OPL3 analyzed

Steffen Ohrendorf

October 31, 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

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.

> ☢ Additionally, you will find paragraphs like this, which start with a familiar warning sign and are typeset in a smaller font; you should read those paragraphs whenever you are left with some questions after the preceding paragraphs or want to understand some core concepts better, as they explain some things in more detail and may answer some of the questions.

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":

$$\ominus x := -x - 1$$
$$x \ominus y := x - y - 1$$

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 "⟨" 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 "⟩" 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 i∂entity 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. . .

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

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:

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

As the powers of the exponential functions can quickly get very lengthy, we will use KNUTH's first-order *up-arrow notation*:

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

☢ You see that the up-arrow 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:

$$\begin{aligned} \alpha \cdot \sin(x) &= e \uparrow \ln\big(\alpha \cdot \sin(x)\big) \\ &= e \uparrow \Big(\ln(\alpha) + \ln\big(\sin(x)\big)\Big) \end{aligned}$$

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. We could have chosen any number 2^n; the higher n would be, the higher the accuracy of the sine wave would be.

It's easy to find the solution:

$$\begin{aligned} \sin(x) &= \sin\left(\frac{p}{2^8} \cdot \frac{\pi}{2}\right) \\ \Rightarrow x &= \frac{p}{2^8} \cdot \frac{\pi}{2} \end{aligned}$$

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:

$$\alpha \cdot \sin\Bigl(\frac{p + 0.5}{2^8} \cdot \frac{\pi}{2}\Bigr) = e \uparrow \biggl(\ln(\alpha) + \ln\Bigl(\sin\Bigl(\frac{p + 0.5}{2^8} \cdot \frac{\pi}{2}\Bigr)\Bigr)\biggr)$$

☢ An offset of 0.5, which slightly shifts the function to the left, solves a simple problem, because 0.5 is a number that excludes the 0 from the argument of $\sin(x)$, while still leaving out the upper border, i. e. $2^8 - 1 + 0.5 = 2^8 - 0.5 < 2^8$. 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. Just remember that $\sin'(\pi/2) = 0$, which is exactly the outcome of this shift.

2.3 Calculating the lookup tables

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

$$\alpha \cdot \sin\Bigl(\frac{p + 0.5}{2^8} \cdot \frac{\pi}{2}\Bigr) = 2 \uparrow \biggl(\log_2(\alpha) + \log_2\Bigl(\sin\Bigl(\frac{p + 0.5}{2^8} \cdot \frac{\pi}{2}\Bigr)\Bigr)\biggr)$$

This will allow us to apply "binary magic" later on.

We can now tear this intriguing monster apart into two smaller pieces: one for calculating the logarithm (which is used to build one lookup table), and one for calculating the exponential part (for the other lookup table). Let's start with calculating the logarithmic lookup table. A careful reader would have already noticed that $-8.35 \lessapprox \log_2((p + 0.5)/2^8 \cdot \pi/2) \lessapprox -6.79 \cdot 10^{-6}$ for $0 \leq p \leq 255$; this is an efficiency problem, as floating point (or even fixed point) numbers are pretty ineffective when they are used in hardware calculations. For this reason, we scale the values by -2^8 and round them to integers, which results in values between 0 and 2137, inclusive.

☢ Again, we could have used any number -2^n for scaling, with the valid statement for precision as mentioned earlier.

We now can define a "binary sine logarithm" as it is indeed used to generate the first lookup table stored in the chip's ROM:

$$\mathrm{bsl}(p) := \biggl[-2^8 \cdot \log_2\Bigl(\sin\Bigl(\frac{p + 0.5}{2^8} \cdot \frac{\pi}{2}\Bigr)\Bigr)\biggr]$$

