# Digital Turntable Setup Documentation

Nathan Artz, Adam Goldstein, and Matthew Putnam

## Abstract

Analog turntables are expensive and fragile, and can only manipulate the speed of music without independently changing its pitch. Additionally, hardware and software for matching beats between songs are expensive and hard to use. The Digital Turntable Setup avoids these pitfalls by implementing frequency- and speed-adjusting features, along with beatmatching and echo and reverb effects, on a standard FPGA. The system's behavior has been tested with music at multiple sampling rates.

# Table of Contents

# List of Figures

# Overview

## Objective

The Digital DJ Setup attempts to provide features to help a DJ play two tracks of music while adjusting the pitch and speed in various ways to make the music more danceable or fun to listen to.

## Goals

The goals of the system are:

- To record two tracks of music and play them back either one at a time or together
- To adjust the pitch of music, either live or recorded, with a knob
- To implement a “turntable” whose rotational speed determines the pitch or speed of the music
- To adjust the pitch of music at standard harmonic intervals using a computer keyboard
- To match the beats of two songs, slowing down or speeding up one as necessary
- To add echo and reverb effects to music
- To record the results of mixing into storage to make it possible to layer mixes

## Solution

The proposed system achieves the above goals through use of a modular hardware design. The system uses sophisticated memory architectures to record tracks as they arrive from an audio input, and to store them permanently in flash memory. Additionally, the memory architecture allows music to be recorded to memory as it is played back, making it easy to iteratively record several layers of music on top of each other.

Additionally, the design makes it possible for the user to choose whichever human interface he prefers; since all human interfaces output a sampling ratio, they require no internal changes to be switched from being used to adjust speed (resampling in the time domain) to being used to adjust pitch (resampling in the frequency domain).

The pipelined nature of the design allows effects like echo and reverb to be added to the “chain” of audio samples traveling through the system, smoothly integrating them with other effects like pitch- and beat-matching.

# Description

## Overall Design

The Digital DJ Setup comprises several interacting subsystems (Figure 1). First are the human interfaces, including a knob, a turntable, and a computer keyboard. These produce sampling ratios that can be used to adjust the pitch or speed of music.

Second is the pitch-adjusting subsystem, encapsulated in the Sound Router, that provides near-real time transposition of music.

Third are the Beatmatcher and Addresser, which let the user input the speeds of two different songs and have the song playback speed matched.

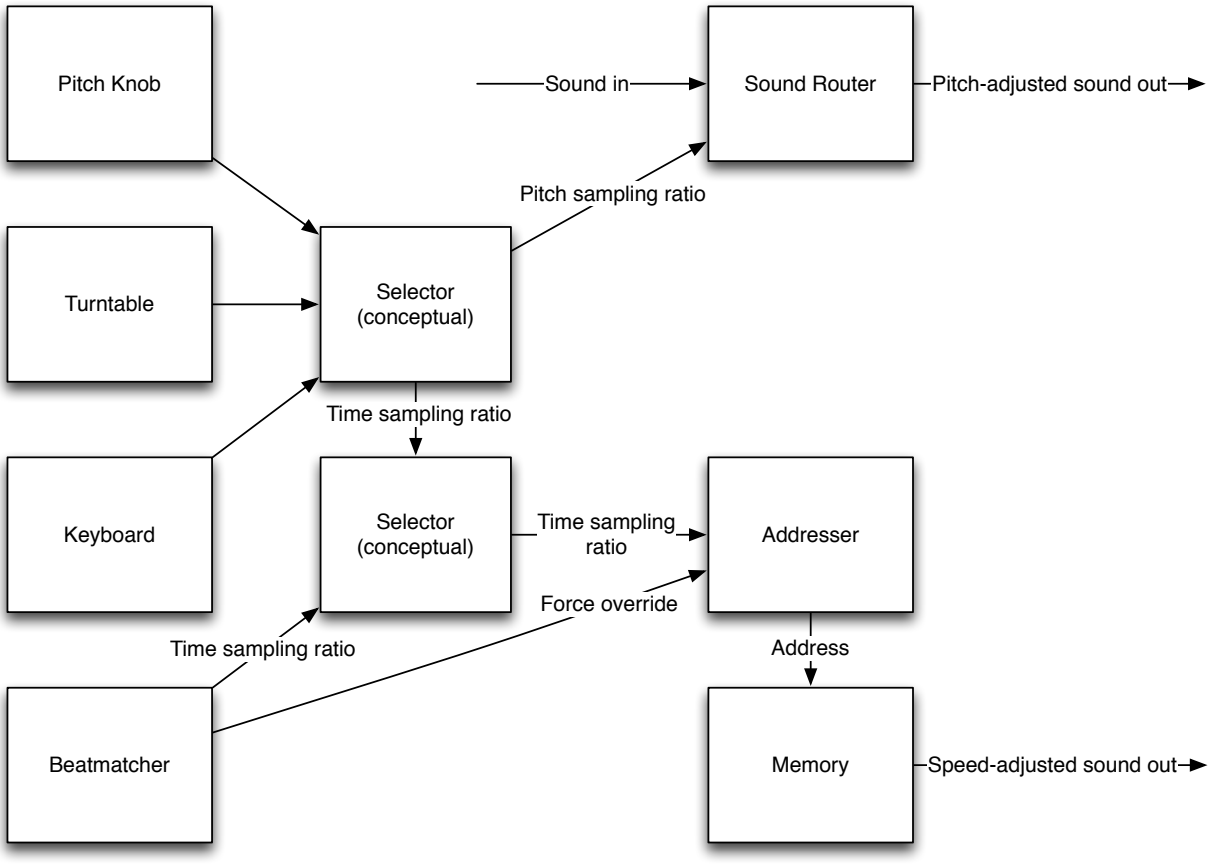


**Figure 1:** The overall design of the Digital DJ Setup. In practice, the "selectors" were not included; rather, the modules on the left were wired directly to the desired modules on the right.

Fourth is the memory subsystem, which handles the details of recording music and playing it back.

The final subsystem handles audio effects like echo and reverb, applied to the output of all the previous subsystems. All of these subsystems and their interactions are described below.

## Input Methods (Adam Goldstein)

There are a number of ways of inputting settings into the system to change pitch, playback speed, and volume.

### Turntable Input (Adam Goldstein)

One of the primary ways to input speed or pitch information into the system is using a "turntable"—a motor connected to a rotary encoder with a protruding plate. The Turntable Input module makes it possible to set a reference speed for the turntable. Later, when the turntable is spun faster or more slowly, Turntable Input produces a sampling ratio of two four-bit numbers that will cause the Addresser to change playback speed proportionally to the change in the turntable's rotational speed. (For instance, if the turntable were spun 50% more slowly than its reference speed, the sampling ratio would be 2/1, so playback would proceed at 50% its normal speed.)

In addition, Turntable Input determines the direction in which the encoder is spinning, and produces an output (*clockwise*) that represents the direction of rotation. This can be connected directly to the Addresser's *read forward* input to ensure that changing the direction of turntable rotation produces a corresponding change in the direction of playback.

Turntable Input works by counting the number of clock cycles that elapse in the periods between changes in the encoder's output (see Design Decision 1). To mitigate the effect of outliers and glitches, *average cycles per period* keeps a constantly running average of the number of cycles from the last four periods.

When *set reference* is asserted high, Turntable Input loads *average cycles per period* into *reference cycles per period* to store the speed of rotation during the turntable's reference state.

Separately, to calculate the sampling ratio, Turntable Input attempts to simplify the fraction of *average/reference cycles per* period by oscillating between two values of *calculating sampling:*

- **When *calculating sampling* is 0,** the module loads the current value of *average cycles per period* into *upsample possibility,* and the current value of *reference cycles per period* into

**Design Decision 1**

To make the system as general as possible, I made the sampling ratio output entirely dependent on changes in the value of *input val,* a two-bit input, rather than the specific operation of the encoder used in this system. The *input val* input can be connected to switches (which is how I tested the module before receiving the encoder), an encoder, or any other device that changes voltage over time.

*downsample possibility.* Then the module sets *calculating sampling to 1.*

- **When *calculating sampling* is 1, if the top bit of either *possibility* register is 1,** the module loads the top four bits of *upsample possibility* into *upsample* and the top four bits of *downsample possibility* into *downsample.* Then the module sets *calculating sampling* to 0.
- **When *calculating sampling* is 1, if the top bit of both *possibility* registers are 0,** the module left-shifts each *possibility* by one bit and tries again.

The process is illustrated in Figure 2.

To determine the direction of rotation, the Turntable Input pays attention to the specs of the specific encoder model used (Clarostat 600EN-128-C24), which says that a change going forward

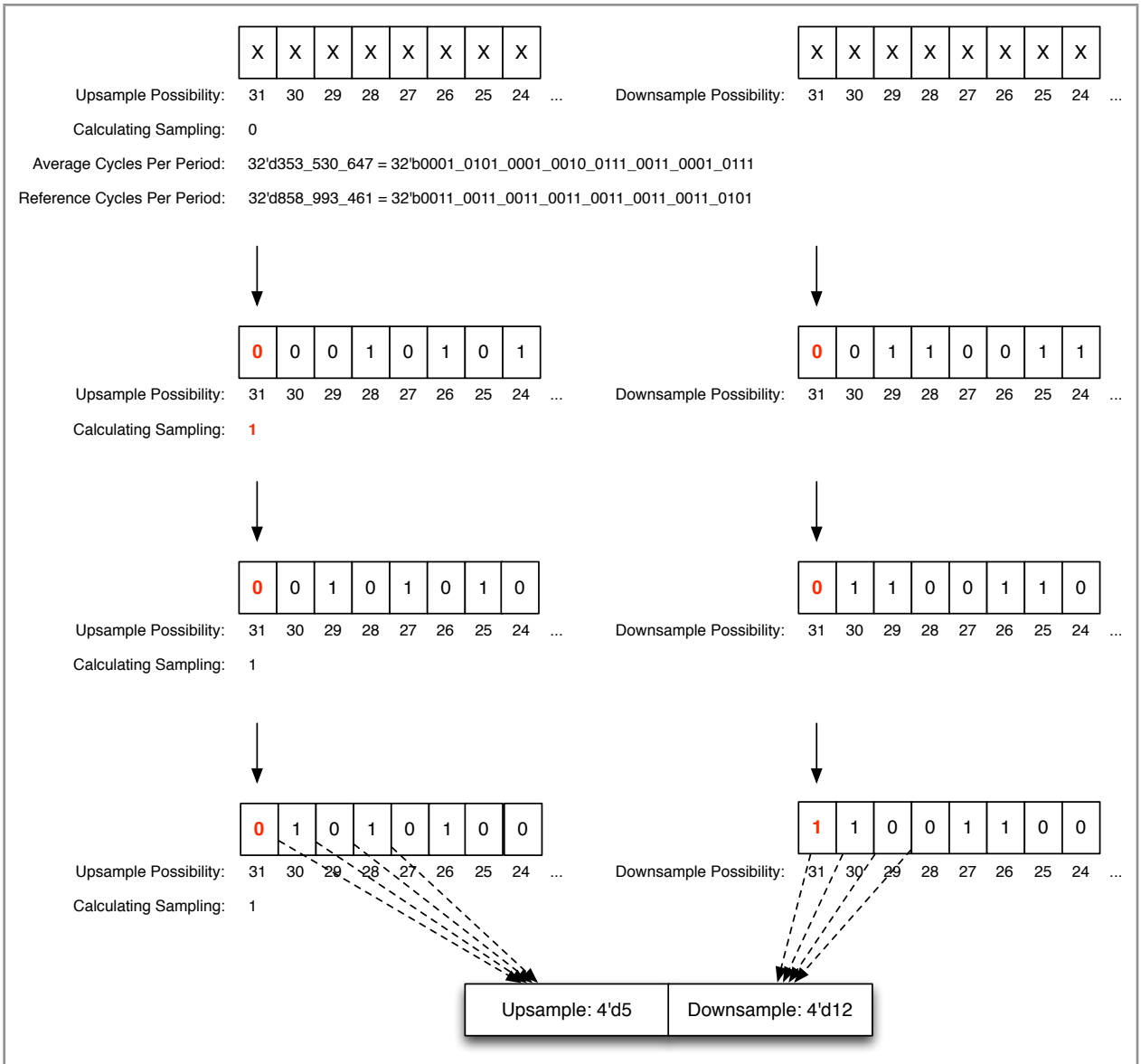


**Figure 2:** From top to bottom, the operation of the Turntable Input during and after the transition of *calculating sampling* from 0 to 1. The resulting ratio (5/12) is an approximation of the original ratio (353,530,647/858,993,461), and in this example only differs by only 1.2%.

in the series [2, 3, 1, 0, 2, 3, 1, 0, . . .] represents clockwise rotation, and a change going backward in the series represents counterclockwise rotation.

Finally, for debugging purposes, Turntable Input includes an additional output, *speed changed,* which represents whether the turntable has changed speed by more than 1/16 of its original value.

## Knobs (Adam Goldstein)

One of the most intuitive interfaces for a music setup is a knob. In this system, there are two modules that interface with a rotary encoder to make it behave like a knob: Volume Knob (for adjusting the volume of output music, naturally), and Speed Knob (for adjusting the speed *or* pitch of music). The two modules are described below.

