OPL3 analyzed

Steffen Ohrendorf

March 2, 2014

Preface

This document aims to be an in-depth analysis of Yamaha's OPL3 chip, to help software developers, mathematicians and electrical engineers understand how this piece of hardware works.

The biggest task the author had to face was to find a way so that all three groups mentioned would be satisfied. Fortunately, the author is both a mathematician and a software developer, so it wasn't a big problem to find the common denominator for these groups of people; on the other hand, the author does only have basic knowledge about electrical engineering, so there's probably one or the other point where the author simply does things in a non-standard way.

Contents

1 Conventions **1**
1.1 Basic operations . . . 2
1.2 Binary notation . . . 2
1.3 Calculus . . . 4

2 Sine wave **4**
2.1 Data storage . . . 4
2.2 Logarithmic/exponential identity . . . 4
2.3 Calculating the lookup tables . . . 6
2.4 Putting it all together . . . 8

1 Conventions

As many things in this document are concerning single bits or selections of bits, a special notation for the incorporated math will be introduced. You should study these definitions closely, so that you understand what's going on.

Here will be no introduction to the basic boolean algebra—if you're very confused after reading this section, read it again. Then, if you're less confused, read it again.

Keep in mind that everything here is about discrete mathematics. If you don't get it, read a book about this topic, and then come back later.

1.1 Basic operations

We will start with some basic things, namely addition and subtraction of numbers. But before we can start, here's an important fact: the OPL3 uses *1's complement* to represent numbers, which makes negation (and thus, subtraction) easy to implement in an integrated circuit. The trade-off of this is that a negation isn't mathematically correct, and thus subtraction isn't either.

This section will be fairly intriguing for the novice reader, but it is important to understand the implications of using the 1's complement arithmetics.

We will start by defining a new operator $\ominus$ for negation, which means "negate using 1's complement":

$$\begin{aligned}\ominus x &:= -x - 1\\ x \ominus y &:= x - y - 1\end{aligned}$$

You can easily prove that negating a number x consisting of n bits can be implemented by swapping all bits of x, i. e.:

$$\ominus x \equiv 2^n - 1 - x$$

(The binary representation of $2^n - 1$ is simply n ones: $2^n - 1 = \sum_{i=0}^{n-1} 2^i$.)

Taking this idea further, a negation can be implemented using the `XOR` operation with $2^n - 1$; electrical engineers can implement this by `NOT`ing all n data lines, and software developers can use the `NOT` operation of their computer if n can be represented as $w \cdot 2^k$, where w is the basic word size of their computer.

1.2 Binary notation

It will be quite convenient to know how many bits a variable can hold, and it will also be very convenient to extract single bits or ranges of bits from such a variable.

First, lets define a simple notation for marking the amount of bits a variable can occupy. For example, if we had a variable x which could hold up to 16 bits, we would write

$$x_{[16]}$$

Now, let's define some easy-to-remember notation for extracting single bits or ranges of bits.

Single bit To access a single bit n of a variable x, we will write

$$x_{\langle n\rangle} := \lfloor x/2^n \rfloor \bmod 2 \in \{0; 1\}$$

Lower bits To extract the n least significant bits of a number x, i. e. the bits $0 \ldots n-1$ of x, we will write

$$x_{\langle n|} := x \bmod 2^n \in \{0, 1, \ldots, 2^n - 1\}$$

(Please pay some attention to the $\langle n|$, where the "$|$" marks the right end of the binary notation of x, and the "$\langle$" should be read like "there could be more bits, but they are of no interest.")

Upper bits Analogous to extracting the lower bits, we will write the following for skipping over the n least significant bits of a number x:

$$x_{|n\rangle} := \lfloor x/2^n \rfloor$$

(As before, the "$\rangle$" should be seen as an "attention stopper", and "$|$" is the left edge of all available bit positions in x.)

Range of bits For extracting a range $m \ldots n$ of bits from a number x, we will write

$$\begin{aligned} x_{\langle m;n\rangle} &:= x_{|m\rangle}{}_{\langle n-m+1|} \\ &= \lfloor x/2^m \rfloor_{\langle n-m+1|} \\ &= \lfloor x/2^m \rfloor \bmod 2^{(n-m+1)} \end{aligned}$$

(You can see that $x_{\langle n\rangle} \equiv x_{\langle n;n\rangle}$.)