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

We can now create a premature "binary sine power" which uses the result of this

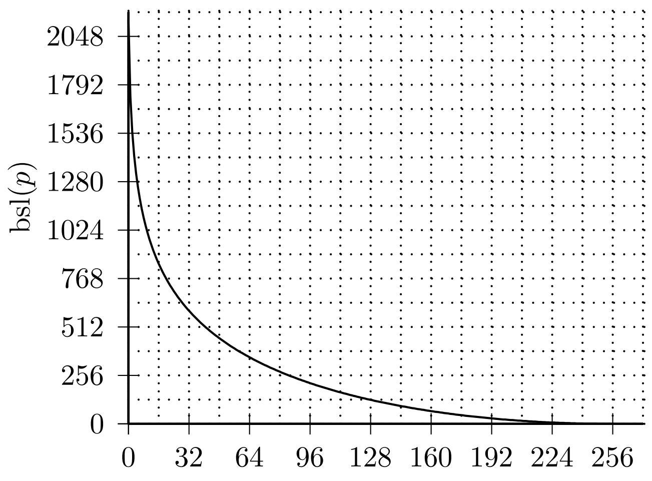

Figure 1: Logarithmic sine table

function and finally take first advantage of "binary magic":

$$\begin{aligned}
\mathrm{bsp}^*(x) &:= 2 \uparrow \frac{x}{-2^8} \\
&= 2 \uparrow \frac{-(x_{\langle 8|} + 2^8 \cdot x_{|8\rangle})}{2^8} && \text{(split } x\text{)} \\
&= 2 \uparrow \left(\frac{-x_{\langle 8|}}{2^8} - x_{|8\rangle} \right) && \text{(reduce fraction)} \\
&= 2 \uparrow \left(\frac{-x_{\langle 8|}}{2^8} \right) \Big/ \, 2 \uparrow x_{|8\rangle} && \text{(allow bit-shift by } x_{|8\rangle}\text{)}
\end{aligned}$$

But still, we have the problem that $0.5 \leq 2 \uparrow (-x_{\langle 8|}/2^8) < 1$; additionally, we have a mathematical (aka inefficient) negation of x. By using a *double negation*, we can get rid of the minus sign, and simply replace it by a 1's complement negation: if we replace the minus sign in the exponent with $\ominus$, we effectively get $255 - x_{\langle 8|}$ as the nominator, and if we also use $\ominus$ on x *before* we put it into $\mathrm{bsp}^*(x)$, we have an identity, because $\ominus(\ominus x) = x$.

☢ Assuming that x has n bits, the identity is practically $2^n - 1 - (2^n - 1 - x) = 2^n - 1 - 2^n + 1 + x = x$; see the definition of $\ominus$.

After getting rid of the minus sign, we will now leave out the "right shift", and create a new, more matured version of $\mathrm{bsp}^*(x)$:

$$\mathrm{bsp}^{**}(x_{[8]}) := 2 \uparrow \frac{x}{2^8}$$

We see that $1 \leq \text{bsp}^{**}(x_{[8]}) < 2$, and can now handle the problem of the non-integer values more easily. At first, we subtract the "base offset" 1 (or, for fans of the IEEE 754 standard, "hidden bit") from our grown function to reduce the need for too much data storage, and then scale it up by 2^{10}; we then round it again, and have our final, matured version used for generating the second chip's lookup table:

$$\text{bsp}(x) := \left[\left(2 \uparrow \frac{x}{2^8} - 1\right) \cdot 2^{10}\right]$$

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

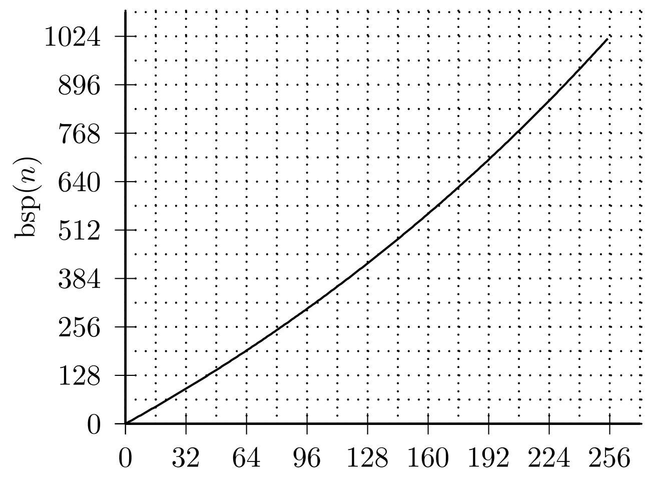

Figure 2: Exponential sine table

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, and create a "real" periodic function, the "sine logarithm", having in mind that every second quarter of a sine wave is mirrored on its x axis:

$$\text{sl}(p_{[10]}) := p_{\langle 9\rangle} \cdot 2^{15} + \begin{cases} \text{bsl}(p_{\langle 8|}) & \text{if } p_{\langle 8\rangle} = 0 \\ \text{bsl}(\ominus p_{\langle 8|}) & \text{if } p_{\langle 8\rangle} = 1 \end{cases}$$