### Volume Knob (Adam Goldstein)

The Volume Knob module has a single purpose: to output an 8-bit number representing the volume at which music should be playing. Volume Knob starts by outputting 8d'128, and increases or decreases its *volume* output when it detects a certain number of encoder output transitions. (The code for doing this is extremely similar to the transition- and direction-detecting code of Turntable Input.)

To change the responsiveness of the Volume Knob, other modules can use the *sluggishness* input. For example, when *sluggishness* is 4, the Volume Knob will only increment the *volume* output by 1 when the encoder signals that it has turned counterclockwise with four transitions.

To prevent wraparound errors, Volume Knob hard-limits *volume* at 0 when the knob is being turned counterclockwise and 255 when the knob is being turned clockwise.

### Speed Knob (Adam Goldstein)

The Speed Knob module, like Turntable Input, outputs a four bit-over-four bit sampling ratio that can be used for changing playback pitch or speed. Unlike Turntable Input, however, Speed Knob outputs its value based on the position of the rotary encoder relative to some reference position, not its speed relative to some reference speed.

For simplicity, the Speed Knob limits itself to outputting ratios with 8 in either the numerator or denominator, resulting in 15 possible outputs (15/8, 14/8, . . . 8/8, . . . 8/14, 8/15).

At the low level, Speed Knob works nearly identically to Volume Knob, detecting the occurrence and direction of encoder output transitions to determine changes to the module's outputs. Like Volume Knob, Speed Knob also prevents overshooting by hard-limiting the ratio at 15/8 and 8/15.

## Keyboard (Matthew Putnam)

The keyboard module takes input from the keyboard and produces the ratio of pitch frequencies that those keys are separated by. We had originally intended to use a MIDI keyboard for this, but all

new keyboards use a USB interface instead of the serial DIN 5/180° connector, making this impossible. Instead, we used a standard PS/2 keyboard and pretended that keys along the home row were the white keys of a musical keyboard, and the appropriate keys along the top row were the black keys. Using the 'a' through ';' keys as C and E respectively, we were able to represent 17 chromatic keys.

To interpret the raw incoming keyboard data, we used a keyboard driver provided by the course staff from the fall 2005 website. The module "ps2_ascii_input" takes in the system clock and reset signals and the PS/2 interface signals and outputs the ASCII code of the last key pressed and a ready signal for new data. This is slightly wasteful, as it means we are first converting raw key codes into ASCII values, and then converting that into pitch information, when we could be translating straight from key codes. However, using ASCII makes the signals easier to understand.

When the user presses two legal keys in succession, the module outputs a ratio in the form of a 4-bit numerator and denominator that represents the ratio (using just intonation) of those pitches. Because pitches are logarithmically spaced, this ratio is simply a function of how many half-steps are between them and is independent of their absolute location on the keyboard. Thus, the way this is calculated is by mapping the keys to the numbers 0-16 in order, taking the difference, and looking up the ratio in a ROM. Additional logic checks whether the input interval is ascending or descending and inverts the ratio if necessary. It should be noted that by limiting the output size to 4 bits, the ratios for a half step and a minor ninth are approximated. Those ratios are very dissonant, however, and would likely never be used.

If any invalid key is pressed, then the internal state of the module is reset. That is, if one key has been pressed and the module is waiting for the second, this will be forgotten and the module will be listening for the first key again. The purpose of this is to allow the user to cancel the ratio selection and start over.

## Sound Router (Adam Goldstein)

The highest-level module for changing pitch is the Sound Router (SR), which takes an incoming music stream and outputs a frequency-adjusted version of the same stream (Figure 3).

The Sound Router works with 512 samples of music at a time: buffering the input samples as they arrive, processing the input samples with the Phase Vocoder once all 512 have been buffered, buffering the processed samples, and outputting the processed samples one at a time (Figure 4). In this way, the SR produces a pitch-adjusted version of the input, at the expense of introducing an approximately 512-sample delay (see Design Decision 2).

The full Moore machine diagram for the Sound Router is shown in Figure 5. The states are as follows:

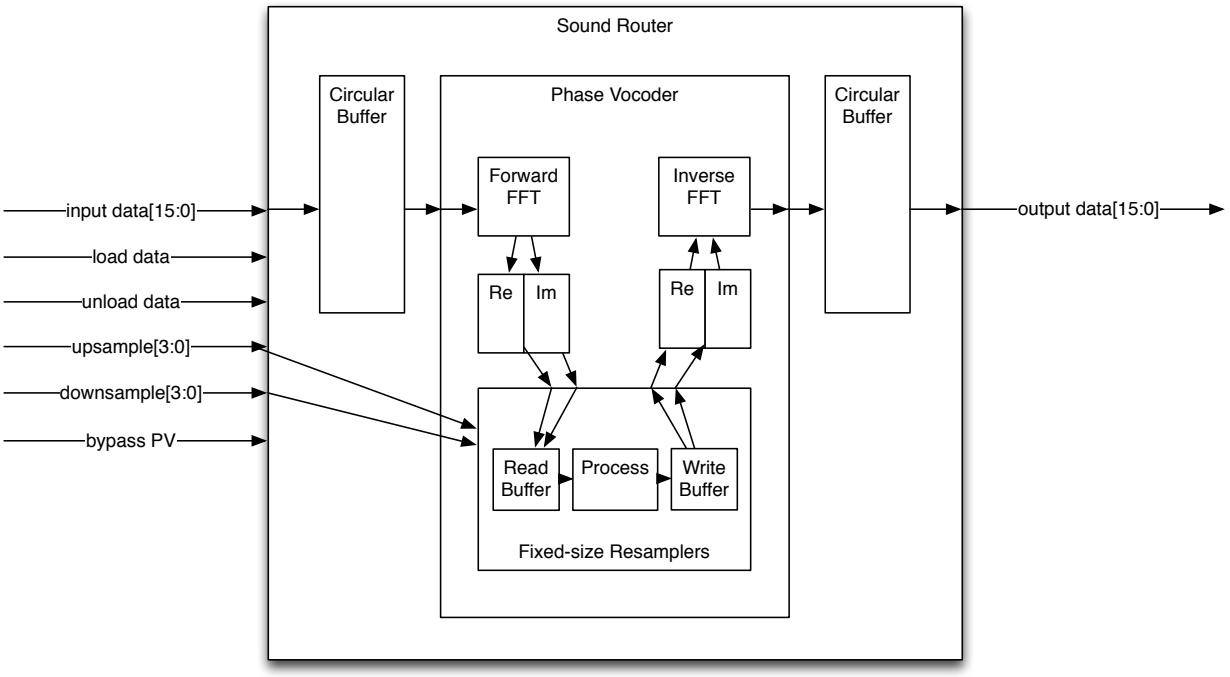


**Figure 3:** The flow of signals through the Sound Router and included modules.

- **Pre-reset.** This is the start state of the SR, and the state it returns to after completing a full load-process-unload sequence. Here the buffers are reset and the Phase Vocoder is instructed not to perform any transformations.
- **Start.** This state indicates the SR is waiting to be told to load an input sample (with the *load data* signal). In the meantime, this state caches each input sample so it isn't gone if the Sound Router jumps to *Load to Pre.*
- **Load to Pre.** This state, only occupied for a single cycle, indicates that the SR has just been told to load an input sample to the input (or "pre") buffer. If the current sample is the 512th, the Sound Router jumps to *Ready to Load to PV*. Otherwise, the SR jumps back to *Start* to wait for another sample.
- **Ready to Load to PV.** This state tells the Phase Vocoder that it should prepare to accept samples for

**Design Decision 2**

Originally, I had envisioned a method of producing the output stream that would only introduce a single-sample delay to the input. The idea was that the most recent 512 input samples would be stored in a circular buffer, and each time a new sample arrived, the entire input buffer would be pitch-adjusted. Then the last of the 512 pitch-adjusted samples would become the output of the Sound Router.

This mechanism didn't work during testing in MATLAB, and I had insufficient signal-processing knowledge to diagnose the reason.

The "Circular Buffer" modules (which are just used as regular FIFO buffers) are a holdover from this ill-fated experiment.

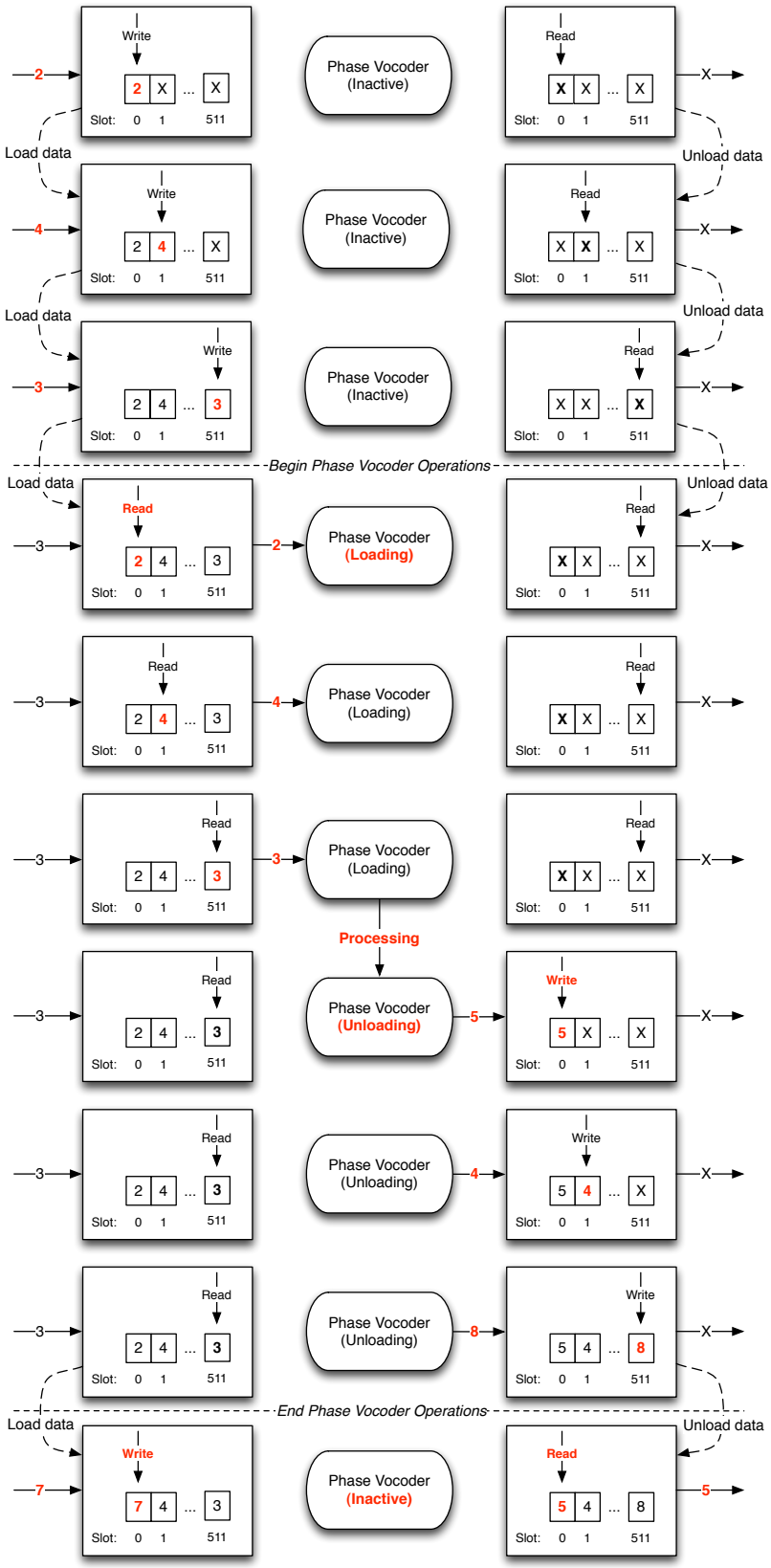


**Figure 4:** Top to bottom: the operations of the sound router. Red marks a changed value or the source of the change.

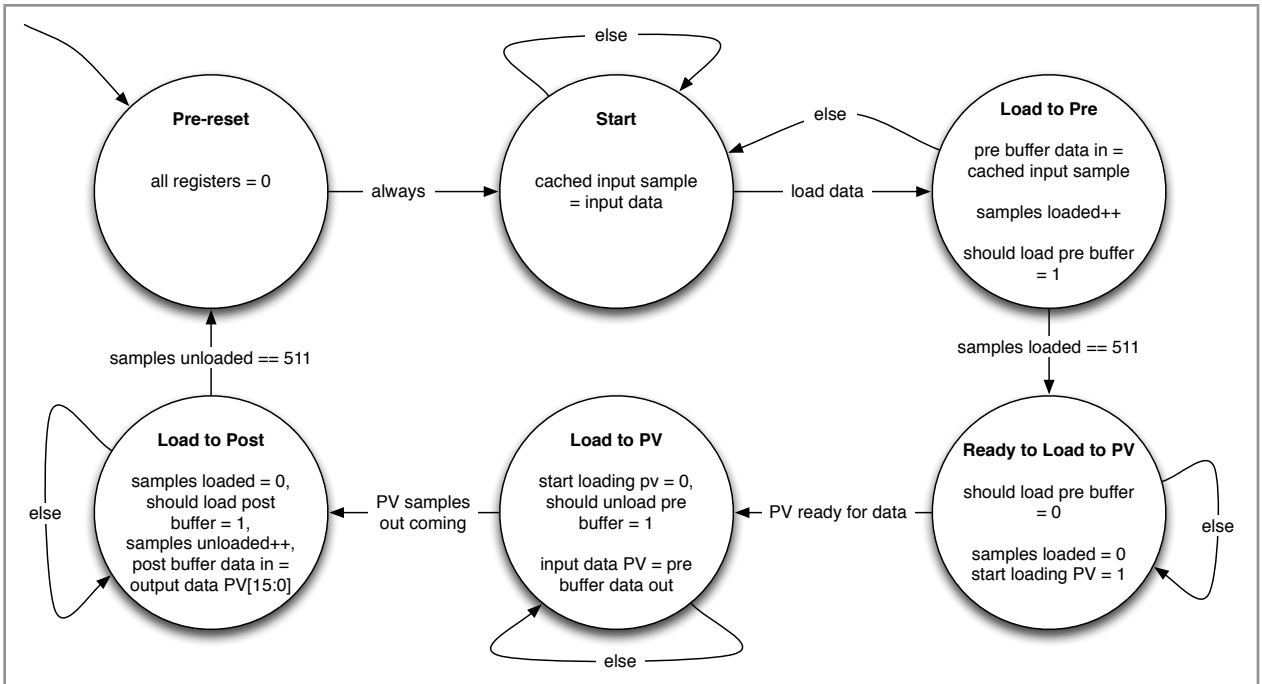


**Figure 5:** The states of the Sound Router module.

processing. When the Phase Vocoder is ready, it asserts *PV ready for data,* and the SR jumps to *Load to PV.*

- **Load to PV.** In this state the SR delivers all 512 buffered input samples to the Phase Vocoder for processing. When the Phase Vocoder is done and ready to output the processed samples, it asserts *PV samples out,* at which point the SR jumps to *Load to Post.*
- **Load to Post.** In this state the SR collects the 512 processed samples and stores them in the output buffer.