We now come to a notation for hexadecimal numbers. This could have been done earlier, but it wouldn't have been of great use, if any; the notation is probably the easiest one to remember within this document. They will be printed in typewriter font, using a \$ as suffix, e. g. $\texttt{1F}_\$$.

Let's summarize this by taking $\texttt{EB}_\$ = (1110\,1011)_2$ apart:

$$\begin{aligned} \texttt{EB}_{\$\langle 1\rangle} &= 1 \\ \texttt{EB}_{\$\langle 2\rangle} &= 0 \\ \texttt{EB}_{\$\langle 4|} &= (1011)_2 \\ \texttt{EB}_{\$|4\rangle} &= (1110)_2 \\ \texttt{EB}_{\$\langle 2;5\rangle} &= (1010)_2 \end{aligned}$$

Because numbers usually don't have a maximum, it's also nice to see that $\texttt{EB}_{\$\langle k\rangle} = \texttt{EB}_{\$|k\rangle} = 0 \; \forall k \geq 9$. In other words, if you go beyond the left border of any binary number, you'll find nothing except zeros. For most of the mathematical (or semi-mathematical) operations following, this means that we usually don't have to care about how large a number really is, as the operations are "justifying" themselves to a proper size.

(This is a "white lie", though. Consider, e. g., the example of swapping bits by using the `NOT` operation as a possible implementation of negation: here you will need to know the actual size of the number you want to negate. Fortunately, this "optimization" is in fact a function $f(x, n) = x \,\texttt{XOR}\, (2^n - 1)$, which is only defined if $x < 2^n$; we will see later that we can—with only a little effort—deduce a minimum n in every case.)

Applications of the notation will be found throughout the document.

1.3 Calculus

Please note that—if not specified otherwise—all arithmetics in this document will be in $\mathbb{N}$ and fractional results of expressions will be rounded towards zero. Unfortunately, this makes calculations sensitive to the order of execution; thus let's define the order of calculation to be from left to right if ambiguous, e. g., $a \cdot b \cdot \frac{c \cdot d}{e \cdot f}$ will be calculated as $(a \cdot b)\big((c \cdot d)/(e \cdot f)\big)$.

(This is just a matter of numerical analysis, which tries to apply infinite principles—such as integration of non-discrete functions or arithmetics with arbitrary precision—on problems where the algorithm only allows for finite principles, such as computers with their inability to safe arbitrarily large numbers or to do arithmetics with very large numbers in a reasonable time.)

Now that we have the easy tools at hand, we can use them to trim a non-discrete problem down to a discrete problem.

2 Sine wave

2.1 Data storage

According to the "OPLx decapsulated" document by Matthew Gambrell and Olli Niemitalo, the OPL3 contains a ROM with the first quarter of a sine wave.[1] This raising quarter of the wave is then mirrored in a combination of two ways to produce the full sine wave.

(Be aware that this was probably the most inaccurate description of the problem arising below, because: (a) we will encounter many logarithms and exponents, (b) we will only look at the first quarter, as mirroring needs comparatively much less brain power, and (c) I'm pretty good at giving inaccurate descriptions.)

2.2 Logarithmic/exponential identity

> **Hold on, Lone Wayfarer!** This section contains lots of wild $f()$rmulas! $\log_a$rithms… e^xponential functions… and i𝔇entity equations, to name fεw of them. Each one more dangerous than the other, and not many people have `return`ed from this forest called ∀lgebra.
>
> Those who have returned, however, became insane from the things they have seen. I'd rather `NOT` walk into there…

The basic idea behind the OPL3 synthesizer approach is to use the exponential identity:

$$e^{\ln(x)} = x \mid x > 0$$

(The $x > 0$ is necessary because $\ln(x)$ is only defined for $x > 0$.)