☢ We return the ninth bit to keep the information ready whether to negate the final values; thus, let's call it the "sign bit".

Now, let's create a real "sine power", by inverting the tricks applied earlier:

$$\operatorname{sp}(x) := \begin{cases} (\operatorname{bsp}(\ominus x_{\langle 9|}) + 2^{10}) \cdot 2/2 \uparrow x_{|8\rangle} & \text{if } x_{\langle 15\rangle} = 0 \\ \ominus(\operatorname{bsp}(\ominus x_{\langle 9|}) + 2^{10}) \cdot 2/2 \uparrow x_{|8\rangle} & \text{if } x_{\langle 15\rangle} = 1 \end{cases}$$

☢ Here, we produce our final results for output; see the usage of our "sign bit". Remember that every second half of a sine wave is negated.

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

$$\operatorname{osin}(p_{[10]}, \alpha_{[14]}) := \operatorname{sp}(\operatorname{sl}(p) + \alpha)$$

☢ To keep our sign bit untouched, we need to make sure that $(\alpha + 2137) < 2^{15} \iff \alpha < 2^{15} - 2137$; because $2^{12} > 2137$, we can tighten the requirement to $\alpha \leq 2^{15} - 2^{12}$ and deduce that α must be $\leq 2^{14}$, because $2^{14} \leq 2^{15} - 2^{12} \iff 2^{14} + 2^{12} \leq 2^{15}$ is always true.

If we look at this function, we see that the maximum (or amplitude) is $\operatorname{osin}(255, 0) = \operatorname{sp}(\operatorname{sl}(255)) = \operatorname{sp}(\operatorname{bsl}(255)) = \operatorname{sp}(0) = \operatorname{bsp}(\ominus 0) + 2^{10}) \cdot 2 = (1018 + 1024) \cdot 2 = 4084$, so we need 13 bits to store that value and its sign bit. But according to 1's complement, the minimum is $-4084 - 1 = -4085$, and, which is sometimes undesired, our $\operatorname{osin}(p, \alpha)$ function produces for sufficient large α *always* the values 0 and -1.

3 Envelopes

After this quite mathematical part, we will discuss the envelope generator in more detail. The envelope generator basically automates the sine wave's amplification, namely in three to four phases, called attack, decay, sustain, and release, where the sustain phase can be skipped. The attack phase controls the amplification when wave generation starts; consider e. g. a trombone, where the brass needs some "warm up time" until it begins to vibrate with its resonance frequency. Also, the trombone is a good example for the decay phase: the musician has to blow a bit stronger at first to reduce the warm up time, so the trombone will sound a bit louder for a short time, until this instrument reaches its sustain phase, i. e. the time the musician finally plays the note he wants to play. After holding the tone for a short time, the musician stops blowing into the mouth piece, but the brass still vibrates a bit longer, which is due to the mentioned resonance; this is the release time.

Another physical effect comes into play: the energy density. Heavily simplified, a sonic wave has an energy density proportional to $\sin^2(2\pi f \cdot t)$, where f is the frequency of the wave, and t is the time we look at it. Now, if we want to know how much energy we need to hold any frequency f for one time unit, we have to calculate

$$\begin{aligned} E &= \int_0^1 \sin^2(2\pi f \cdot t)\,\mathrm{d}t \\ &= \Big[t/2 - \frac{1}{8\pi f}\sin(4\pi f \cdot t)\Big]_0^1 \\ &= \Big(1/2 - \frac{1}{8\pi f}\sin(4\pi f)\Big) - \Big(0 - \frac{1}{8\pi f}\sin(0)\Big) \\ &= 1/2 - \frac{1}{8\pi f}\sin(4\pi f) > 0 \end{aligned}$$

If we try to find a more simplified form without the sine, we come to $\hat{E} \sim 4\pi - 1/f$ (for $1/f \leq 4\pi$), which lets us conclude two things: (1) if we want to play high frequencies as loud as low ones, we have to use more energy, or (2) if our energy is constant, higher frequencies decrease in their amplitude. A similar statement holds for the times of attack, decay and release times, as the instruments lose their "vibration energy" faster with higher frequencies.