Note that a sample is being output from the SR during all these states; another module can read the output sample at any time via the *output data* bus. The other module indicates it wants to read the next output sample by asserting *unload data*.

Finally, to make it easy to compare processed and unprocessed sound, the Sound Router module has a *bypass PV* input that, when asserted high, short-circuits the input to the output. When connected to a button or switch, this feature makes it possible to rapidly compare the processed and unprocessed version of, for example, a song's chorus.

### Phase Vocoder (Adam Goldstein)

The Phase Vocoder (PV) handles the mechanics of the frequency manipulation for the Sound Router. The Phase Vocoder works with discrete 512-sample chunks of music, rather than continuous streams like the Sound Router (Design Decision 3).

The central idea of the Phase Vocoder is to take a set of consecutive music samples and a frequency scaling ratio, and to output a same-length set of consecutive music samples whose frequencies have been scaled as requested.

The PV achieves this by taking the Fourier transform of the incoming samples, resampling both the real and imaginary parts of the transformed frequencies using Fixed Size Resamplers, and outputting the inverse Fourier transform of the resampled frequencies, as shown in Figure 6. The PV uses a 512-long Fourier Transform module produced using CoreGen.

**Design Decision 3**

Originally, I had intended to make the "Phase Vocoder" module a true phase vocoder—a module that actively attempted to prevent distortion and to increase frequency accuracy by examining the phase of the transformed input samples.

Due to the long amount of time it took to learn how to work with the FFT module and the rapidly diminishing time remaining on the project, I decided to try the more straightforward spectral manipulation described here. To save hunting down and renaming dozens of signals and abbreviations, however, the "Phase Vocoder" module retained its name, even though it would be more accurately described as a spectral resampler.

The full Moore machine diagram for the Phase Vocoder is shown in Figure 7. The states are as follows:

- **Pre-reset.** This state is where the module begins. Upon getting a reset signal, the PV jumps to *Start.*
- **Start.** Here the Phase Vocoder indicates to the FFT module that it's ready to input data. When the FFT module replies with *ready for data* high, the PV jumps to *Load FFFT.*
- **Load FFFT.** Here the PV loads the 512 time-domain samples into the FFFT module to have it do a forward FT. When the FFT module has been done for 7 clock cycles (the number necessary to satisfy timing specifications for the FFT module), the PV jumps to *Unload from FFFT.*
- **Unload from FFFT.** In this state the PV writes the real and imaginary outputs from the forward FT to a pair of Fixed Size Resamplers. Once the FFT module has returned the last transformed sample, the PV jumps to the *Process* state.
- **Process.** Here the PV asks the Fixed Size Resamplers to upsample or downsample the frequencies provided by the forward FFT. Once the Fixed Size Resamplers are done, the PV jumps to *Start Load for IFFT.*
- **Start Load for IFFT.** Here the PV reconfigures the FFT module to perform an inverse FT. The PV then waits for a *ready for data* signal from the FFT module, at which point the PV jumps to *Load for IFFT.*

- **Load for IFFT.** In this state the PV loads the FFT module with the frequencies provided by the Fixed Size Resamplers. Once the FFT module indicates it's finished with an *early done* signal, the PV jumps to *Unload from IFFT.*

- **Unload from IFFT.** Here the PV indicates to other modules with the *samples out valid* signal that the data on the *FFT out real* bus is the inverse-FT'ed samples produced by the FFT module. The Sound Router, for example, uses this data to load its post-processing buffer so the SR can continue to output a frequency-adjusted stream of samples. Once the FFT module is done outputting its inverted samples, the PV jumps back to *Pre-reset.*

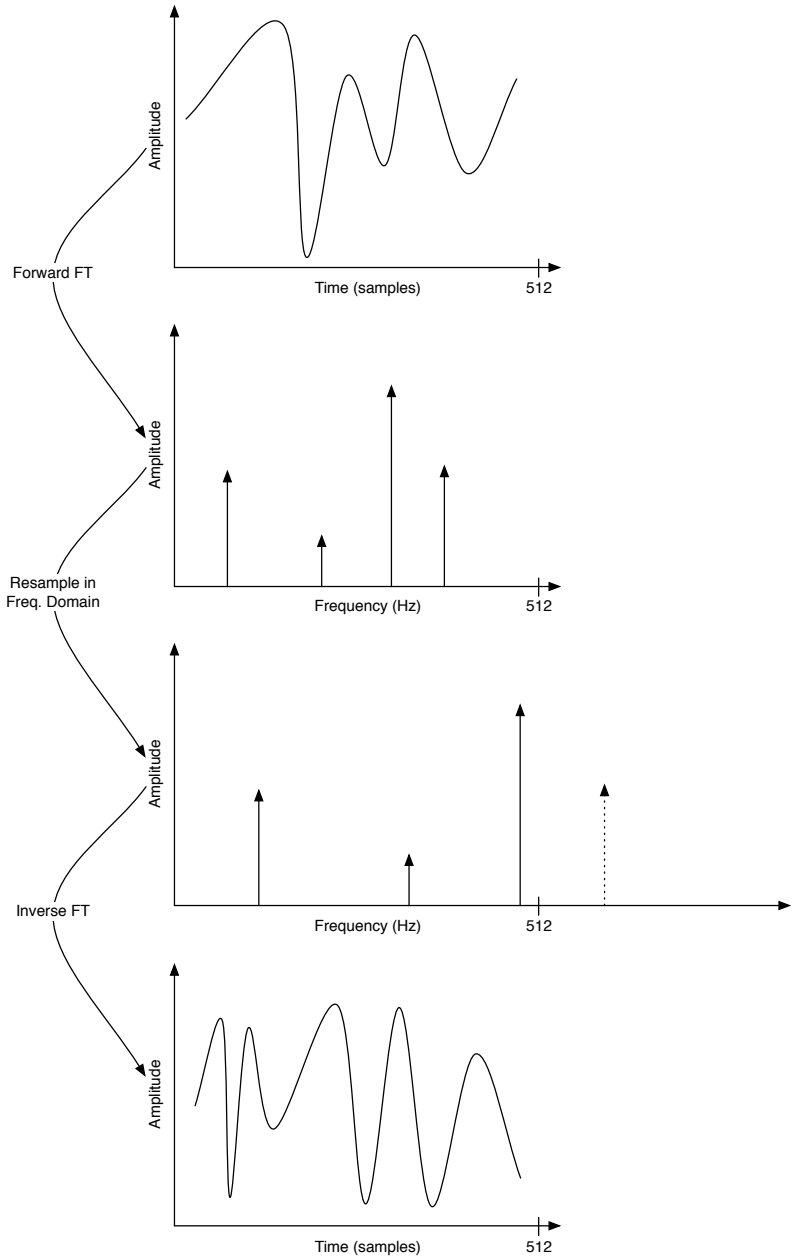


**Figure 6:** The operation of the Phase Vocoder. The input samples are Fourier-transformed, resampled in the frequency domain, and inverse Fourier-transformed to get a new time-domain signal with the scaled frequencies of the original. (The imaginary portions of the Fourier transformed samples, not shown here, are also resampled, and are incorporated into the inverse Fourier transform operation so as to stretch phases with their corresponding frequencies.) Note that if the frequencies are upsampled, the highest frequencies of the original signal may be cut off, as represented by dotted impulse in the stretched-frequency graph.

## Fixed Size Resampler (Adam Goldstein)

The Fixed Size Resampler (FSR) is the module that does the work of shifting frequencies for the Phase Vocoder. It is called "fixed size" because it turns a set of 512 input samples into a set of 512 output samples, as distinct from an infinite-length resampler like the Addresser (described later) that outputs new values forever.

The FSR uses an algorithm I developed independently

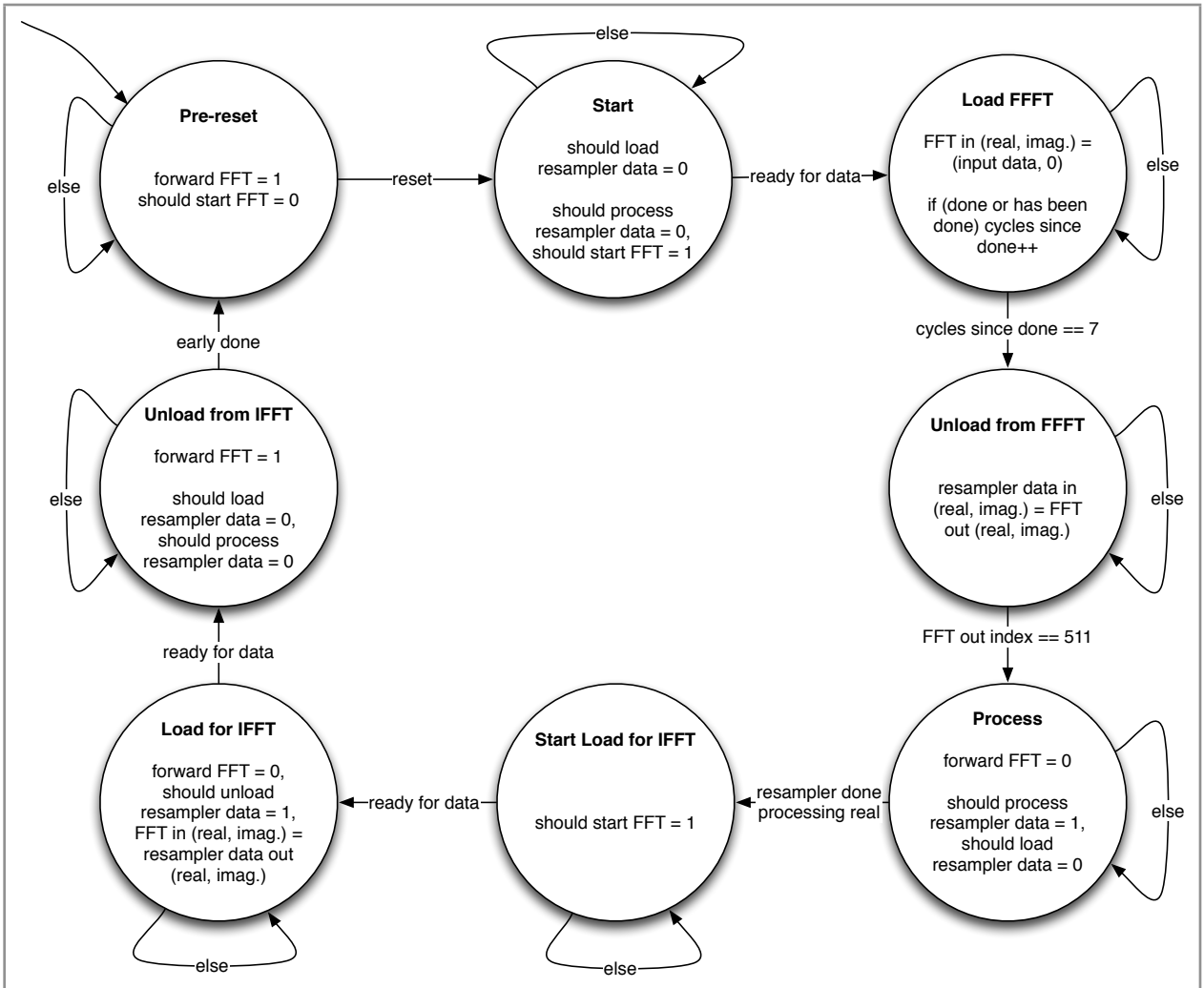


**Figure 7:** The states of the Phase Vocoder module.

(although I imagine an algorithm like it is fairly standard for doing quick-and-dirty resampling). The algorithm is somewhat different when upsampling versus downsampling; both varieties are discussed below.

