A VLSI ARCHITECTURE FOR SOUND SYNTHESIS

John Wawrzynek

and

Carver Mead

Computer Science
California Institute of Technology

5158:TR:84

A VLSI ARCHITECTURE FOR SOUND SYNTHESIS

by John Wawrzynek and Carver Mead

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125, U.S.A.
818-356-4858

October 10, 1984

5158:TR:84

This work was supported by the System Development Foundation.

1 Introduction

Sounds that come from physical sources are naturally represented by differential equations in time. Since there is a straight-forward correspondence between differential equation in time and finite difference equations, we can model musical instruments as simultaneous finite difference equations. Musical sounds can be produced by solving the difference equations that model instruments in real time.

The computational bandwidth that is needed to compute musical sounds is enormous. For the sampled waveform representation of sound, we need to produce samples at a rate of about 50K samples/sec. If we assume that there are about 100 computational operations per sample for each voice, that is 5 million operations per second per voice. An operation involves a multiplication and an addition. By a voice we mean one horn or one string of a string instrument. A mid-size computer of today is capable of about only 250,000 arithmetic operations per second which means by our model, it is only capable of computing about 1/20 of a single voice. When the data-shuffling and housekeeping operations necessary to run a real instrument model are included, the factor increases another order of magnitude – so it is hopeless to compute the sounds in real time. Today's most powerful computers are capable of computing only a small number of voices.

Even the new concurrent machines do not hold much promise. These machines, sometimes called homogeneous machines, fail to support the generation of sound because they are built with a fixed interconnection between their processors. In order to map a problem like musical sound generation onto such a machine, the processors must be programmed to provide the communication between various parts of the model. This results in the machine spending much of its time shuffling data around.

In the past the enormous computation bandwidth of sound generation has been avoided by using musical shortcuts such as waveform table lookup and interpolation. While this approach and those built upon it can produce pleasing musical sounds, the attacks, dynamics, continuity, and other properties of real instruments simply cannot be captured. In addition, traditional methods suffer from the shortcoming that the player of the instrument is given parameters that don't necessarily have any direct physical interpretation and are just artifacts of the model. It would be nice, for example, to supply a musician or composer with a string instrument with string whose mass, stiffness and tension can be varied dynamically. This capability is possible if a

representation of the instrument is based on its physics.

An even larger problem with the shortcut methods of the past is that they have produced models that require updates of internal parameters at a rate that is many times that which occurs in real musical instruments. The control, or update, of parameters has become an unmanagable problem.

A natural architecture for solving finite difference equations is one with an interconnection matrix between processors that can be reconfigured (or programmed), as illustrated in figure 1. A realization of a new instrument involves a reconfiguring of the connection matrix between the processing elements along with configuring connections to the outside world both for control and updates of parameters.

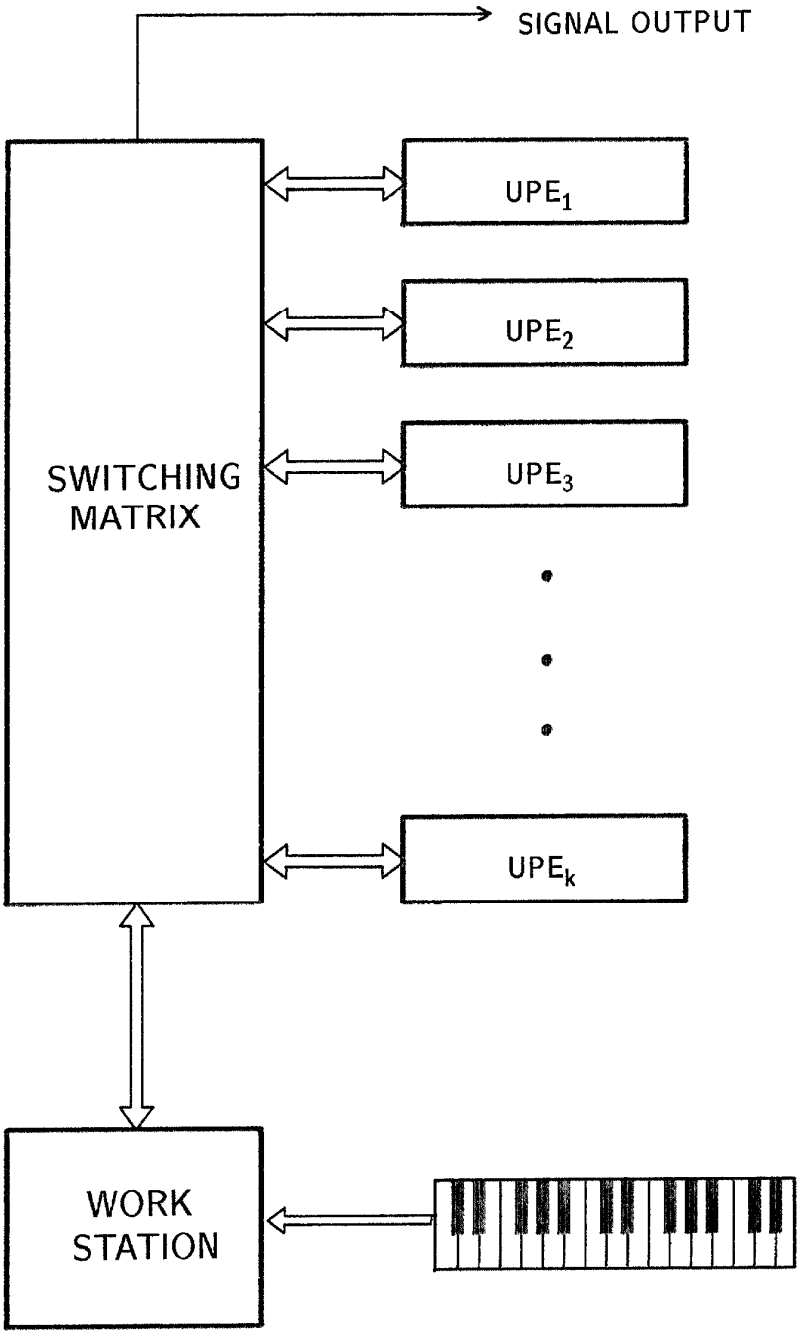


Figure 1: Sound Synthesis Architecture

Processing elements are placed together to form an array and then joined by a reconfigurable interconnection matrix. A general purpose computer supplies updates of parameters to the processing elements and provides an interface to the player of the instrument. The external computer also supplies the bit patterns for the interconnection matrix. Synthesized signal outputs go to a digital to analog converter.

In order to implement a reconfigurable connection matrix, a bit serial representation of samples facilitates the use of single wire connections between computational units, drastically reducing the complexity of implementation. In fact, a bit serial implementation makes the entire approach possible.

Bit serial implementations also have the advantage that computational elements are very small and have inexpensive realizations. One potential drawback with bit serial systems is that they must run at a clock rate that is higher then that of their parallel counterparts. In our implementations, even with 64 bit samples, the bit clock rate is only 3 MHz, which is far below the limits of current IC technology.