We can use this to transform $\sin(x)$ to an equivalent form, producing the first quarter of the sine wave:

$$\sin(x) = e^{\ln\big(\sin(x)\big)} \mid 0 < x \leq \frac{\pi}{2}$$

[1] `https://docs.google.com/document/d/18IGx18NQY_Q1PJVZ-bHywao9bhsDoAqoIn1rIm42nwo/edit`

As the powers of the exponential functions can quickly get very lengthy, here's yet another notation for them. You probably won't find this notation anywhere else in the world; it's only for better readability here. If you actually see it somewhere else, it has pretty sure a completely different meaning.

$$a^{b+c} \cdot d \quad \equiv \quad a \Lsh (b+c) \cdot d$$

(You see that the new $\Lsh$ operator has a higher precedence than a multiplication; this matches the default behavior in most programming languages like Python (`2**3*3`), Octave/Matlab (`2^3*3`) or R, where both notations are allowed.)

The complicated formula presented right before this little excursion allows us for some nice trick:

$$\begin{aligned} \alpha \cdot \sin(x) &= e \Lsh \ln\bigl(\alpha \cdot \sin(x)\bigr) \\ &= e \Lsh \Bigl(\ln(\alpha) + \ln\bigl(\sin(x)\bigr)\Bigr) \end{aligned}$$

In other words, if α has some value > 0, we can scale $\sin(x)$ up and down by reducing the multiplication to an *addition*. It looks tedious to do so every time, but "fortunately" the OPL accepts only scaling factors that are already in their logarithmic form.

The main problem we have to solve now is to turn the non-discrete formula above to something working with integers. To achieve this, let's try to replace x with something that's discrete. We want to put a value p ranging from 0 to, say, $256 = 2^8$ into the formula to get the same output as when we put x ranging from 0 to $\pi/2$ into it. Remember that $\pi/2$ isn't included in the allowed values, so 256 isn't either.

(These ranges aren't quite accurate: remember that we cannot put 0 into the logarithm. If you calculate $\sin(0)$, you'll get 0, which isn't in the allowed range for $\ln(x)$. But assume for the moment that it's possible, as we will cope with that problem later. Additionally, 2^8 isn't a randomly chosen value; it's indeed used in the real chip.)

It's easy to find the solution:

$$\begin{aligned} \sin(x) &= \sin\Bigl(\frac{p}{256} \cdot \frac{\pi}{2}\Bigr) \\ \Rightarrow x &= \frac{p}{256} \cdot \frac{\pi}{2} \end{aligned}$$

We handle the problem that $\sin(x) = 0$ if $x = 0$, which turns out to invalidate the $\ln\bigl(\sin(x)\bigr)$, by adding a very small offset to p, so we get the final formula:

$$\alpha \cdot \sin\Bigl(\frac{p+0.5}{256} \cdot \frac{\pi}{2}\Bigr) = e \Lsh \Biggl(\ln(\alpha) + \ln\biggl(\sin\Bigl(\frac{p+0.5}{256} \cdot \frac{\pi}{2}\Bigr)\biggr)\Biggr)$$

(An offset of 0.5 isn't a great problem, because we only shift the function slightly to the left with it. Also, 0.5 is a decision that excludes the 0 from the argument of $\sin(x)$, while still leaving out the upper border, i. e. $255 + 0.5 < 256$. If you think about what happens when we mirror the quarter sine wave horizontally to reconstruct the full sine wave, you will see that from the right border of the first quarter to the left border of the next one is exactly one unit (of x) apart, which is just perfect.)

2.3 Calculating the lookup tables

Before we continue our journey, let's quickly change the base of our formula from e to 2:

$$\alpha \cdot \sin\Big(\frac{p+0.5}{256} \cdot \frac{\pi}{2}\Big) = 2 \Lsh \bigg(\log_2(\alpha) + \log_2\Big(\sin\Big(\frac{p+0.5}{256} \cdot \frac{\pi}{2}\Big)\Big)\bigg)$$

We can now tear this intriguing monster apart into two smaller pieces: one for calculating the logarithm, and one for calculating the exponential part. Let's call them $\mathrm{bsl}^*(x)$ and $\mathrm{bsp}^*(x)$, for the "binary sine logarithm" and the "binary sine power", which we will use to derive yet another formula:

$$\begin{aligned} \mathrm{bsl}^*(p) &:= \log_2\Big(\sin\Big(\frac{p+0.5}{256} \cdot \frac{\pi}{2}\Big)\Big) \\ \mathrm{bsp}^*(x) &:= 2 \Lsh x \end{aligned}$$