### Upsampling with the Fixed Size Resampler (Adam Goldstein)

The resampler has three registers within its control: the position of the read pointer in the pre-resampling memory (*data in read pointer,* or DIRP), the position of the write pointer in the post-resampling memory (*data out write pointer,* or DOWP), and *surplus,* a signed variable (starting value 0) that keeps track of how close the resampler is to incrementing DIRP.

In addition, the resampler knows the sampling ratio (represented as a ratio of the integers *upsample* and *downsample*), as well as the difference between them (*difference*).

Figure 8 shows the algorithm in action. Each clock cycle, the resampler does the following:

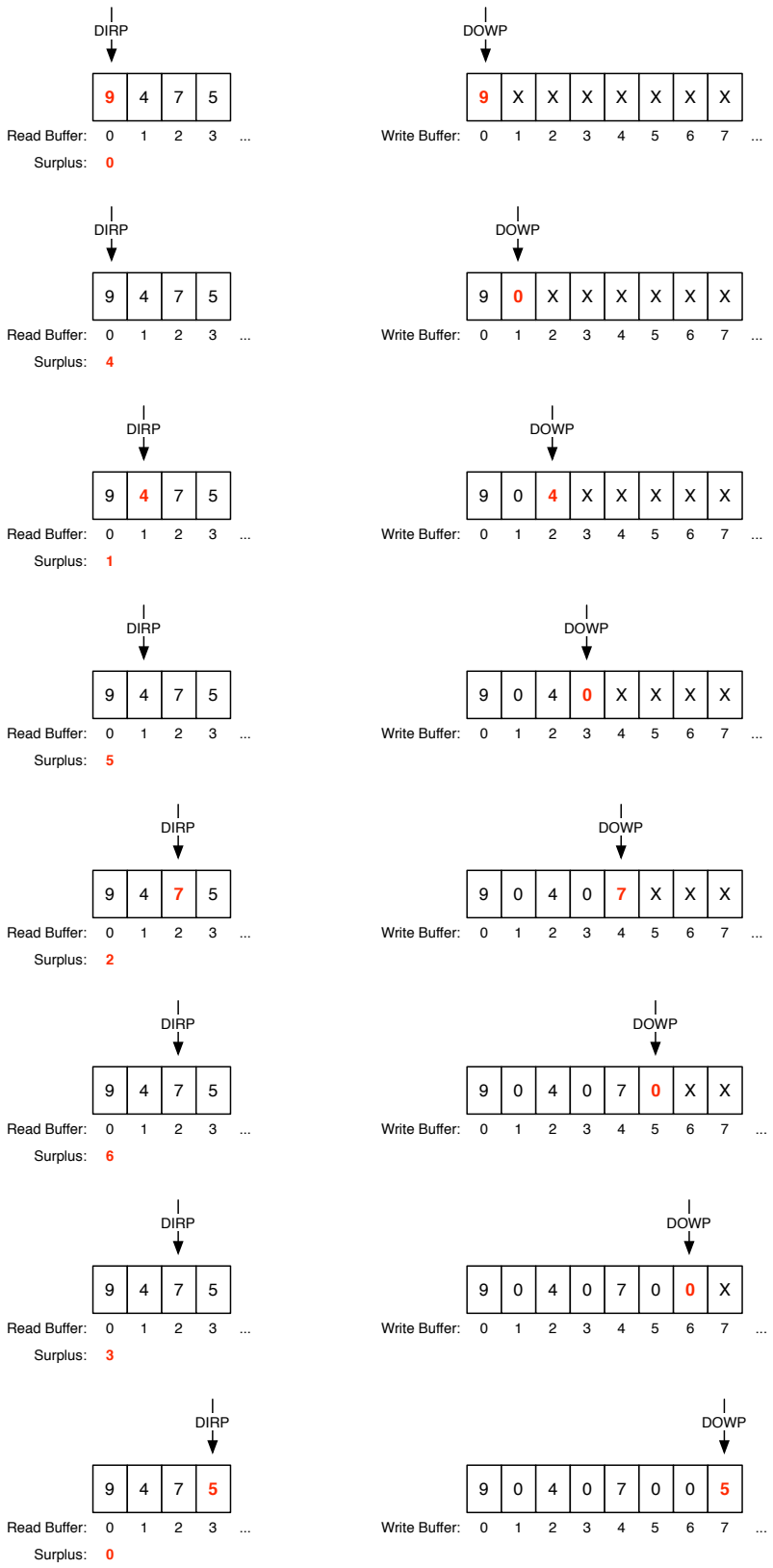


**Figure 8:** From top to bottom, the operation of the Fixed Size Resampler with a sampling ratio of 7/3. By the end, the first three items from the read buffer are stretched across the first seven slots in the write buffer.

- **Add 1 to DOWP.** The resampler is writing a set of samples serially, so each cycle it should be writing to a new output position.
- **If *surplus* < *downsample*:** The resampler writes the memory at DIRP to the memory at DOWP, and increases *surplus* by *difference.*
- **If *surplus* ≥ *downsample*:** The resampler writes 0 to the memory at DOWP, and reduces *suplus* by *downsample.*(See Design Decision 4.)
- **If 0 ≤ (the new value of *surplus*) < *downsample*:** The resampler increments DIRP by 1.

Using this method, after processing each set of *x* input values with a sampling ratio of *y/x,* the output buffer will contain the original signal stretched into *y slots.*

**Downsampling with the Fixed Size Resampler (Adam Goldstein)**

The resampler uses the same three registers when downsampling. The algorithm is different, however, as shown in Figure 9. Every clock cycle, the resampler does the following:

- **Add 1 to DIRP.** When downsampling, the resampler will by definition be writing fewer samples to output than it's reading. As a result, the resampler can always advance to the next input sample before deciding whether to write it to output or not.
- **If *surplus* < *0*:** *surplus* is increased by *upsample.*
- **If *surplus* ≥ *downsample*:** The resampler writes 0 to the memory at DOWP, and decreases *surplus* by *downsample.*

**Design Decision 4**

Originally when upsampling, I had the Fixed Size Resampler repeat each pre-resampling magnitude at *every* corresponding post-resampling frequency. For example, if the original spectrum had the value magnitude(f) at frequency *f,* and the upsampling ratio was 2, the new spectrum would have value magnitude(f) at frequency 2*f* and 2*f*+1.

The result of this, however, was that the inverse-FT'ed spectrum had many adjacent frequencies of the same magnitude, which led to some strange-sounding music. I decided instead to only repeat original magnitudes at one new frequency when upsampling, and to put 0's for the magnitude at others. In the example above, the frequenc at 2*f* would be magnitude(f), but the frequency at 2*f*+1 would be 0.

- **Otherwise, if 0 ≤ *surplus* < *downsample*:** The resampler writes the memory at DIRP to the memory at DOWP, and increases *surplus* by *difference.*

Using this method, the resampler skips writing a value to the output buffer when *surplus* is 0, which occurs 1-(*upsample/downsample)* of the time. (For example, for a sampling ratio of 3/4, the

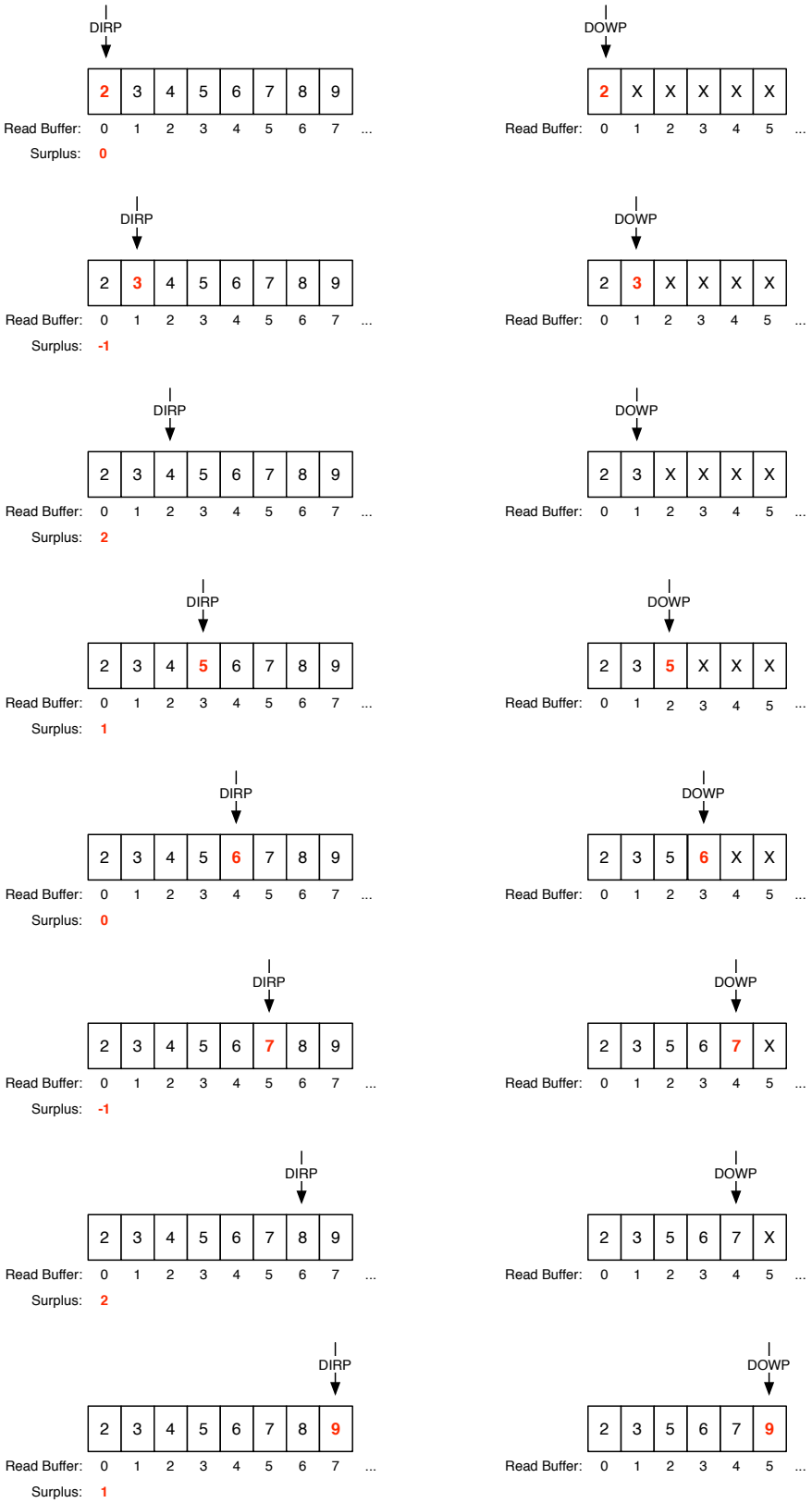


**Figure 9:** From top to bottom, the operation of the Fixed Size Resampler with a sampling ratio of 3/4. By the end, the first eight items from the read buffer are compressed into the first six slots in the write buffer.

resampler will skip writing 1/4 of input values to the output memory.)

## Memory (Nathan Artz)

The high-level memory structure is illustrated in Figure 10.

### Record Mode (Nathan Artz)

In record mode (Figure 11), data is sampled from the AC97 input jack and stored into the ZBT RAMs. Data comes in at 48 khz and then is down sampled and precision is removed; the final input is an 8-bit 6 khz sample. The down sampling is accomplished by simple storing one out of every 8 values from the AC97 input. Then, these samples are passed through a low pass filter to remove high frequency aliasing introduced in the down sampling process. After each sample is acquired, it is written into the ZBT ram using a special writing scheme we will now describe.