For our basic unit of computation we have chosen a unit we call a UPE (Universal Processing Element) [1] which computes the function:

$$A + B \times M + D \times (1 - M) \tag{1}$$

It is very similar to the two's complement bit serial multipliers proposed by R. F. Lyon [2]. In its simplest mode of computation, where D=0, the function of a UPE is a multiplication and an addition. This simple element forms a digital integrator that is the basic building block for solving linear difference equations. If D is not set to 0, the output of the UPE is the linear interpolation between B and D where M is the constant of interpolation. Interpolation is important in sound synthesis in particular for mixing signals.

All the inputs and outputs to the UPE are bit serial. UPE's can be connected together with a single wire.

2 The Processing Element and Connection Network

Each UPE consists of 32 stages 0, 1, ... 31, as shown in Figure 2. There is one simple stage for each bit in the multiplier word, B, applied as an input to the UPE. The multiplier bits are stored in inverse order in flip-flops, such as the one shown in the detail of stage 0.

Each simple stage contains an AND function for one bit of multiplication, a flip-flop for one bit of storage for the carry, and a three input adder to sum the output of the preceding stage (or the input A in the case of the first stage) with the one bit product and the carry from the last one bit multiply. At each bit time the output of each adder, a_{i+1}, contributes to one bit in the final result $A + (M \times B)$.

The AND function is implemented with a multiplexer that chooses the input to the adder between a bit of the stored word B and a bit of the stored word D. The multiplexer is controlled by the multiplicand M so that each stage computes $b \cdot m + d \cdot (1 - m)$ and the entire array computes $A + [B \times M + D \times (1 - M)]$. If the word D is zero, then each of the multiplexers effectively performs as an AND gate, with each stage computing $b \cdot m$, and the entire array of UPE stages computing $A + [B \times M]$. If the word D is not zero, the final result is the linear interpolation between D and B, with M being the interpolation constant, i.e., the result equals $A + (B - D) \times M + D$.

The multiplier B is stored in the multiplier register in reverse order, that is with bit b_0 in stage 0, bit b_1 in stage 1, and so on, by placing the multiplier on the B input line one bit at a time, as a load control pulse is passed from state to stage. As each stage receives the load pulse, it loads its flip-flop with the current bit on the B input line. The D input is loaded into a separate register in the same manner when it is required. The multiplicand M is not stored in a register, but is delayed one bit cycle in each stage so that it can flow through and be operated by each bit of the multiplier B, one bit at a time. Thus, as the multiplier B is being loaded, it is possible to begin passing the multiplicand M into the array of stages and perform the first 32 bits of multiplication.

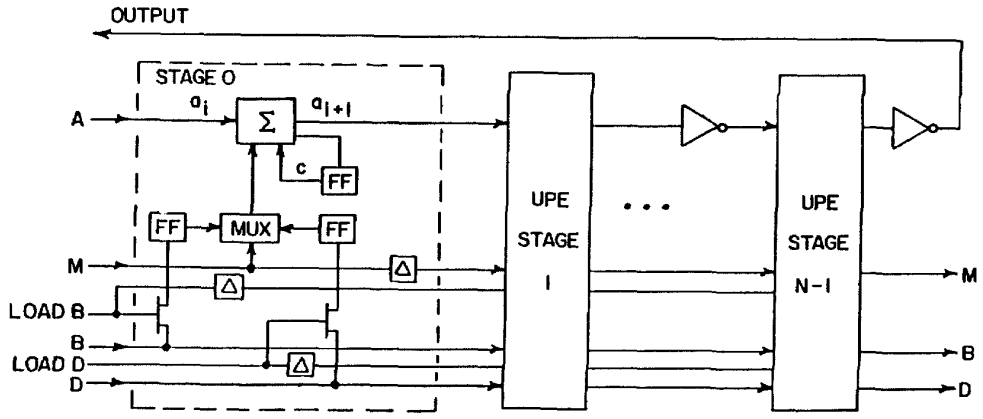


Figure 2: UPE stages

In the course of the mutiplication operation, each bit of the final result is formed by every stage adding its result to the result from the previous stage, and passing it on. Consequently, there is a propagation delay for each bit of the final result proportional to the number of stages. This delay can be avoided by using a conventional pipelining technique [3] which consists of the addition of an extra bit-time delay element on the a_{i+1} line, and on every one of the lines which connects from one stage to the next. These extra delay elements are not shown in Figure 2 to simplify the diagram.

The advantage of pipelining is that propagation delay for the array is proportional only to the delay in one stage, and not to the number of stages, although it does cause an initial delay through the pipeline. However, if the data being processed is a continuous stream, as in sound synthesis, this delay proportional to the total number of stages contributes only to the latency of the system, but does not affect its throughput.

Figure 3 illustrates the architecture used in each UPE. It contains 32 pipelined stages (0 through 31), along with the same number of stages of a shift register, shown as flip-flops $FF_0, FF_1 ... FF_{31}$. The end result Y at the output of the 32 stages is fed into a sign extension circuit which generates a U output by passing only the most significant 32 bits of the Y output, and then extending its sign bit over the next 32 bit cycles. Because the Y output is the product of two 32-bit numbers, it consists of 64 bits. Consequently, the first 32 bits of that product not used for the U output are stored in the 32-bit shift register. Since the Y output is thus delayed by 32-bit cycles, both the Y output and the U output appear in synchronism. It should be noted that the entire system of Figure 1 is synchronized by word pulses (not shown). In our system the word pulses are those controlling the digital to

analog conversion.

The B input and the D input (not shown), are 32-bit two's complement numbers, whereas M and A are 64-bit two's complement numbers. However, it should be understood that the bit serial architecture implemented to perform multiplication and linear interpolation does not depend upon use of the two's complement. The two's complement representation is chosen only because it is more convenient.

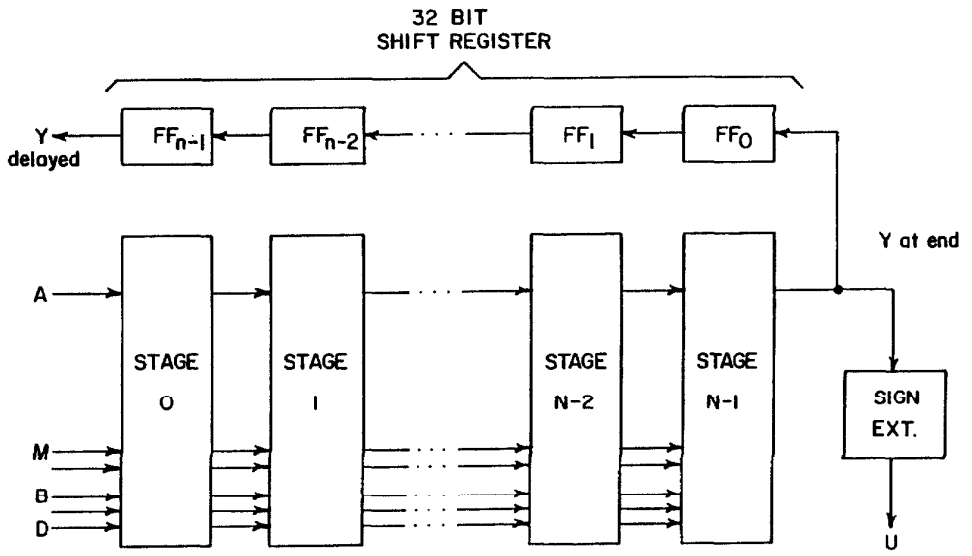


Figure 3: UPE architecture

A modification to the array of stages is necessary to accommodate two's complement numbers. Any n bit two's complement number with m bits of fraction may be written as:

$$\sum_{i=0}^{n-2} b_i \cdot 2^{i-m} - b_{n-1} \cdot 2^{n-1-m},$$

where b_0 represents the least significant bit (LSB). Since each stage of the multiplier holds one bit of the word B, with stage $n-1$ holding b_{n-1}, the last stage (most significant) must perform a subtraction of the incoming signal instead of an addition as in the other stages. The last stage is implemented with an inverter on the incoming partial product along with an inverter on its output as shown in Figure 2. A two's complement number at the M input must be sign extended to guarantee correct operation. For example, if M is a 32-bit number then after all 32 bits of M have been fed in, an additional 32 bits, each a copy of the sign bit, must follow.