☢ You might think that the energy becomes negative if f were small enough, but fortunately $\lim_{x\to 0}\sin(x)/2x = 0$.

The OPL chip supports this effect in a simple way called the "key scale rate" for the ADR phases as well as the "key scale level" for the $1/f$ part.

3.1 Effective Rates

Hold on! This section contains not yet fully understood content, although the formulas given here match the effects observed from the chip exactly. If you have notes or ideas (or even formulas) explaining what happens here, don't hesitate to contact the author!

We start with discussing the calculation of the ADR timings; for this, we need to know how the OPL calculates its "effective speed" using the "rates" it gets supplied from the user. The user may specify a rate ranging from 0 to 15, where 15 is the fastest; then, the key scale rate (which we will call KSR from now on) as well as a "note type selection" (NTS) flag are used to calculate the effective rate ranging from 0 to 63. As stated earlier, the physical model in the OPL is very simple; for example, the rate adjustments cannot be disabled, and they also are very rough.

The OPL uses something called a "rate offset", so that the effective rate $\hat{r}$ is calculated as $\hat{r} = 4r + \Delta r$; this offset Δr depends on the frequency $f_{[10]}$ and the "block" $b_{[3]}$, the NTS flag (also called "keyboard split method") for fine-tuning and the KSR flag for rough-tuning as well.

☢ The final frequency is calculated as $2^b f$.

The general outline is as follows: (1) if we have a rate r of zero, we also have an effective rate $\hat{r} = 0$; (2) if the KSR flag is not set, we only use the upper 2 bits of b;

(3) otherwise, we use the full three bits of b and either the eighth or the ninth bit of f, depending on the NTS flag.

> ☢ This can be deduced from the YMF262 manual. Additional information about the formula for $\hat{r}$ has been retrieved by testing a real chip and information about the MSX Audio Chip, which has some similarities with the OPL3.

$$\Delta r = \begin{cases} 0 & \text{if } r = 0 \\ b_{\langle 2\rangle} & \text{if KSR=0} \\ 2b + f_{\langle 9\rangle} & \text{if NTS=0} \\ 2b + f_{\langle 8\rangle} & \text{if NTS=1} \end{cases}$$

The effective rate $\hat{r} = 4r + \Delta r$ is then used to calculate a counter increment i:

$$i = \min\bigl\{(\hat{r}_{\langle 2|} + 4) \cdot 2 \uparrow \hat{r}_{|2\rangle};\ 4 \cdot 2^{15}\bigr\}$$

> ☢ The formula for i has been deduced from the timing information in the MSX Audio Chip manual by trial-and-error. The counter $c_{[15]}$ is part of the "envelope generator" and is used to determine when (and how much) to change the envelope value α. Also note that the maximum for i is $4 \cdot 2^{15}$, not $7 \cdot 2^{15}$; this is evident if you look at the timings in the Y8950 manual and it has also been proven by testing a real OPL3 chip.

Let's take $\hat{r}$ apart, so that we can calculate i more easily and eventually get a better understanding of the things that happen here:

$$\hat{r}_b = 4 + \begin{cases} 0 & \text{if } r = 0 \\ b_{\langle 2\rangle} & \text{if KSR=0} \\ 2b_{\langle 0\rangle} + f_{\langle 9\rangle} & \text{if NTS=0} \\ 2b_{\langle 0\rangle} + f_{\langle 8\rangle} & \text{if NTS=1} \end{cases}$$

$$\hat{r}_e = \begin{cases} r & \text{if KSR=0} \\ r + b_{|1\rangle} & \text{otherwise} \end{cases}$$

$$i = \min\bigl\{\hat{r}_b \cdot 2 \uparrow \hat{r}_e;\ 7 \cdot 2^{15}\bigr\}$$

The increment i is now added to the counter $c_{[15]}$ on every output sample. As soon as c overflows (i. e., $\hat{c} = c_{|15\rangle} > 0$), this overflow $\hat{c}$ is added to the envelope value $\tau_{[9]}$ in the decay and release phases; in the attack phase, we have to use a different calculation: $\tau' = \tau \ominus \lfloor \tau\hat{c}/8 \rfloor$.