Each address in the ZBT Ram is 36 bits wide. Four 8-bit samples are stored at each ZBT RAM location, leaving the top four bits as 0's. Thus, a write to the ZBT ram is accomplished after four 6 khz samples are ready, and they are written into the slot such that the later samples are written into

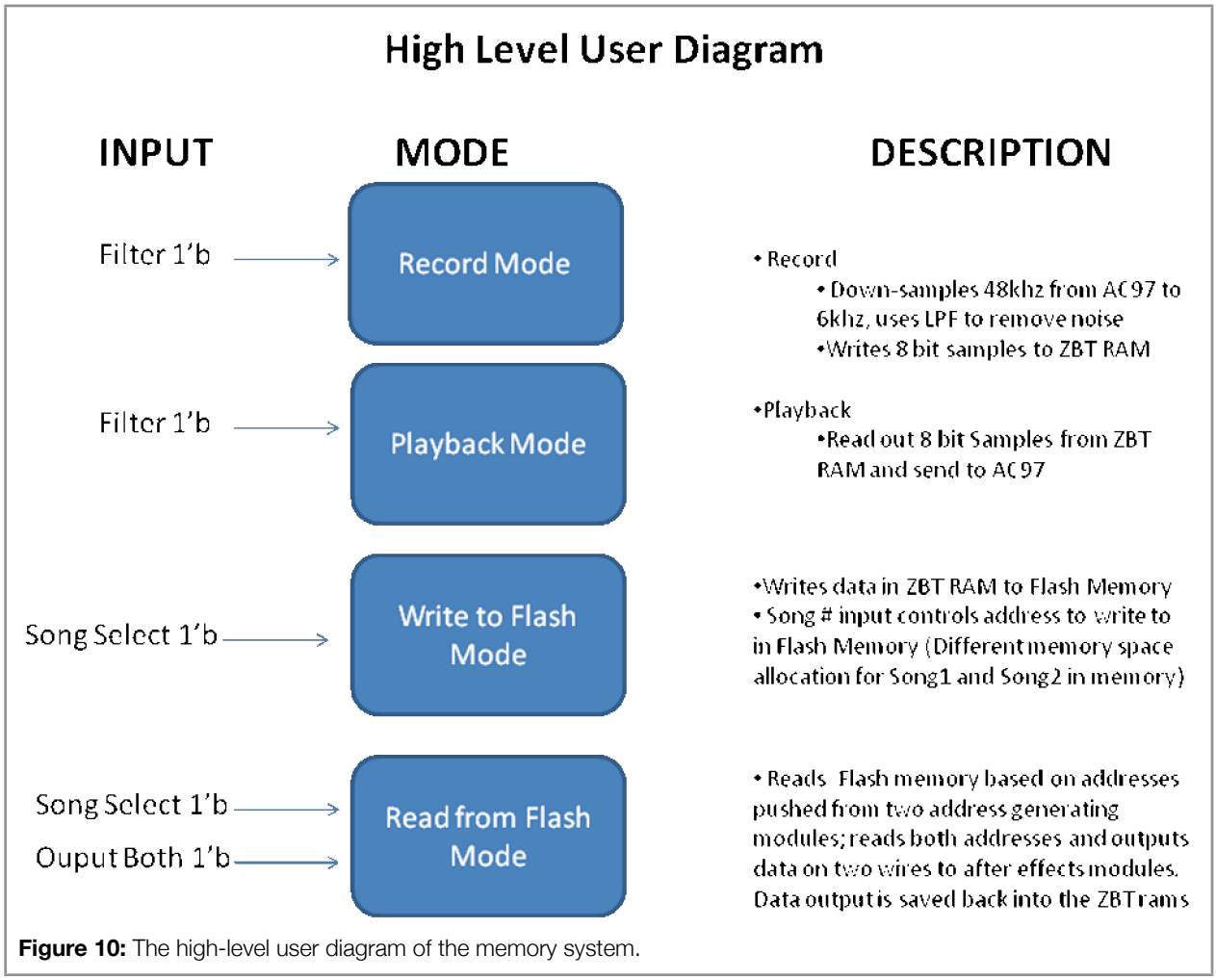


**Figure 10:** The high-level user diagram of the memory system.

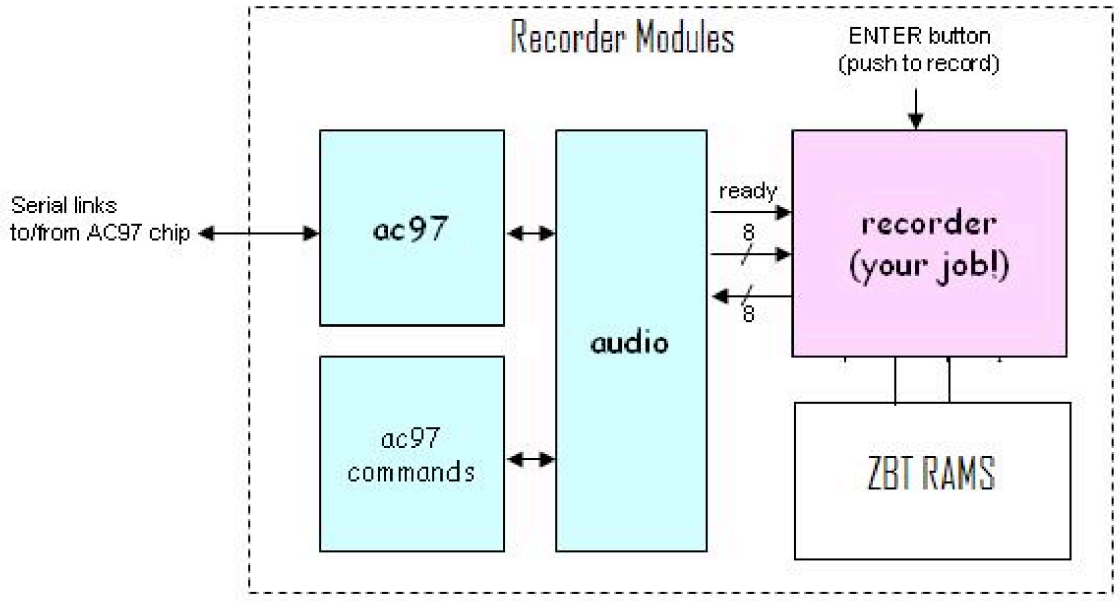


**Figure 11:** This figure describes the modules for the recorder. See Terman, http://web.mit.edu/6.111/www/f2008/index.html for more details.

the lower order bits, and, likewise, the earlier samples are written into the higher order bits. Writing to the ZBT is simply a matter of asserting *write enable* equal to 1, providing a 19 bit address, and providing 36 bits of data; the data is written to the ZBT 2 clock cycles later, as it employs a delay for maximum throughput.

The ZBT RAMs were chosen to store the samples for a number of reasons. One, unlike b-rams, they provided a lot more space for input, namely, 4 megabytes. Second, they allowed for faster compilation times (as huge b-rams weren't necessary to be built). Finally, other modules utilized b-ram space, and thus we required some of the b-rams to be free for use of other modules.

### Playback Mode (Nathan Artz)

In playback mode, data is read from the ZBT RAMs and sent to the AC97 output jack. The data operates on the 48 khz AC97 *ready* signal for output. When the no filter is employed, the data is sent 8 times every *ready,* i.e. 48 khz out. When the filter is employed, data is passed through the filter in a zero-expanded fashion and then sent to the AC97 out.

Each address in the ZBT RAMs holds four samples of 6kHz 8-bit audio (Figure 12). Playback involves reading one of these addresses four times, each time selecting the proper sample to play. The ZBT Rams reads are delayed by 2 clock cycles, and thus 2 clock cycles are given before data is sent to the AC97 out.

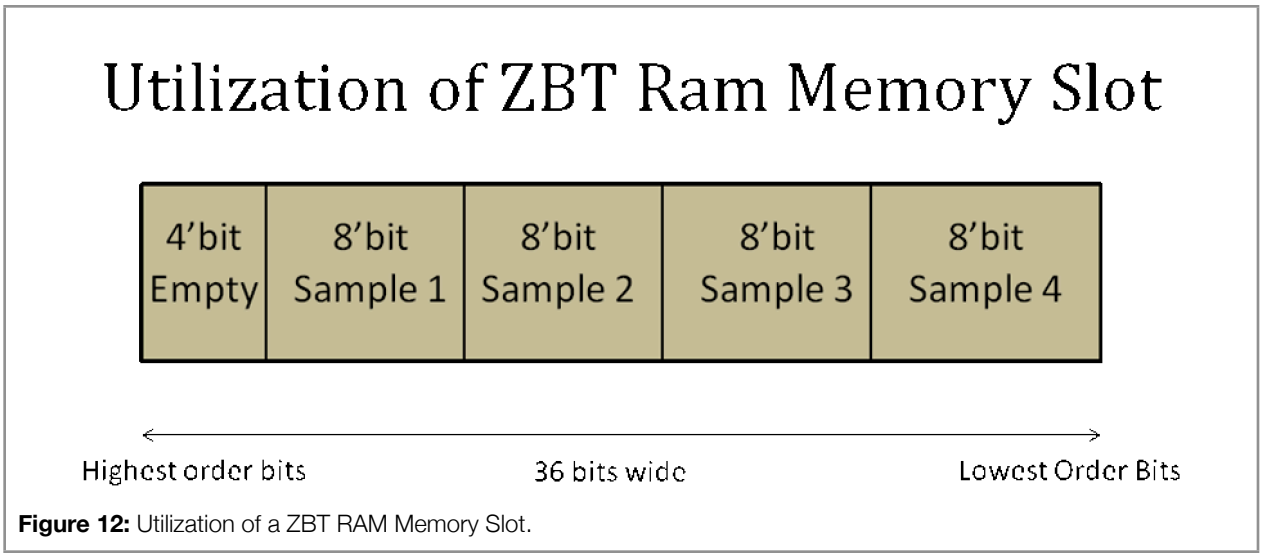


**Figure 12:** Utilization of a ZBT RAM Memory Slot.

### Write to Flash Mode (Nathan Artz)

In this mode, four steps occur: clearing of lock bits, erasing proper locations, writing to the proper locations, reading back data and outputting to ac97 for verification (Figure 13). Any data in the ZBT RAMs is copied into a specified slot in the flash memory. The Flash memory consists of 16 megabytes; the first four megabytes are allocated for song 1, the second four are allocated for

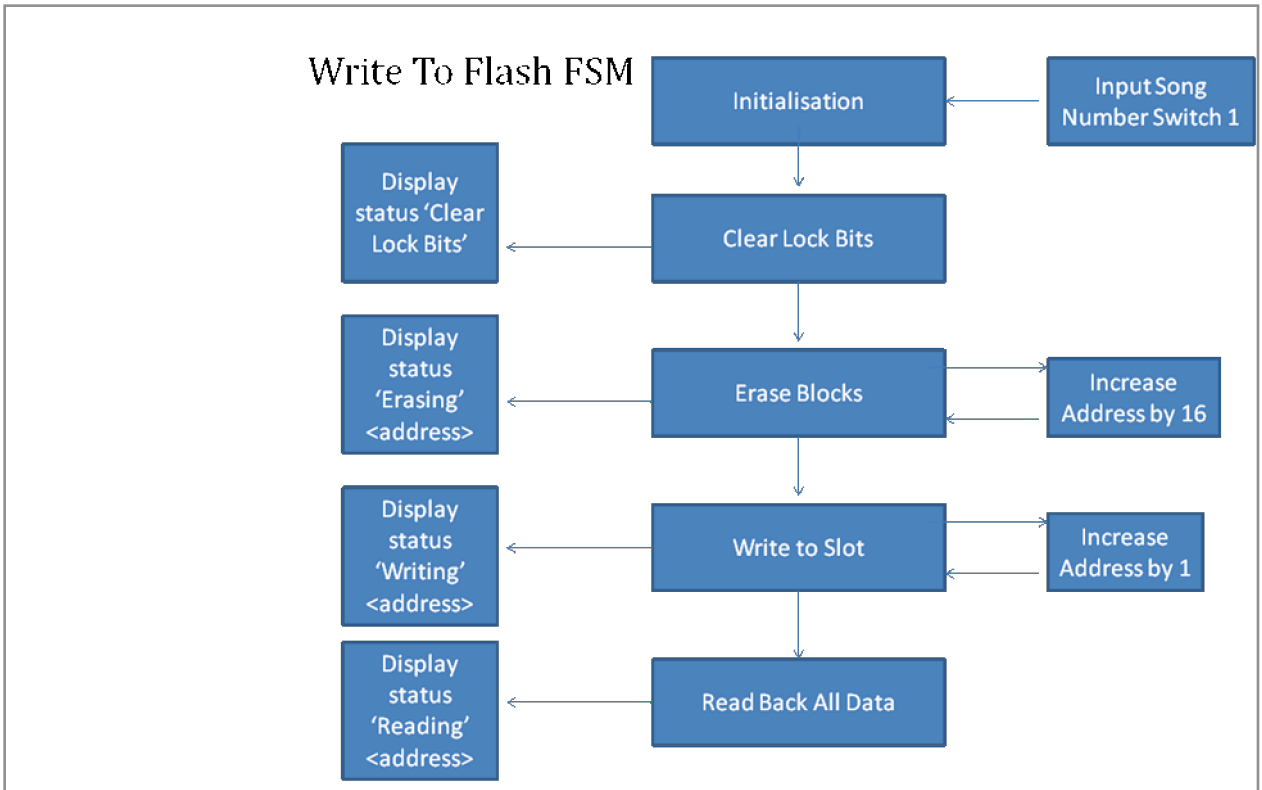


**Figure 13:** Write to Flash FSM.

song 2, and the last eight are designated for the output of the after effects (filters, pitch adjusters, etc) modules (Figure 14). Flash memory addresses are 16 bits wide.

First, the lock bits on the flash memory are cleared, allowing writing to occur. This is accomplished by issuing a few special setup commands to the flash memory (see specification for details).

The user indicates which song locations he/she wishes to write to using *switch[1]*. If *switch[1]* is 0, then address locations from 0 to (4megabytes / 16 bits - 1) will be written to, otherwise, if *switch[1]* is 1, addresses from (4 megabytes / 16bits) to (8 megabytes/16 bits – 1) will be written to, always starting with the first address. Once the user switches into write mode, the song number is immediately stored; thus, changing the song number in the middle of a write will have no effect, which keeps the operations stable.