Using a fractional representation for numbers facilitates the computation of linear interpolations with the same efficiency as multiplication. This capability is made possible by the fact that if the multiplicand M is a positive fraction and is represented by .xxxxx, then the one's complement $\overline{M} \approx 1-M$. It is this fact that is employed in implementing the AND function required for the one bit multiplication in each stage by a multiplexer (MUX), as shown in Figure 2. It should be recalled that the MUX is controlled by the multiplicand M to choose between the two signals B and D.

The last point that should be noted about the basic architecture of the UPE is that each stage receives its input from the stage of lower order. The first stage (stage 0) has no stage of lower order and therefore takes its inputs from the switching matrix shown in Figure 1. The input A for stage 0 need not be 0 in which case a number A is added to the final result.

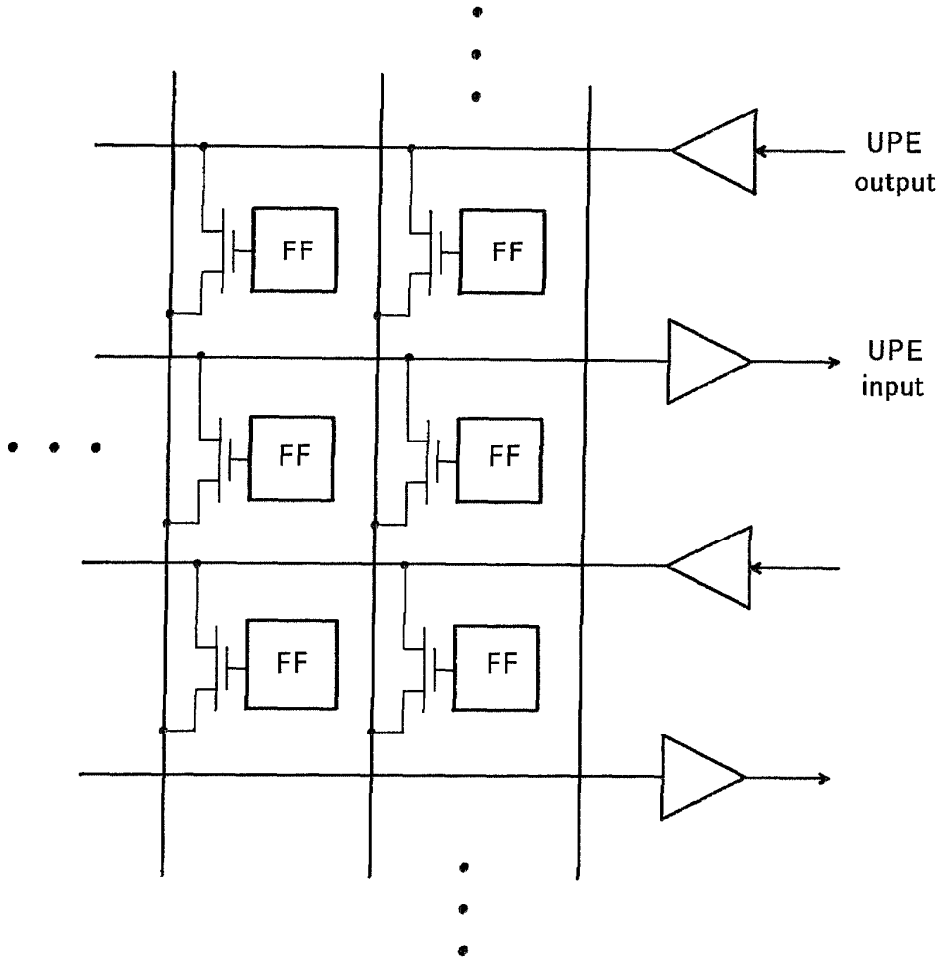


Figure 4: Interconnection Matrix

The interconnection matrix is shown in more detail in figure 4. Each UPE output is programmed to connect to one line that is broadcast to a neighborhood of other UPE's. Inputs to UPE's are programmed in a similar manner by connection to one of the broadcast outputs. Configuring the interconnect is achieved by placing bit patterns in the control flip-flops.

Inputs to UPE's that do not come from other UPE's, come from the controlling computer through an interface similar to the one connecting UPE's. Once a UPE receives an input it is held, so new values are sent only when the parameters of the model change.

For music synthesis, most interconnection patterns exhibit a high degree of locality. For this reason, the interconnection network need not provide full

connectivity. Figure 5 shows a scheme where there are a large number of short local wires, and proportionally fewer wires of greater length. Many instrument models have been found to map well into this wiring scheme.

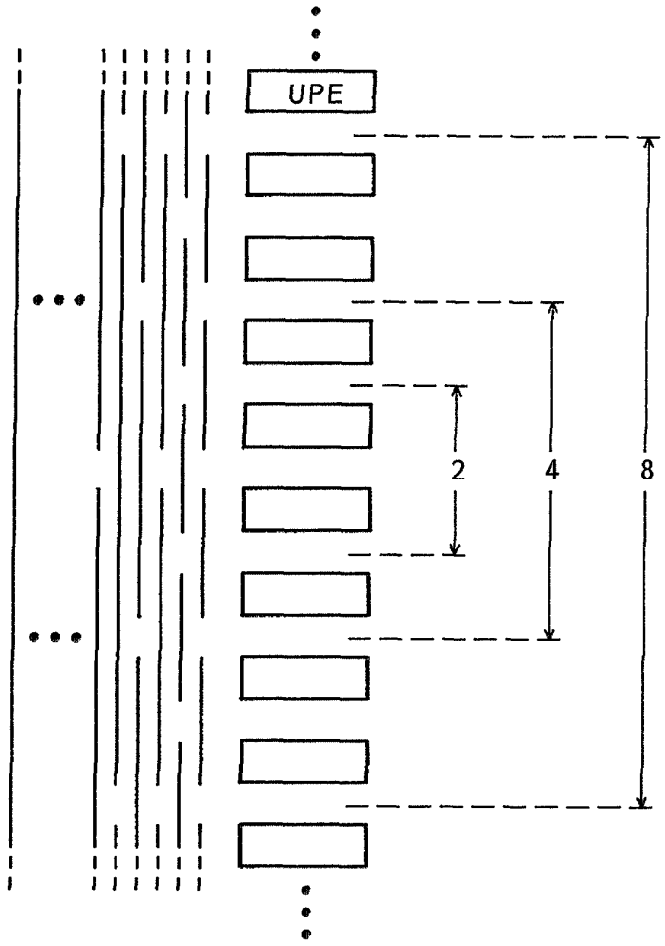
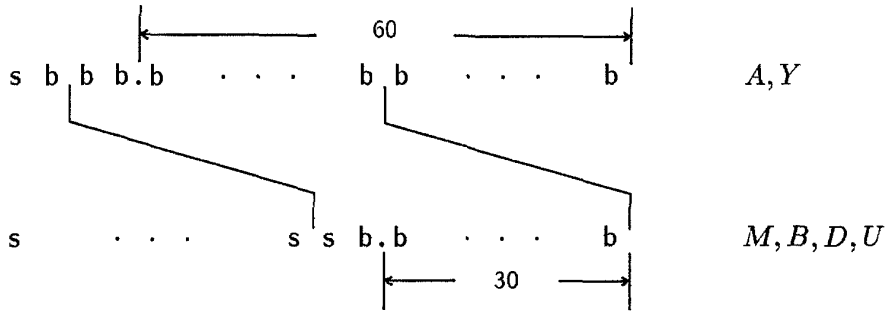


