result_2cmp : quantized_result; //If the result is negative, we return a 2's co
                                                                 //of the output value
_result[2*N-2:N-1+Q] > 0) ? 1'b1 : 1'b0;
```

hold the result of the multiplication between two **N - bit** numbers right? Right.

vidth as the multiploer and multiplicand?

ı that needs to be done at some point in the process in order to preserve the width of our data-path. If we keep increasing the size of the output at every n ȷe datapath. This quantization adds something called the *quantization-noise* to our data since we're truncating the 2*N bit output to N bits. We should kee truncation process. This is true most of the time if the numbers are pretty small and within the dynamic range of the fixed-point representation.

test benches can be found at the Github Repo

ı Source on 23 July, 2020.

Batman

I'm **Batman**, a silent nerd and a watchful engineer obsessed with great Technology. Get in touch via the **Discord community** for this site