Each Flash memory address is 16 bits wide; however, for simplicity, only one eight bit sample is stored into each flash memory location. This decision was so that playback would be easier, simply increasing the address by one to get the next sample. Thus, we can see from Figure 15 that *flash address* changes four times as quickly as *zbt address*. This is because each ZBT address is 36

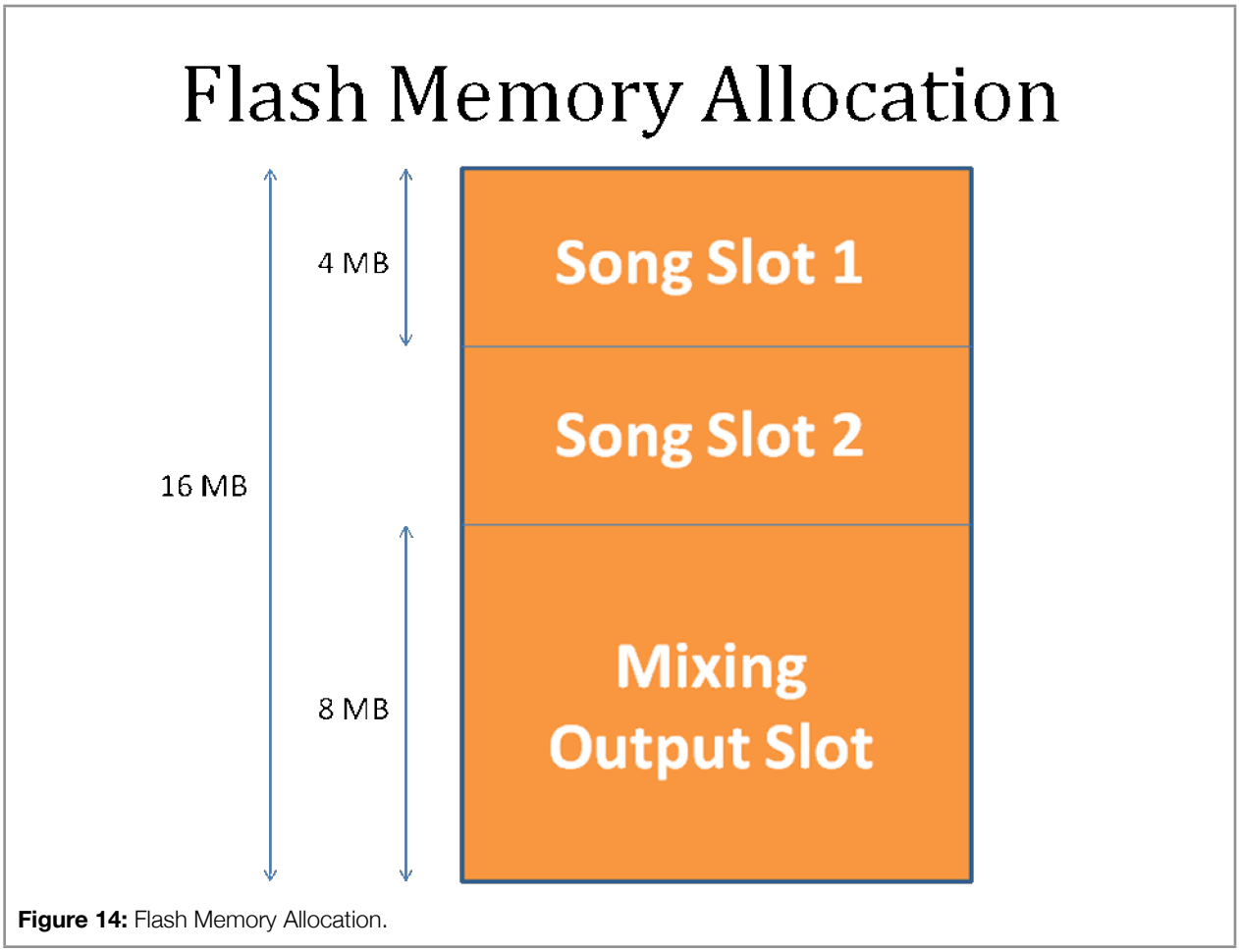


**Figure 14:** Flash Memory Allocation.

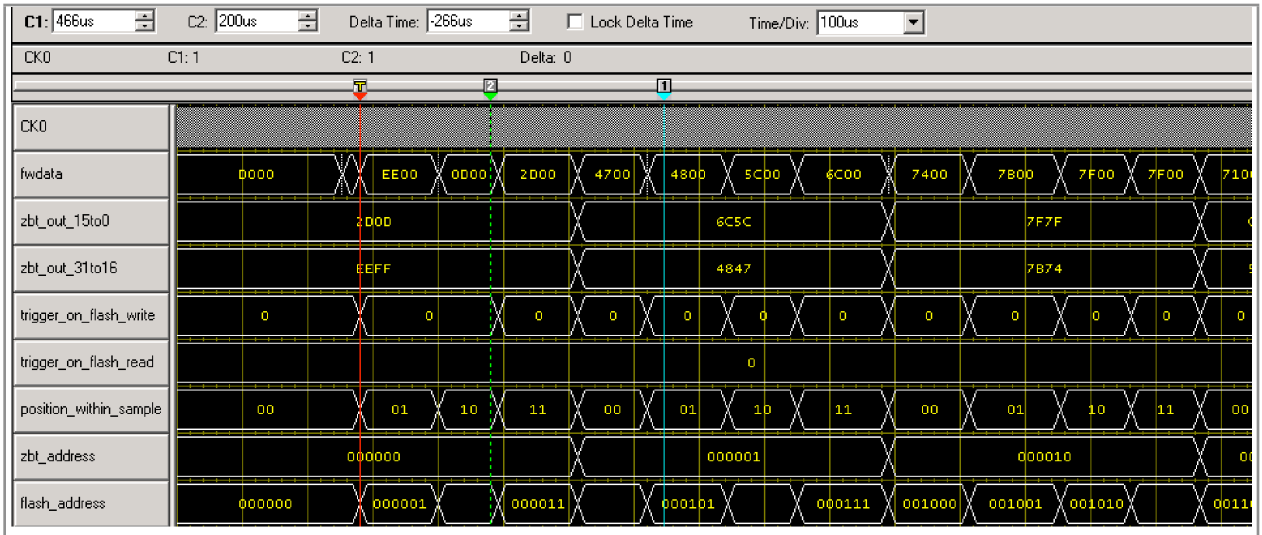


**Figure 15:** This figure is a timing diagram demonstrating the process to copy data from the ZBT Ram into the flash. First, a user switches into 'Write to Flash mode', signaled by the 'trigger on flash write' signal above next to the red line. The Flash address changes four times as fast as the ZBT address because each ZBT address holds four samples, while each flash address only holds one. Position within sample demonstrates the location within each ZBT address to read from. Fwdata is the data being written to the flash, and ZBT_outAtoB represents the data coming out of the ZBT memory from bits A to B.

bits, holding four 8-bit samples (the upper four bits are 0000), while only one 8-bit sample is stored in each flash address. *Position within sample* indicates which among the four samples in the ZBT data should be copied into the flash. We easily notice the correct operation by comparing the equivalent values of *fwdata* and of *ZBT out*, i.e. notice, for example, close to the blue line, how *zbt_out_31to16* is 4847, and this is going into the flash, first 47, then 48, then likewise with *zbt_out_15to0*.

A *highest address* is remembered after the two ZBT RAMs are written to. This highest address is used to know how much to write in the flash. This speeds up erasing, writing, and reading times. Thus, we write only to the *highest address* + *offset* in the flash memory. When we have finished writing, we start from the beginning write address and read back each address from the flash memory. Each time read data comes back, the AC97's *ready* timing signal is used to time when to send data so that the user can verify that the data was correctly written. Once finished, 'passed' is displayed on the fluorescent hexadecimal display (Figure 16).

### Dual Read from Flash Mode (Nathan Artz)

In this mode, the module inputs two address offsets from the two addresser modules, and outputs the respective data from the flash memory. The way this works is first, a *reset* pulse is sent to each addresser modules to reset their initial address offsets to zero. Next, a *6khz clk out* is 'turned on', which tells the addressers to request addresses every 6 khz. Then, the module enters into a Finite State Machine (Figure 17) that follows the following steps:

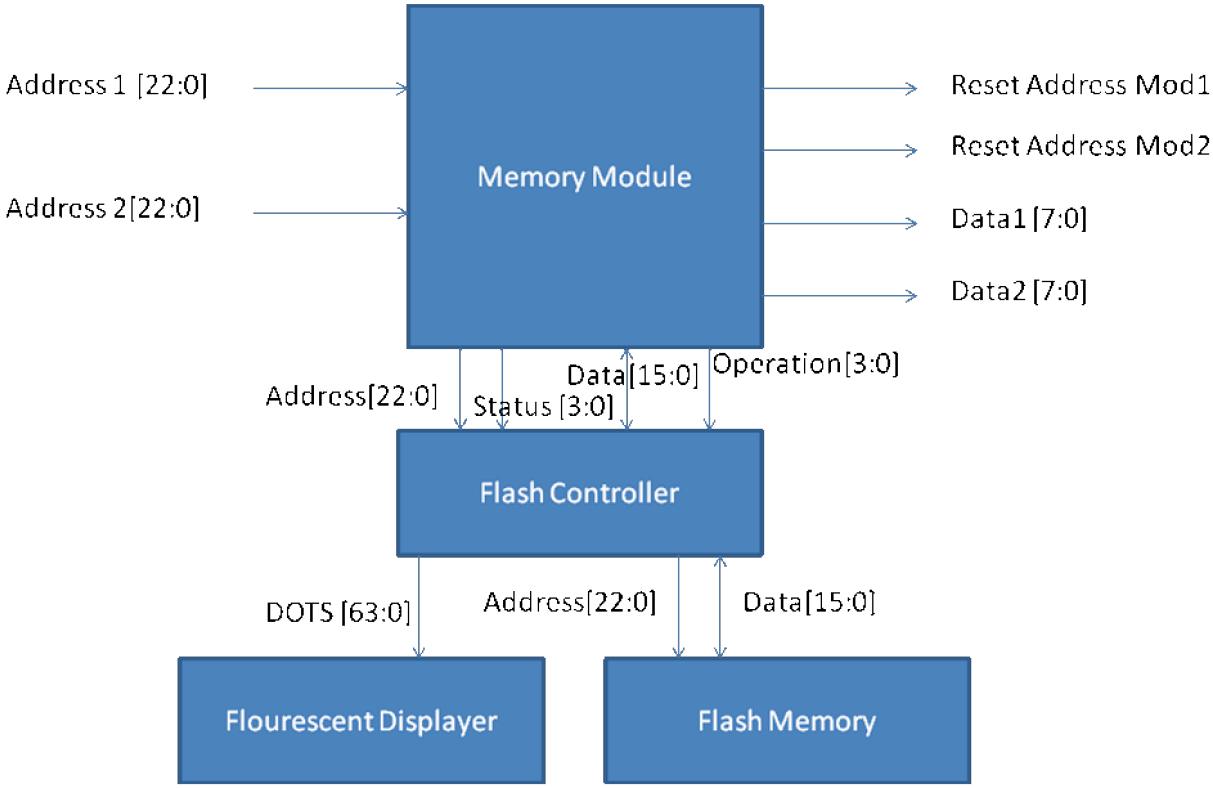


**Figure 16:** Memory Module Visualization in Write to Flash Mode.

Addresses are expected to arrive to the Memory Module one clock cycle after *6khz enable* is high. Once both addresses have been read and data is gathered, this data is sent simultaneously to each addresser modules on the same clock cycle. This data is also sent to the AC97; *switch[1]* selects data from the first addresser to be output to the AC97, however, if *switch[2]* is 1, then both outputs are sent to the AC97 out simultaneously in the form of adding the signals together – which produces the effect of two songs playing on top of each other.

This user control with the switches was done to aid in the beat matching. When beat matching, the user must be able to set the beats of the songs individually, and then listen to the beat matched output. The selection scheme with switches allows the user to do just that. Samples are sent every 6 khz, at a constant rate guided by the *6khz enable* clock. If samples need to be sent faster, this clock can be changed.