Figure 5: Discretionary Interconnect

Before describing typical applications of the UPE's with various examples, we will introduce a symbol to be used for a UPE, with delays implemented as described above. It consists of a rectangle with the four inputs A, M, B and D, and the two outputs Y and U. The M, B and D inputs and the U output are 32-bit two's complement numbers between 2 and -2, which are sign extended to 64 bits in the case of M and U. The A input and the Y

output are two's complement numbers between 8 and -8, as follows:



These two types of numbers restrict the way several UPE's may be interconnected. Except in special cases, the type of any output which feeds an input must match. For simplicity we adapt the convention that when $D = 0$, it is not shown.

3 Applications in Sound Synthesis

3.1 Basic Elements

3.1.1 General Linear Filter

An M^{th} order linear difference equation [4] may be written as:

$$y_n = \sum_{i=0}^{N} a_i x_{n-i} + \sum_{i=1}^{M} b_i y_{n-i} \tag{2}$$

where x_n is the input at time sample n; y_n is the output at time sample n; and the coefficients $a_0 \ldots a_N, b_1 \ldots b_M$ are chosen to fulfill a given filtering requrement. The function is evaluated by performing the iteration (2) for each sample time. This is the general form of a linear filter; any linear filter can be described as a special case of (2).

Figure 6 illustrates a UPE network which directly implements the general linear filter equation.

Each UPE, (with $D = 0$) performs the function $(A + M \times B)z^{-1}$, i.e. a multiply, an addition and one unit of delay. Referring to figure 6, the input values are processed by distributing the input signal x to each of $N+1$ UPE's, each one multiplies the input by a filter coefficient a_i, sums the result of the last UPE, and passes the total on for furthur processing. Since each UPE provides one unit of delay, the signal at the output of the input processing section is:

$$X = a_0 x_{n-1} + a_1 x_{n-2} + a_2 x_{n-2} + \ldots + a_M x_{i-M+1}. \tag{3}$$

This result is summed with the result of the output processing section.

The output y_n is distributed back to each of M UPE's. Each UPE multiplies the output by a filter coefficient b_i, provides one unit of delay, sums it's result with that of the last UPE, and passes the total on. The result at the end of the output processing section is:

$$y_n = b_1 y_{n-1} + b_2 y_{n-2} + \ldots + b_N y_{n-N} + X. \tag{4}$$

The result of the input processing section is added to the output processing section by feeding it into the UPE holding the b_n coefficient, since it's A (addend) input is not used. Adding the result from the input processing section to the UPE holding the b_n coefficient has the effect of adding a net delay through the system equal to the number of UPE's in the output processing section.

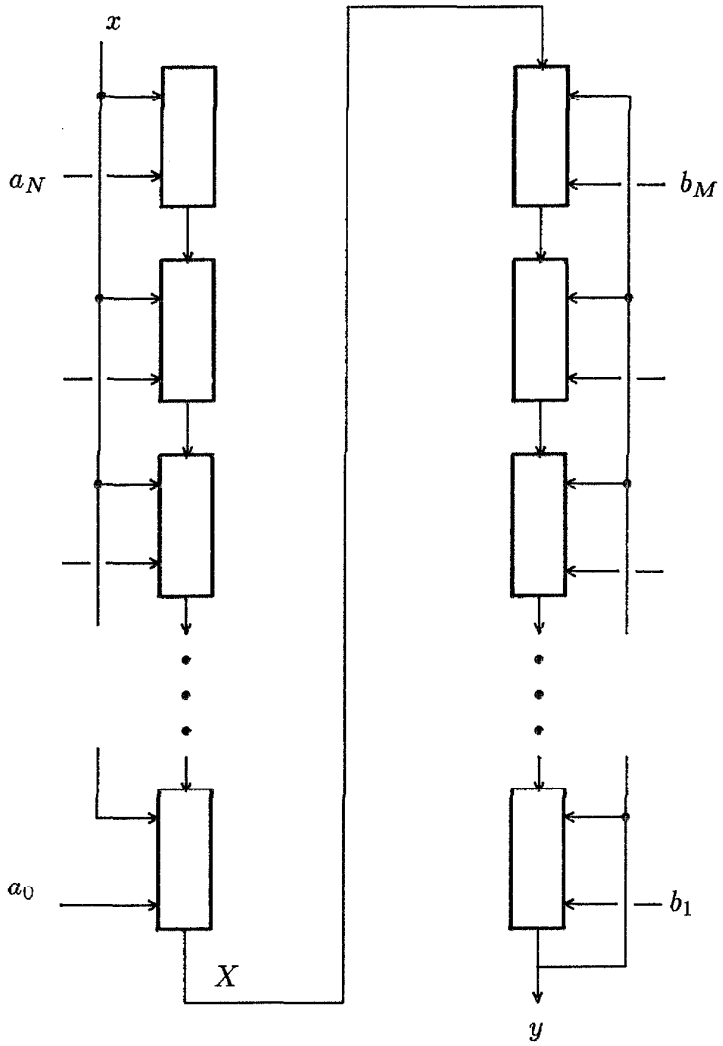


Figure 6: UPE implementation of General Filter

From figure 6 it is clear that the number of UPE's needed to implement equation (2) is equal to *the number of coefficients in the input (non-recursive)*

processing section plus the number of coefficients in the output (recursive) processing section.

3.1.2 Second Order Section

As an example of a linear filter, consider the second order linear difference equation:

$$y_n = \alpha y_{n-1} + \beta y_{n-2} + x_n. \tag{5}$$

Applying the z-transform we form the system function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1}{1 - \alpha z^{-1} - \beta z^{-2}}. \tag{6}$$

Solving for the roots of the denominator leads to two cases. In the case where $\alpha^2 + 4\beta \leq 0$ the poles of $H(z)$ are complex conjugates. They appear in the z-plane at $z = Re^{j\theta_c}$ and $z = Re^{-j\theta_c}$ as shown in figure 7. Here $\theta = 2\pi \times freq/f_s = \omega T$, where $f_s = 1/T$ is the sampling frequency. R is the radial distance of the poles from the origin in the z-plane and θ_c is the angle off the real axis.