> ☢ We need to check that $\hat{c} > 0$ because $\ominus 0 = 1$, which would let the attack phase work even in the case $r = 0$; on the other hand, we always change τ, even in the case $\lfloor \tau\hat{c}/8 \rfloor = 0$, because of the special subtraction. Also, τ is a "saturating value": as soon as it becomes < 0 or > 511, it gets clamped to these values so that no overflow occurs.

We can now compare these formulas to the timings given in the Y8950 manual; for this purpose, we do a "full attack" and a "full release", ranging from full silence (96 dB) to full loudness (0 dB) and measure the time.

(All times in ms, OPL times from the formulas in the text.)

$\hat{r}$	MSX Attack	MSX Decay	OPL Attack	OPL Decay
59	0,20	2,74	0,20	2,94
58	0,24	3,20	0,24	3,44
57	0,30	3,84	0,30	4,12
56	0,38	4,80	0,38	5,15
55	0,42	5,48	0,42	5,87
54	0,46	6,40	0,48	6,86
53	0,56	7,68	0,58	8,23
52	0,70	9,60	0,70	10,28
51	0,80	10,96	0,80	11,75
50	0,92	12,80	0,95	13,72
49	1,12	15,36	1,13	16,45
48	1,40	19,20	1,41	20,55
47	1,56	21,92	1,61	23,49
46	1,84	25,56	1,89	27,41
45	2,20	30,68	2,25	32,90
44	2,76	38,36	2,82	41,11
43	3,12	43,84	3,22	46,98
42	3,68	51,12	3,76	54,82
41	4,40	61,36	4,51	65,79
40	5,52	76,72	5,63	82,22
39	6,24	87,68	6,44	93,96
38	7,36	102,24	7,52	109,63
37	8,80	122,72	9,01	131,55
36	11,04	153,44	11,26	164,43
35	12,48	175,36	12,87	187,92
34	14,72	204,48	15,02	219,26
33	17,60	245,44	18,02	263,10
32	22,08	306,88	22,53	328,87
31	24,96	350,72	25,74	375,85
30	29,44	408,96	30,05	438,50
29	35,20	490,88	36,04	526,21
28	44,16	613,76	45,05	657,74
27	49,92	701,44	51,49	751,70
26	58,88	817,92	60,07	877,00
25	70,40	981,76	72,08	1052,39
24	88,32	1227,52	90,10	1315,47
23	99,84	1402,88	102,97	1503,40
22	117,76	1635,84	120,15	1753,97
21	140,80	1963,52	144,16	2104,76
20	176,64	2455,04	180,20	2630,95
19	199,68	2805,76	205,95	3006,80
18	235,52	3271,68	240,28	3507,94
17	281,60	3927,04	288,32	4209,52
16	353,28	4910,08	360,40	5261,90
15	399,36	5611,52	411,89	6013,60
14	471,04	6543,36	480,55	7015,87
13	563,20	7854,08	576,65	8419,05
12	706,56	9820,16	720,81	10523,79
11	798,72	11223,04	823,78	12027,19
10	942,08	13086,72	961,08	14031,74
9	1126,40	15708,16	1153,29	16838,08
8	1413,12	19640,32	1441,62	21047,58
7	1597,44	22446,08	1647,56	24054,38
6	1884,16	26173,44	1922,17	28063,45
5	2252,80	31416,32	2306,58	33676,14
4	2826,24	39280,64	2883,23	42095,17

The average relative errors are about 7% for the attack and 2% for the decay times; the variance of the errors is at most 0.01%.