If we look closer at it, we can see that $\mathrm{bsl}^*(p)$ is always ≤ 0. More precisely, if we check the range of the results of $\mathrm{bsl}^*(p)$ for every $0 \leq p < 256$, we see that $-8.35 \lessapprox \mathrm{bsl}^*(p) \lessapprox -6.79 \cdot 10^{-6}$. We now have two problems: (a) these numbers are too small to be simply rounded to integers, and (b) negative numbers aren't very nice for computation. We can solve this by negating the function and multiplying it with 2^8, and then apply the Gaussian brackets to denote the rounding step:

$$\begin{aligned} \mathrm{bsl}^{**}(p) &:= \bigg[-256 \cdot \log_2\Big(\sin\Big(\frac{p+0.5}{256} \cdot \frac{\pi}{2}\Big)\Big)\bigg] \\ \mathrm{bsp}^{**}(x) &:= 2 \Lsh \frac{x}{-256} \end{aligned}$$

Our function $\mathrm{bsl}^{**}(p)$ now has a range $0 \leq \mathrm{bsl}^{**}(p) \leq 2137$.

(The rounding introduces an absolute error of at most 0.5; if we take this into the $\mathrm{bsp}^{**}(x)$ function, i. e. $\max\big(2 \Lsh \frac{x \pm 0.5}{-256} - 2 \Lsh \frac{x}{-256}\big) \mid 0 \leq x < 256$, we get a maximum absolute error of about $1.35 \cdot 10^{-3}$, which is less than $1/7$ of 1% relative difference to an exact calculation.)

With this, we can now create the first piece of code to generate the first lookup table:

Algorithm 1 Sine logarithm calculation

```
var sinLogLookup: array[0...255] of $\mathrm{int}_{[12]}$

func[12] bsl(p):= -- This is in fact the $\mathrm{bsl}^{**}(p)$ function
    return round( -256 * log2( sin( (0.5+p)/256 + $\pi$/2 ) ) )

func[$\varnothing$] generateFirstLookupTable():=
    for i ← 0 to 255:
        sinLogLookup[i] ← bsl(i)
```

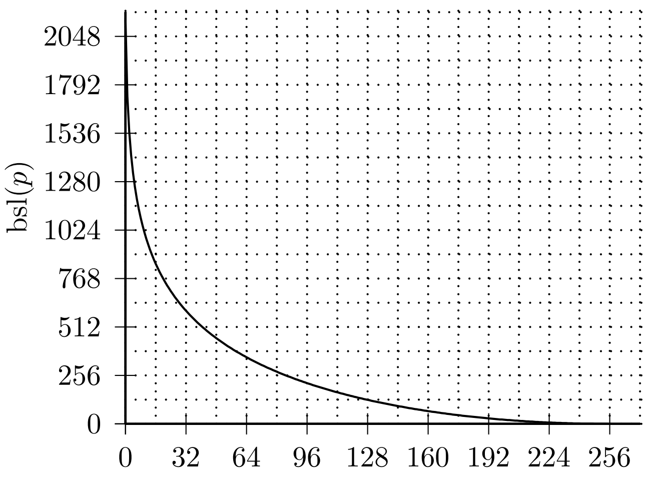

Figure 1: Logarithmic sine table

A plot of the generated values can be found in Figure 1.

Our next task is to "discretize" the $\text{bsp}^{**}(x) = 2 \Lsh \frac{x}{-256}$ function. At first, we will transform the function a bit, so that we can create a lookup table from it consisting of 256 entries. Fortunately, we can again use some tricks which let us reconstruct the full function for every $x \geq 0$ from some basic values $0 \leq x < 256$:

$$\begin{aligned} 2 \Lsh \frac{x}{-256} &= 2 \Lsh \frac{-(x_{\langle 8|} + 2^8 \cdot x_{|8\rangle})}{2^8} \\ &= 2 \Lsh \left(\frac{-x_{\langle 8|}}{2^8} - x_{|8\rangle} \right) \\ &= 2 \Lsh \left(\frac{-x_{\langle 8|}}{2^8} \right) \Big/ \, 2 \Lsh x_{|8\rangle} \end{aligned}$$

You see that the division we introduced in the transformation can be implemented easily with a bit-shift to the right by $x_{|8\rangle}$ bits, so we only need to calculate the dividend of the new formula and can reconstruct everything else from it; but we're not completely done with it yet.