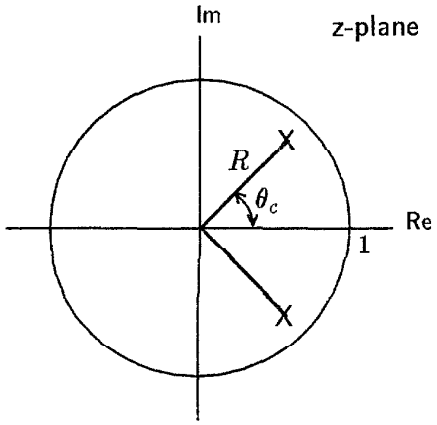


Figure 7: Second-order resonator poles

Now we can rewrite equation (6) as:

$$H(z) = \frac{1}{(1 - Rc^{j\theta_c} z^{-1})(1 \quad Rc^{-j\theta_c} z^{-1})}. \tag{7}$$

Multipling out the denominator we get:

$$H(z) = \frac{1}{1 - 2R\cos\theta_c z^{-1} + R^2 z^{-2}}. \tag{8}$$

Rewriting equation (5) yields:

$$y_n = 2R\cos\theta_c y_{n-1} - R^2 y_{n-2} + x_n. \tag{9}$$

It is easy to show that equation (9) leads to a sinusoidal time domain impulse response of the form:

$$\gamma R^{n-1}\cos[(n-1)\theta_c + \phi], \qquad n \geq 1 \tag{10}$$

where γ and ϕ depend on the partial fraction expansion of equation (9). For values of $R < 1$ the response is a damped sine wave with R controlling the rate of damping and θ_c controlling the frequency of oscillation.

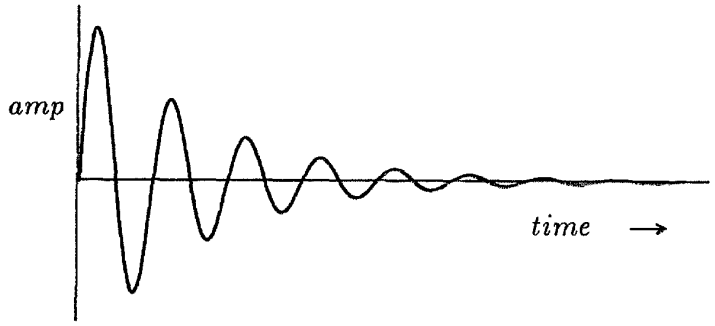


Figure 8: Time domain impulse response

With $R = 1$, the impulse response is a sine wave of constant amplitude, i.e. the system is an oscillator.

The system frequency response is found by substituting $e^{j\theta}$ for z in $H(z)$. At $z = e^{j\theta}$, $H(z)$ is identical to the discrete Fourier transform. The digital resonator acts as a bandpass filter in this case, with a center frequency of θ_c and a bandwidth proportional to R.

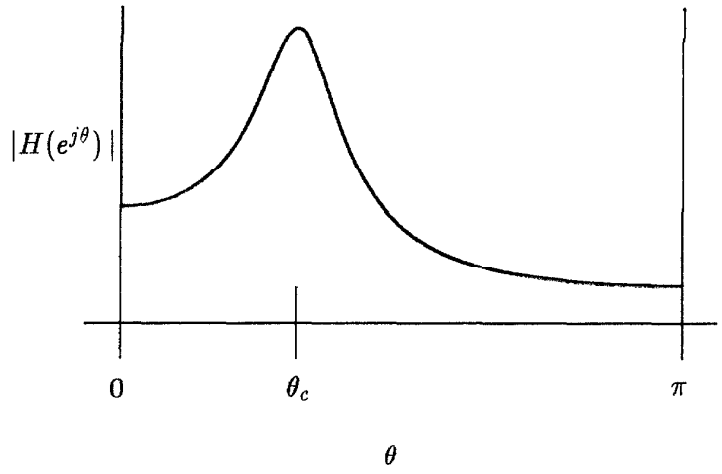


Figure 9: Magnitude of Frequency Response of case 1

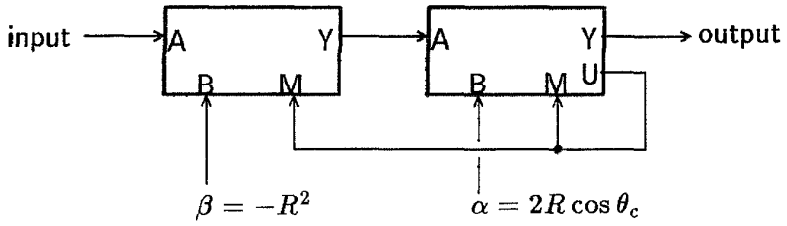


Figure 10: UPE implmentation of second-order section

The digital resonator is implemented directly using two UPE's. Refering to figure 10, the left UPE computes:

$$(-R^2Y + X)Z^{-1}$$

the right UPE computes:

$$[2R\cos\theta_c Y + (-R^2Y + X)z^{-1}]z^{-1} = 2R\cos\theta_c Y z^{-1} - R^2 Y z^{-2} + Xz^{-2}$$

hence,

$$y_n = 2R\cos\theta_c y_{n-1} - R^2 y_{n-2} + x_{n-2}.$$

Using the UPE implementation described above, oscillations in audio range have been run for tens of hours with no detectable change in amplitude.

3.1.3 Nonlinear Element

The range of functions computable by UPE's is not restricted to linear ones. Certain phenomena in nature are best modeled as nonlinear functions. For example, consider the class of functions that relate pressure to velocity at the mouthpiece of a blown musical instrument. A function that is characteristic of flute-like instruments is shown in figure 11c. This function and it's variations, shown in figure 11a through 11d, are computed using three UPE's, as is shown in figure 12. The input signal x is sent to u_1 that multiplies x by itself creating a squared term. This same technique is used again to arrive at the function:

$$y = k_0 + k_2 k_3 + k_3 G x + k_2 x^2 + G x^3,$$

which is a 3^{rd} order polynomial. For $k_0 = 0$ and $k_3 = -1$ the coefficient G controls the nonlinear gain, as illustrated in figure 11c and 11d. The coefficient k_2 controls the symmetry about the vertical axis, as shown in figures 11a through 11c.

This technique of generating polynomials can be extended to produce polynomials of arbitrarily high degree.

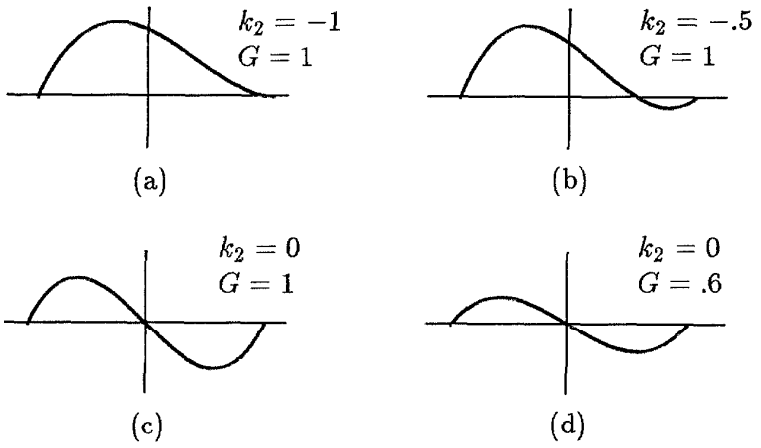


Figure 11: Non-Linear Function

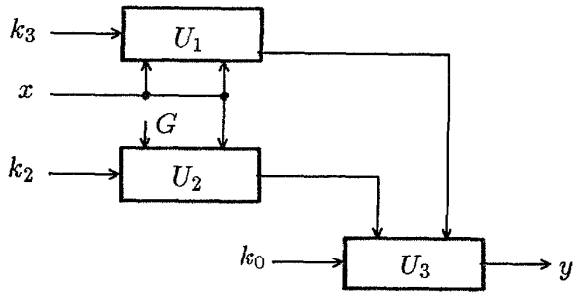


Figure 12: Non-Linear Element Implementation

3.1.4 Integrator

A very simple configuration using one UPE forms a digital integrator. The Y output is fed-back to the A input and the B and M inputs are controlled externally, as shown in figure 13a. The computation performed is:

$$y_n = B \times M + y_{n-1}.$$

At each step in the computation, the quantity $B \times M$ is summed with the result of the last step. This arrangement produces a ramp function whose slope is the product $B \times M$. As the computation proceeds, the output y_n eventually overflows the number representation and wraps around to a negative number, where the computation continues. The waveform for constant B and M is drawn in figure 13b.

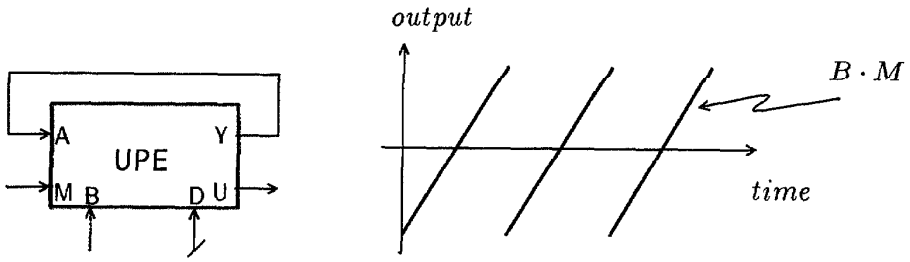


Figure 13: Integrator

3.1.5 FM

Because of the discontinuity, the ramp signal is not bandlimited, and therefore cannnot be used directly for sound synthesis, without aliasing components. However, in a scheme suggested by R. Lyon, the signal is remapped by passing it through a function, such as the one described in section 3.1.3. In the case where the remapping function is equal at the extremes of the number representation, as in the third order polynomial presented above, the resulting waveform is continuous. The resulting signal does not have the aliasing problems of the ramp function and can be used directly for musical sound application.

In this composite system, where the ramp output feeds the non-linear section, as shown in figure 14, the $B \times M$ input to the ramp may be thought of as controlling the phase of some periodic function y, and is either positive or negative. Since the $B \times M$ input may be a signal generated by another arrangement of UPE's, frequency modulation (FM) may be attained. The function generated by the non-linear element is the carrier signal and the signal fed to the $B \times M$ input is the frequency modulation signal. This scheme is therefore equivalent to the waveform-table lookup techniques commonly used in conventional computer music programs.

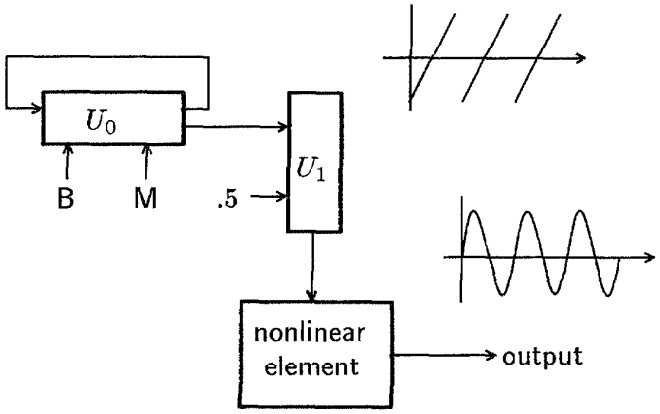


Figure 14: Frequency Modulation

3.1.6 Noise

Random signals find frequent application in sound synthesis. A pseudo-random number generator can be contructed with one UPE as shown in

figure 15. This approach uses a linear congruence method [5] implementing:

$$x_n = p \cdot x_{n-1} mod_r + q,$$

where

$$r = 2^{32}.$$

The mod_r operation is acheived by feeding the 64 bit output Y into the 32 bit input, B. Only the low 32 bits of Y get loaded, which effectively generates $mod_{(2^{32})}$.

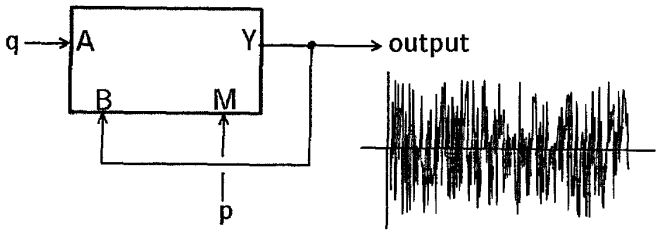


Figure 15: Random Number Generator

3.1.7 Mixer

The linear interpolation feature of the UPE's can be used for mixing signal. Referring to figure 16, one signal is fed into the B input and another into the D input. The M input controls the relative balance of the two signals in the output signal. This approach has the advantage over other schemes, that the output level is held constant as the relative mix of the two input signals is changed.

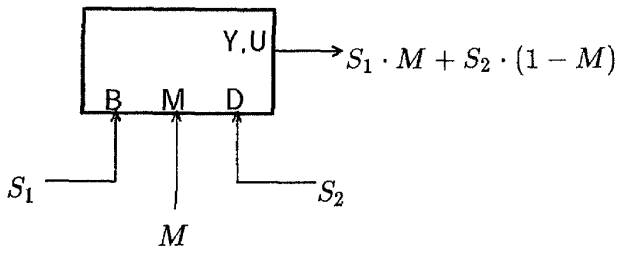


Figure 16: Mixing Signals

3.2 Musical Instrument Models

This section describes two simple musical intrument models based on UPE's. Both models are implemented and are being used to generate musical sounds. While these models have been used to produce extremely high quality timbres of certain instruments, they are certainly not capable of covering the entire range of timbres in the class. The development of a new timbre can be thought of as building an instrument, learning to play it, and then practicing a particular performance on it. This activity requires a great deal of careful study, and may involve extensions or modifications to the model.