This is because our dividend is a non-integer function with values between 1 and 2. As a first step, we subtract the offset 1 from it; then, we multiply the whole thing with 2^{10} and round it to an integer:

$$\left[2 \Lsh \left(\left(\frac{-x_{\langle 8|}}{2^8} \right) - 1 \right) \cdot 2^{10} \right] = \left[2 \Lsh \left(10 - \frac{x_{\langle 8|}}{2^8} \right) - 2^{10} \right]$$

Et voilà, we have our second lookup table:

Algorithm 2 Sine exponential calculation

```
var sinExpLookup: array[0...255] of $\text{int}_{[10]}$

func[10] bsp(n):= -- This isn't the actual bsp function, neither is it the dividend of it.
  return round( $2 \Lsh (10 - n/2^8) - 2^{10}$ )

func[∅] generateSecondLookupTable():=
  for i ← 0 to 255:
    sinExpLookup[i] ← bsp(i)
```

You can see a plot of the generated values in Figure 2.

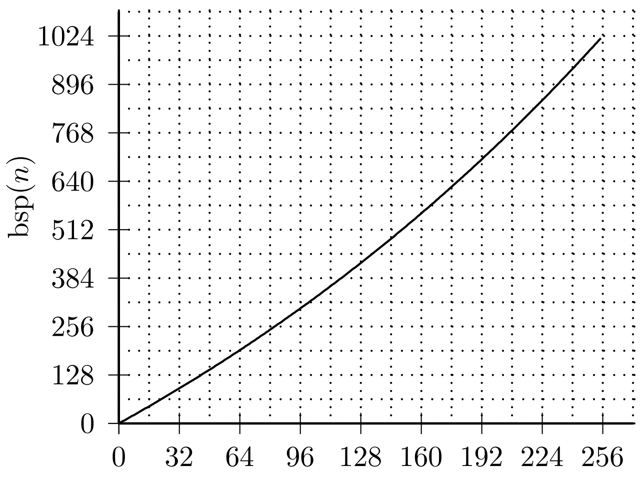

Figure 2: Exponential sine table

2.4 Putting it all together

Now, let's say we want to calculate the actual sine wave value at a position p. At first, we need to look up our logarithmic value in our pre-calculated table, taking into account that we only have the first quarter calculated; thus, we have to "mirror" p between the first and second quarter as well as between the third and fourth quarter. What we get is this function:

$$\text{bsl}(p) := \begin{cases} \text{sinLogLookup}[p_{\langle 8|}] & \text{if } p_{\langle 8\rangle} = 0 \\ \text{sinLogLookup}[\ominus p_{\langle 8|}] & \text{if } p_{\langle 8\rangle} = 1 \end{cases}$$

(We can map any p to be $0 \leq p < 2^{10}$, because a quarter of the sine wave we can produce has a length of 256 samples: $2^8 \cdot 4 = 2^{10}$. Furthermore, we need to mirror p in the second and fourth quarter, i. e., whenever $2^8 \leq p < 2^8 + 2^8$ or $2^8 + 2^9 \leq p < 2^{10}$. If you look at it, you will see that $p_{\langle 8 \rangle}$ is only 1 in exactly these ranges.)

Now, let's deal with the exponential part, as we have everything at hand now:

$$\mathrm{bsp}(n) := \begin{cases} (\mathrm{sinExpLookup}[\ominus n_{\langle 8|}] + 2^{10}) \cdot 2/2 \Lsh n_{|8\rangle} & \text{if } x_{\langle 9 \rangle} = 0 \\ \ominus(\mathrm{sinExpLookup}[\ominus n_{\langle 8|}] + 2^{10}) \cdot 2/2 \Lsh n_{|8\rangle} & \text{if } x_{\langle 9 \rangle} = 1 \end{cases}$$

To calculate a sine wave with a period of 1024 and an "attenuator" α, we can finally say:

$$\mathrm{oplSin}(p, \alpha) := \mathrm{bsp}(\mathrm{bsl}(p) + \alpha)$$