3.2.1 Struck Instrument

Struck or plucked instruments are those that are played by displacing the resonant element of the instrument from its resting state and then allowing it to oscillate freely. Tone quality in such instruments is a function of how the system is excited, and of how it disipates energy. Examples of plucked and struck instruments include: plucked and struck strings, struck bells, and marimbas, etc.

Figure 17 illustrates a struck instrument model implemented with UPE's. The model may be decomposed into two pieces; the *attack section* and the *resonator bank*. The attack section models the impact of the striking or plucking device on the actual instrument. An impulse is fed to a second-order section that is tuned with a Q value close to critical damping. A detailed version of the attack section is shown in figure 18. In this figure, the output of the *attack resonator* is feed to the input of the *noise modulation section*. The noise modulation section generates the function:

$$y = NM \cdot x \cdot RNG + SG \cdot x,$$

where RNG is the output of a random number generator. This computation adds to the signal input x an amount of noise proportional to the level of x. The balance of signal to noise is controlled by the ratio, $SG : NM$, and the overall gain is controlled by $SG + NM$.

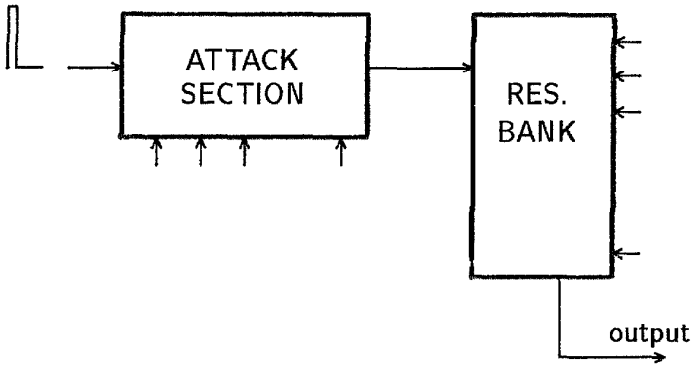


Figure 17: Struck Instrument

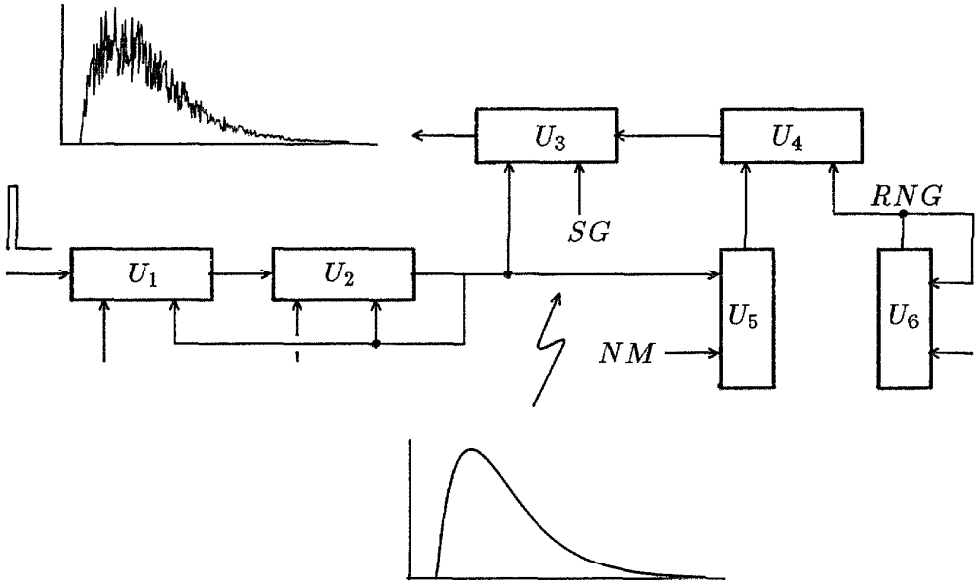


Figure 18: Attack Section

The output of the noise modulation section is used to drive a parallel connection of second-order sections used as resonators. The resonators are tuned to the major resonances of the instrument being modeled. The parameters of the attack section: attack resonator frequency and Q value, signal to noise ratio, and attack level, are all adjusted to produce a variety of musical timbres.

Second order sections are combined to form a resonator bank, as shown in figure 19. Each resonator, labeled RES_1 through RES_n, is implemented as

described in section 3.1.2. The output of each resonator is connected to a single UPE that scales the output of the resonator and adds the signal to the signal from the other resonators. The final output emerges at the output of the UPE connected to RES_n.

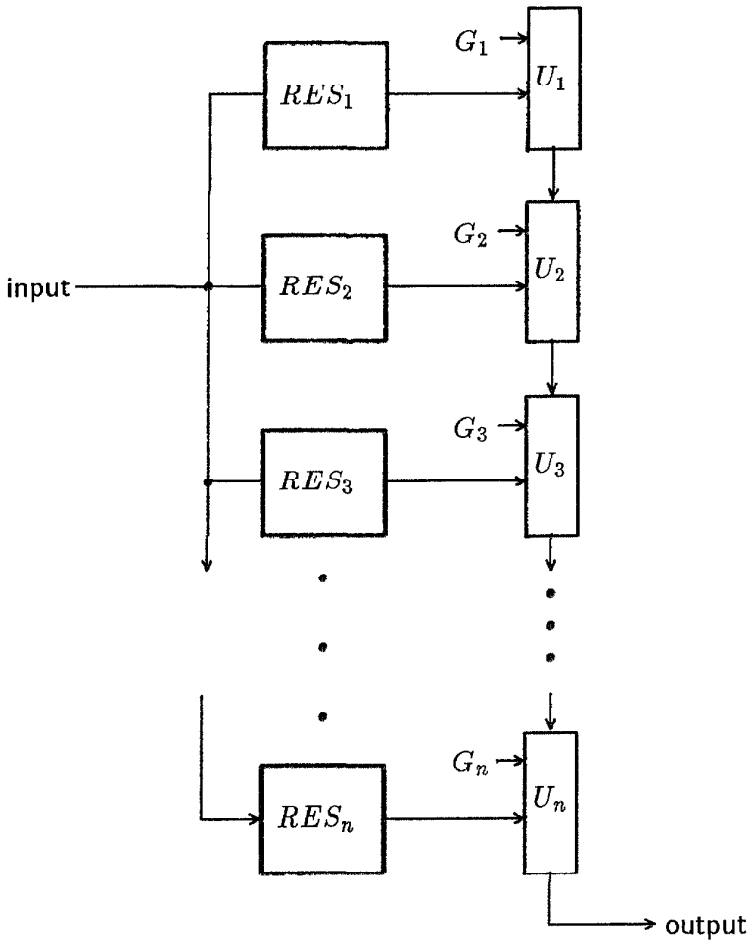


Figure 19: Resonator Bank

The gain at resonance of a 2-pole second order section varies drastically over the frequency range. This variation causes scaling problems when fixed point arithmetic is used. Either the input to or the output from each resonator must be adjusted to compensate for the implicit gain of the resonator. Several techniques exist for normalizing resonator gain. One proposed by Smith and Angell [6], uses the addition of two zeros to the second-order system function. By placing a zero at $\pm\sqrt{R}$ the dependence on θ in the system function may

be eliminated. Resonator gain normalization could pose a particularly severe problem in the case of resonators banks as shown in figure 19. Scaling the input to each resonator increases the amount of UPE's by a factor of one third and increases the control bandwidth by the same amount. Alternatively, the input to the entire system can be scaled down, to avoid overflow in the section with the most gain, and then the output scaled up to the appropriate level. This approach is a problem in systems that use fixed point arithmetic because the amount of gain available at each multiplication is limited, and hence many multiplier stages at the output must be used.

In many sound generation applications the R values of each stage in the resonator bank are close in value. Therefore, it is possible to synthesis two zero's using an average value for R and then distributing the result to each resonator.

In a typical application, a piano-like keyboard is used to control the instrument. The pressing of a key triggers the following actions: 1) the key position determines the coefficients loaded into the resonator bank, 2) the key velocity controls the level of the coefficient NM in the attack section (higher key velocities correspond to more noise being introduced into the the system and hence a higher attack level), and 3) the key press generates an impulse that is sent to the attack resonator.

3.2.2 Dynamic Model

Figure 20 shows a simple model for blown instruments, implemented using UPE's. This model has been motivated by the observation that a blown musical instrument may be viewed as a nonlinear forcing function at the mouth-piece exciting the modes of a linear tube.

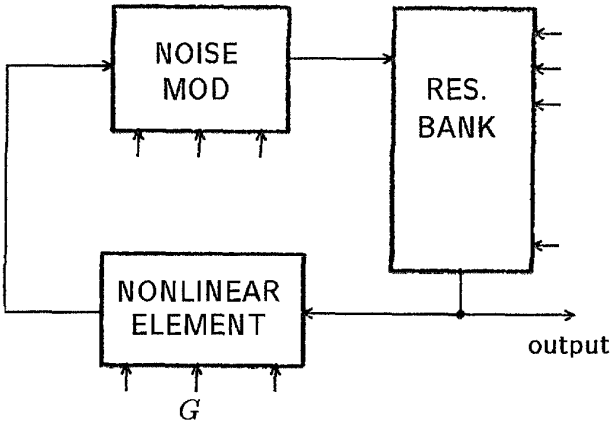


Figure 20: Dynamic Model

The dynamic model is composed of three pieces described in earlier sections: 1) the nonlinear element that computes a 3^{rd} order polynomial, 2) the noise modulation section that adds an amount of noise proportional to the size of the signal at it's input, and 3) the resonator bank that has second-order resonators tuned to frequencies corresponding to the resonances of the musical instrument.

These elements are connected in a cascade arrangement forming a closed loop. In the case where the closed loop gain is sufficiently high, and the system is disturbed, it oscillates with modes governed by the tuning of the resonator bank. Typically, the loop gain is controlled by the gain of the nonlinear element G. For small values of G the feedback is too small and the system does not oscillate. If G is just large enough, the system will oscillate with a very pure tone as it operates in the nearly linear range of the nonlinear element. If the non-linear gain G is set to an even higher value, the signal is increased in amplitude and is forced into the nonlinear region. The nonlinearity shifts some energy into higher frequencies, generating a harsher, louder tone.

In a typical application, the loop gain is set by controlling the nonlinear gain G according to the velocity of a key-press on a piano-like keyboard. A slowly pressed key corresponds to a small value for G, and thus a soft pure tone. A quickly pressed key corresponds to a larger value for G, and hence a louder, harsher tone. When the key is released G is returned to some small value, one that is just under the point where the loop gain is large enough to sustain oscillation. By not returning G to zero the signal dies out exponentially with

time, with a time constant that is controlled by the value of G used.

A small amount of noise is injected constantly into the loop, using the the noise modulation section, so that the system will oscillate without having to send an impulse to excite it.

This model has been used successfully for generating flute-like tones.

4 Conclusion

Our solution to the problem of sound synthesis is one that employs the flexibility provided by VLSI to build an architecture that is tailored to the computation involved in physical modelling of musical instruments. Our architecture is one that exploits the natural parallelism of the problem at every possible level. With current IC technology it is possible to place approximately 40 UPE's and an interconnection matrix on a single chip. This configuration allows the realization of a single instrument voice of about the complexity of the instruments presented in this paper on a single IC. Such a chip computes more than 3 million operations/second.

Musical sound synthesis has many attributes common with other problems in science and engineering. These are problems where a fixed (or slowly varying) interconnection between processing elements is sufficient. Once the interconnection topology is defined, the computation proceeds for a relatively long time before another interconnection change is made. Conventional signal processing can be viewed in this manner. In general this class of problems are those that may be represented as systems of difference equations, where "time" in the problem being modeled may be represented by time in the computation. Our belief is that the architecture presented here will find general application among this class of problems, as an efficient and sometimes necessary alternative to general purpose computers.

5 Acknowledgements

Many people have contributed in unique ways to the Music Project at Caltech. Tzu-Mu Lin (at the time, a Ph.D. candidate at Caltech) did much of the basic work on the UPE design. Lounette Dyer (graduate student) has built a high-level front end to the hardware and brings a musical sensitivity to the project. Hsui-Lin Liu (postdoctorate, now at Schlumberger – Research) comes from a background in seismology and acoustics and did work on physical modelling of musical instruments. Dick Lyon (Fairchild Research Lab), whose multipliers the UPE is based, has provided countless ideas and is always an inspiration. Greg Bala (Undergraduate, now at IBM) has worked

with the project from its beginning and has contributed application software. Ron Nelson (composer and Professor of Music at Brown University) has worked with the project to make our instrument models more realistic and useable. David Feinstein (graduate student) has done some exquisite mathamatics which helped us develop instrument models. Vibeke Sorenson (instructor at Art Center College of Design) was the first user of the sound synthesis hardware.

Special thanks are due Telle Whitney (graduate student) for her critiques and many discussions.

We would also like to thank Ron Ayres (USC/Information Sciences Institute) whose integrated circuit layout programs were used to generate the custom chips. The circuit boards that hold the custom chips were designed and built by Brian Horn. Jim Campbell and Alan Blanchard have contributed hardware support.

This work was supported by the System Development Foundation.

6 References

[1] Wawrzynek, J. C., and Tzu-Mu Lin
A Bit Serial Architecture for Multiplication and Interpolation
California Institute of Technology, Computer Science Department
Display File 5067.

[2] Lyon, R. F.
A bit-Serial VLSI Architecture Methodology for Signal Processing
VLSI 81 Very Large Scale Integration,
(Conf. Proc., Edinburgh, Scotland, John P. Gray, editor)
Academic Press, August 1981.

[3] Cheng, E. K., and C. A. Mead
A Two's Complement Pipeline Multiplier Proc. of 1976 IEEE International Conf. on Acoustics,
Speech and Signal Processing, Philadelphia, PA.

[4] Oppenheim A. V., and R. Schafer
Digital Signal Processing
Prentice-Hall: Englewood Cliffs, New Jersey, 1975.

[5] Knuth, D. E.
The Art of Computer Programming, Vol. 2
Addison Wesley, Reading, 1968.

[6] Smith, J. O., and J. Angell
A Constant-Gain Digital Resonator Tuned by a Single Coefficient
Computer Music Journal, vol. 6, no. 4, Winter 1982.