## Addresser (Adam Goldstein)

Another of the main features of the system is the ability to play songs back at different speeds. This is a feature that's useful to the Turntable Input (so spinning the disc make the song play back

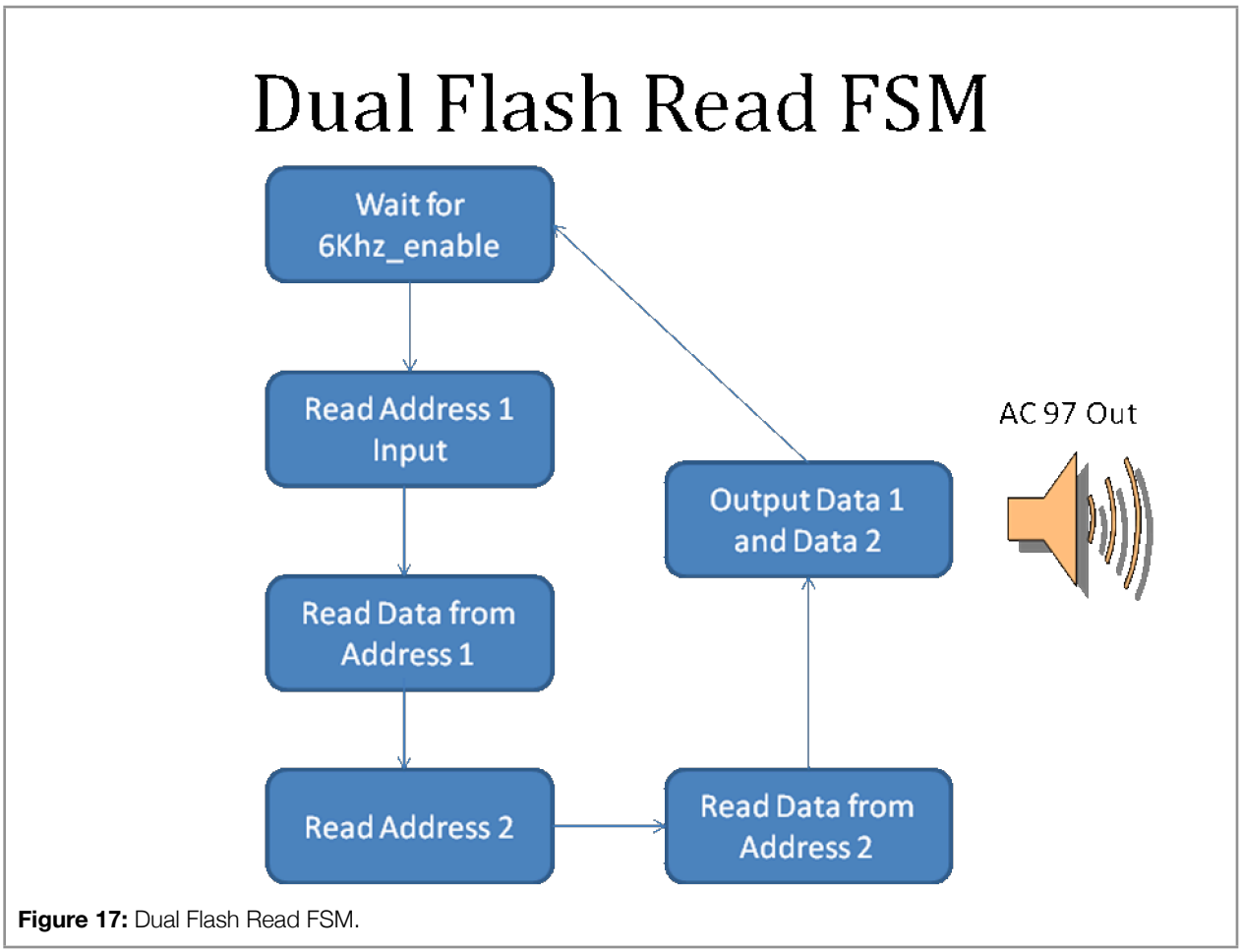


**Figure 17:** Dual Flash Read FSM.

faster) and the Beatmatcher (which slows down or speeds up the one song to match the speed of the other).

The Addresser's job is fairly straightforward: it takes a sampling ratio and a direction of playback, and, upon request, outputs the next address to read from memory. The relevant signals for the Addresser are:

- **Upsample and downsample.** These provide the desired sampling ratio, and can adjust during playback without causing the Addresser any trouble. Since the Addresser works in the time domain, providing a sampling ratio greater than one slows down the music playback and its pitch (as opposed to the Phase Vocoder, which works in the frequency domain, where providing a sampling ratio greater than one raises the pitch and keeps music playback the same).
- **Read forward.** This signal indicates whether the Addresser should be increasing (1) or decreasing (0) the addresses it provides, to read forward or backward in memory, respectively.

- **Override address and force override.** These provide a mechanism for forcing the Addresser to jump to a particular address in memory. When *force override* is asserted high, the Addresser outputs *override address* and begins providing new addresses from that address.
- **Get next address.** This input is asserted for a single cycle whenever another module wants a new address to look up. A cycle later, the new address appears on *address out*, and remains there until the the next time *get next address* is asserted.

The Addresser is different from the Fixed Size Resampler in that the Addresser must provide a new address each time one is requested, rather than resampling from a fixed-size input to a fixed-size output. As a result, the Addresser uses a slightly different algorithm from the FSR, described in detail below.

**Upsampling with the Addresser (Adam Goldstein)**

The Addresser uses a *surplus* register (just like the FSR) to keep track of when to jump to outputting the next address. Each time *get next address* is asserted, the Addresser does the following:

- **If *surplus* ≥ *downsample*:** *surplus* is reduced by *downsample.*
- **If *surplus* < *downsample*:** *surplus* is increased by *difference*.
- **If 0 ≤ (the new value of *surplus*) < *downsample*:** *output address* is incremented (or decremented, if *read forward* is 0) by 1.

An example of the values of the Addresser's signals over time are shown in Table 1.

**Downsampling with the Addresser (Adam Goldstein)**

When downsampling, the Addresser will always skip ahead by at least one address each time *get next address* is asserted. To keep track of how much to jump ahead, the Addresser uses two registers:

- **Least multiple of up to get down.** This long-named register is equivalent to *ceiling(downsample/upsample),* and is an approximate measure of how any addresses the

**Table 1:** The Addresser's signals over successive assertions of *get next address,* with a sampling ratio of *upsample/downsample* = 5/4 (so *difference* = 1) and a starting value of 47. *Read forward* is 1. Repeated addresses are in red.

| Assertion | 1st | 2nd | 3rd | 4th | 5th | 6th | 7th | 8th | 9th | 10th | 11th |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Address Out** | *47* | *48* | *49* | *50* | *51* | ***51*** | *52* | *53* | *54* | *55* | ***55*** |
| **Surplus** | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | 0 |
| **New Surplus** | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | 0 | 1 |

Addresser should advance each step. In order to avoid using a divider module, the *least multiple of up to get down* value is calculated by iterating over all possible values from smallest to largest until a satisfactory value is found each clock cycle.

- **Imperfects count.** This register keeps track of the number of times an address has been output that isn't exactly in line with the sampling ratio. For example, if the Addresser were resampling by 3/4 with a starting value of 30, the "perfect" output would be [30, 31.33, 32.67, 34], but will instead be [30, 31, 32, 34] due to rounding. The intermediate values—31 and 32—are called "imperfect" because they've been rounded. The *imperfects count* register is cumulative, and would thus be 1 when *address out* was 31 and 2 when *address out* was 32.

(The *last ref address* register is a holdover from a previous system and isn't used by the algorithm.)

The algorithm for downsampling has two varieties. If *least multiple of up to get down* is exactly equal to *downsample/upsample*, the Addresser increments *address out* by *least multiple of up to get down* each time a new address is requested. This produces precisely the expected output; for example, with a sampling ratio of 1/5 and a starting value of 40, the output would be [40, 45, . . .].

When *downsample* isn't a multiple of *upsample,* each time *get new address* is asserted:

- **If *imperfects count* = *upsample* - 1:** The output address is incremented (or decremented, if *read forward* is 0) by *downsample*. Then *imperfects count* is reset to 0.
- **Otherwise:** The output address is adjusted by *least multiple of up to get down* - 1 in the correct direction (see Design Decision 5).

An example of the downsampling algorithm is shown in Table 2.

## Beatmatcher (Adam Goldstein)

One of the coolest features of the system is the Beatmatcher, which lets the user input the beat of a "reference" song and a "floating" song. On demand, the Beatmatcher can then automatically adjust playback of the floating song to match the beat of the reference song.

The Beatmatcher is a fairly intricate system, but its operation is similar in several important ways to other modules. The components of the Beatmatcher are described below.

**Table 2:** The Addresser's signals over successive assertions of *get next address,* with a sampling ratio of *upsample/downsample* = 9/12 (so *least multiple of up to get down* = 2) and a starting value of 232. *Read forward* is 1.

| Assertion | 1st | 2nd | 3rd | 4th | 5th | 6th | 7th | 8th | 9th | 10th | 11th |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Address Out** | *232* | *233* | *234* | *235* | *236* | *237* | *238* | *239* | *240* | *244* | *245* |
| **Imperfects Count** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 0 | 1 |

**Determining the relative speed of the two beats (Adam Goldstein)**

Like the Turntable Input, the Beatmatcher must output a sampling ratio based on the speed ratio between the reference beat speed and floating beat speed. With this, the Beatmatcher can adjust the speed of the floating song to roughly match the speed of beats of the reference track.

The Beatmatcher achieves this by listening to the *pressed* input (typically connected to a button), and counting the number of cycles that elapse between button presses. The Beatmacher then keeps a running average of the last eight periods between button presses (*average cycles per period*).

Like the Turntable Input, when *set reference* is asserted high (due to a button being pressed, for example), *average cycles per period* is stored in *reference cycles per period.* This way, the Beatmatcher can compare the speed of the reference beats to the beats of the floating song ("floating beats").

**Extrapolating future beats (Adam Goldstein)**

If the Beatmatcher only changed the speed of the floating song to match the speed of the reference song, however, there would be two problems. First, the beats would likely not be aligned from the start since there was nothing to make them line up; the beats of the floating song would instead come a roughly fixed delay before or after the beats of the reference song.

Second, though, the fixed-precision of the sampling ratio would mean that the songs' speeds would likely not be perfectly aligned. As a result, the floating song's beats would shift over time with

**Design Decision 5**

Why does the Addresser sometimes increment the address by *least multiple of up to get down* - 1, instead of just *least multiple of up to get down*? The examples below illustrate.

There are two ways of producing new addresses for a sampling ratio of, for example, 7/9, if for simplicity the goal is to make the algorithm only have to increment by one of two possible values.

One approach is to increment the address by 1 for 6 consecutive requests, and then increment by 3 on the seventh request (e.g. [{19,} 20, 21, 22, 23, 24, 25, 26, 27, 28] → [20, 21, 22, 23, 24, 25, 28]).

The other approach is to increment the address by 2 for 6 consecutive requests, and then decrement by 2 on the seventh request (e.g. [{19,} 20, 21, 22, 23, 24, 25, 26, 27, 28] → [20, 22, 24, 26, 28, 30, 28]).

Although the second method stays closer to the "perfect" values for this resampling ratio and requires a smaller adjustment at the end (decrementing by 2 instead of incrementing by 3), I decided against using it (or choosing between the two methods based on which was closer to perfect) because the second method can require jumping *back* in addresses, which would sound like unwanted skipping. This is why the Addresser jumps by *least multiple of up to get down* - 1 rather than *least multiple of up to get down.*

respect to the reference beats, making the beat matching even less effective.

To solve these two problems, the Beatmatcher predicts future beats for each song based on the beats that have already been entered. Then, each reference beat, the Beatmatcher forces the floating song to jump to the appropriate address to makes sure the two songs are aligned again.

The Beatmatcher calculates future beats by counting from 0 up to *average cyles per period* and *reference cycles per period.* Each time the Beatmatcher finishes a count, it inverts the appropriate signal: either *floating beat* or *reference beats,* respectively.

**Doing the matching (Adam Goldstein)**

When *should start matching* is asserted high, the Beatmatcher initiates the processes necessary to match the beats. The sampling ratio is constantly being output on the *upsample* and *downsample* busses (which feed directly into the Addresser for the floating song). Therefore, the only signals the Beatmatcher needs to generate separately are to explicitly override the address of the Addresser to make the floating song jump to a particular sample. The relevant outputs are *force floating override* and *override floating address,* which connect directly to the Addresser's *force override* and *override address* inputs (see above).

These signals change as part of a state machine, shown in Figure 18. The states are as follows:

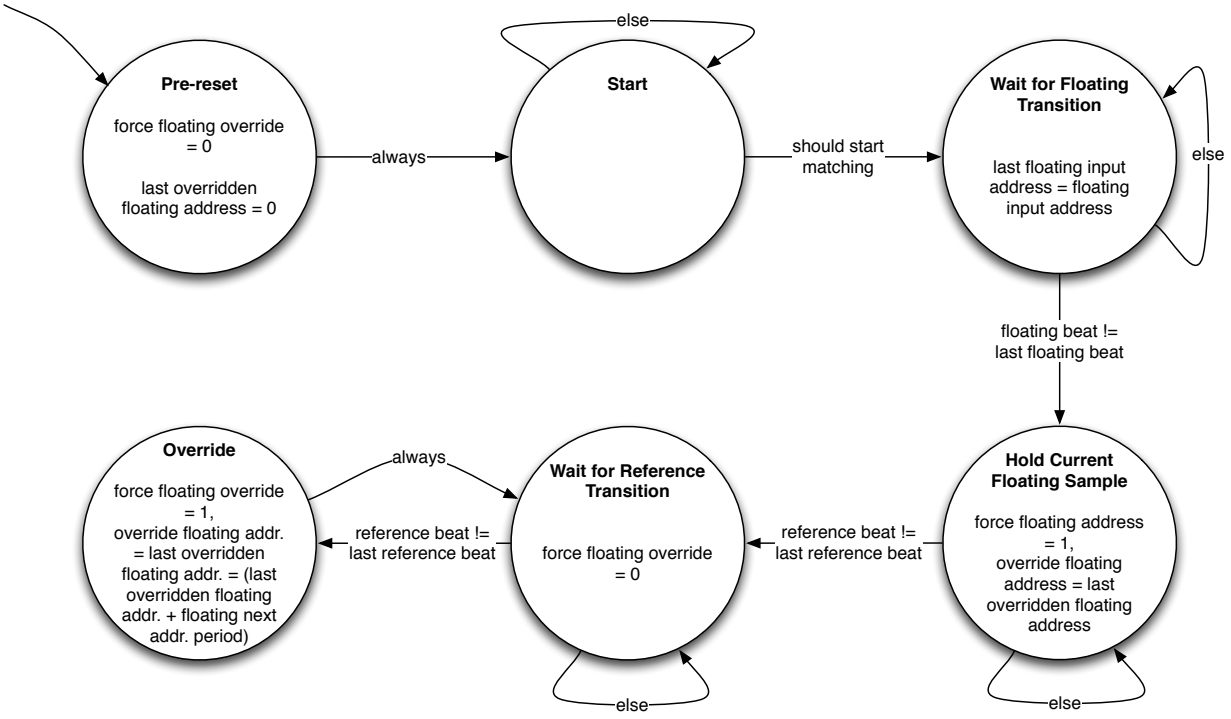


**Figure 18:** The states of the Beatmatcher module.

- **Pre-reset.** Here the Beatmatcher sets *force floating override* to 0, to ensure that the Addresser can play back music normally. Additionally, the Beatmatcher initializes *last overridden floating address* with 0; later, this register will keep track of the playback address where the floating song last jumped to match the beat.
- **Start.** This state exists to wait for the user to assert *should start matching,* at which point the module jumps to Wait for *Wait for Floating Transition.*
- **Wait for Floating Transition.** In this state, the Beatmatcher is waiting for a beat from the floating song to happen. Once one occurs, the Beatmatcher jumps to *Hold Current Floating Sample.* In the meantime, the Beatmatcher set the *last floating input address* register to the current address coming out of the Addresser; this register stores the address of the last floating beat in memory.
- **Hold Current Floating Transition.** Here, the floating song has just had a beat. In order to line up the beats of the two songs, this state forces the floating song's Addresser to hold its current value—that is, to pause. Once the reference song has a beat, the Beatmatcher jumps to *Wait for Reference Transition.*
- **Wait for Reference Transition.** In this state, the Beatmatcher withdraws its override of the floating song's Addresser, since the beats are now lined up. The Addresser then continues playback normally (at the speed provided by *upsample* and *downsample*), until another reference beat occurs, at which point the Beatmatcher jumps to *Override.*
- **Override.** In this state, occupied for a single clock cycle, the Beatmatcher has just noticed a reference beat, so it instructs the floating song's Addresser to jump to the address of its next beat by adding the address of the last floating beat (*last overridden floating address*) to the number of addresses the music system has requested between floating beats (*floating next address period*).

The result of this process is that, if the two songs are of constant (but potentially different) speeds, and the reference and floating beats are input perfectly aligned with the beats of the songs, the Beatmatcher will slow down or speed up the floating song to the closest it can to match it to the speed of the reference song, and ensure that even if they come slightly out of alignment, they'll be realigned every reference beat.

## Audio Effects (Matthew Putnam)

Two audio effects were designed to work with audio of all sampling rates: echo and reverb. Both effects are described below.

### Echo (Matthew Putnam)

The echo effect is a simple repeat of past information that is added on top of the current audio sample. The module takes as input the incoming sample and *ready* signal and outputs a new audio sample with the echo effect added as well as the new *ready* signal for that sample.

This module works by creating a large circular BRAM buffer and keeping an offset pointer to the next location to write. When a new sample is ready, this sample is written to *buffer[offset]* and the offset is incremented. Then, the sample at *buffer[offset-delay]* is read, and this stored sample is added to the incoming sample. To prevent overflow, this sum is stored in a 9-bit register, and the high 8 bits are passed to the output.

Since this module uses a lot of storage, an external BRAM module was used. This created some timing issues with the BRAM's parameters. To prevent these issues, a small linear FSM was used to control the stage of the calculation. The FSM was overly cautious, only doing one computation per state, but since only a few computations are done per sample, and there are thousands of clock cycles between samples, this excess was not a problem.

When the new sample is calculated, the new *ready* signal is asserted for one clock cycle. It's not entirely necessary for subsequent modules along the data path to use the new *ready* signal, as using the old one will only mean that the samples being read are one sample old. This only affects the timing of the signal, not any computation being done on it, an is an unnoticeable delay.

### Reverb (Matthew Putnam)

The reverb effect took many tries and many different approaches to get right. The initial idea was to use convolution reverb using the same architecture as the FIR filter module from lab 4. The module would convolve the incoming samples with the impulse response of a resonant room, recreating what playing that sound in that room would sound like. This design has the promise of perfectly mimicking the acoustic properties of any room, but is impractical because of the amount of data needed. For it to work properly, a very high sampling rate is needed, and the audio samples need to have enough bits to be accurately scaled. Neither of these were available.

The next idea was to approximate the impulse response as a series of impulses separated widely by zeroes. This would drastically reduce the number of samples and arithmetic operations needed at the expense of quality. For this, a BRAM was created that stored the last few thousand samples. However, figuring out the correct delay between impulses and tuning the pulses' magnitudes was incredibly difficult. If the pulses were just a little too large, the output was a distorted, robotic sound. If they were too low, then the reverb couldn't be heard.

The third idea was to use feedback. For this, two basic building blocks were created. The first was a comb filter, which delays the sample by some constant amount and has a feedback loop that adds a scaled copy of the output back into the input. This has the effect of sound bouncing off a single surface. The second module was an allpass filter, which is the same as the comb filter, but includes an additional feed-forward loop with the negative of the gain from the feedback loop. The effect of this module is that it smoothes the frequencies of the input. Various reverberators could then be made by adding together several comb filter echoes and smoothing the sum with allpass

filters. However, the problem with this approach was that because of digital logic's discrete nature, the fractional feedback component continuously lost precision (and compounded the problem when a truncated signal was further truncated), introducing lots of noise.

The final decision was to go back to the second idea, the reduced form of the convolution reverb. A few days were taken to carefully calculate the ideal parameters, and the result was as good as 8-bit, 6kHz (48kbps) audio is going to get. The signal is slightly distorted, but the reverberation is clear.

The module utilizes two ROMs, one storing a list of delays (measured in number of samples), the other storing a list of scaling factors as fractions of 1024. There is an index into these ROMs, and an 18-bit accumulator. When a new sample is ready, it is written to the buffer at the current offset, the offset is incremented, and the index and accumulator are zeroed. Then, on every clock cycle, the following things happen:

- **The delay and scale are read out of the ROM using the current index**
- **The sample at *buffer[offset-delay]* is read and multiplied by the scale,** and this product is added to the accumulator
- **The index is incremented**

When the index reaches the end of the samples, this process is halted, and *accumulator[17:10]* is passed to the output. Note that for the output to be in the right bits, the scale factors need to add up to 1024, or at least very close to it. As with the echo module, a new ready signal is asserted for one clock cycle.

## Testing and Debugging

We tested most modules of the Digital DJ Setup independently before integrating them into the completed system. The system was loaded onto a Xilinx FPGA as part of a Labkit with a 16-character hexadecimal display, making it possible to examine the states of various modules. The hardware constraint file and top-level module code for connecting inputs and outputs to the Digital DJ Setup were re-used from a past project.

Below are descriptions of the testing performed on each subsystem.

### Input Methods (Adam Goldstein)

Testing the turntable and knob was very easy; both worked the first time I downloaded them to the FPGA and connected wires to the appropriate *user* inputs.

### Keyboard (Matthew Putnam)

The keyboard input module was tested by displaying the current ASCII value, the difference (in number of half steps) between the two keys pressed, and the resulting ratio on the 16-digit hex display and visually verifying those values to be correct.

## Sound Router (Adam Goldstein)

Testing of the Sound Router was by far the most time-consuming part of my work on the project. Originally, I ran into problems with the timing of the FFT module; because I was not waiting for the requisite seven cycles after receiving a *done* signal from the FFT module, I would end up with frequencies out of alignment, producing strange overtones (Figure 19).

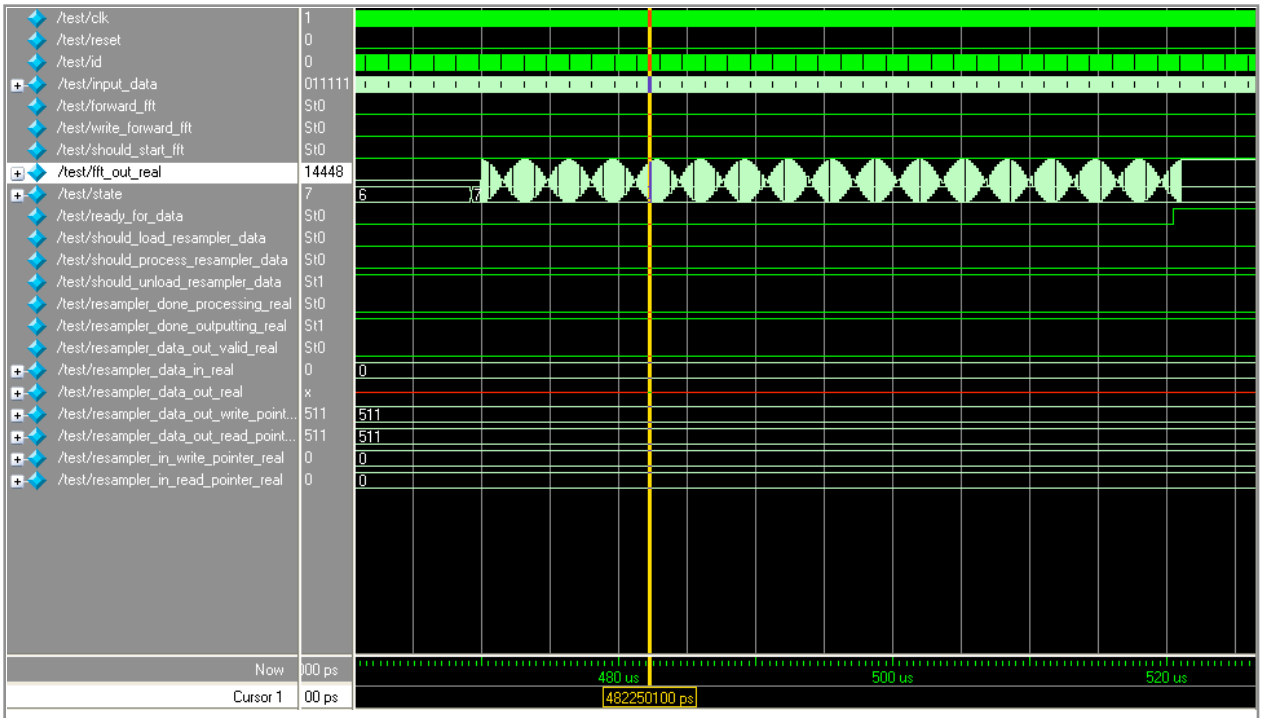


**Figure 19:** During this test, I was inputting a signal that oscillated from 16'b1000_0000_0000_0000 to 16'b0111_1111_1111_1111 (signed) every clock cycle, then forward FT'ing it and inverse FT'ing the result. I expected to recover the original signal, but because I was not waiting for enough cycles before unloading the FFT module, frequencies got shifted, leading to the lower-frequency oscillation pictured here.

Another time-consuming problem was the fact that the FIFO provided by Prof. Terman worked differently with the FFT module in ModelSim and on the logic analyzer. After several days of attempted diagnoses by the lab assistants, I gave up and wrote the memories myself.

Finally, I ran into a bizarre problem with ISE, where trying to synthesize the FSR module as originally written would eat up more than 50% of the chip's resources by allocating memories as registers instead of LUTs, and lead to an order-of-magnitude slowdown in compile time. I was able to fix the problem by replacing:

```
if ((surplus > 0) && (surplus >= downsample)) begin
      data_out_store[data_out_write_pointer] <= 0;
      ...
else if (surplus < 0) begin // We're downsampling
      ...
```

```
end
else begin
     data_out_store[data_out_write_pointer] <=
data_in_store[data_in_read_pointer];
     ...
end
```

With the following, equivalent expression:

```
if ((surplus > 0) && (surplus >= downsample)) begin
     // data_out_store[data_out_write_pointer] <= 0;
     ...
else if (surplus < 0) begin // We're downsampling
     ...
end
else begin
     // data_out_store[data_out_write_pointer] <=
data_in_store[data_in_read_pointer];
     ...
end
if (surplus >= 0)
     data_out_store[data_out_write_pointer] <= (((surplus > 0) &&
(surplus >= downsample)) ? 0 : data_in_store[data_in_read_pointer]);
```

Once I made this change, the improved compile time made it much easier to quickly test Sound Router features that were working in ModelSim on the Labkit itself.

### Addresser (Adam Goldstein)

Testing the Addresser in ModelSim was very easy; I simply provided it a sampling ratio and a signal for *get next address,* and after a few minor tweaks got the output shown in Figure 20.

### Beatmatcher (Adam Goldstein)

To test the Beatmatcher in ModelSim, I had to generate signals representing two different speeds of someone pressing the button—one for the reference speed and one for the floating speed. When I set them at a ratio of approximately 4 to 3, I got the output shown in Figure 21. Note that as expected, once *should start matching* is asserted high, the system waits for the next transition of *floating beat,* then forces an override of the output address until the next transition in *reference beat.* The sampling ratio (12/9) is constantly provided so the Addresser knows what speed to play at, and at reference beats, the override mechanism forces a new address into the Addresser for a single cycle (far right of Figure 21).

### Memory (Nathan Artz)

Please see the main Memory section.

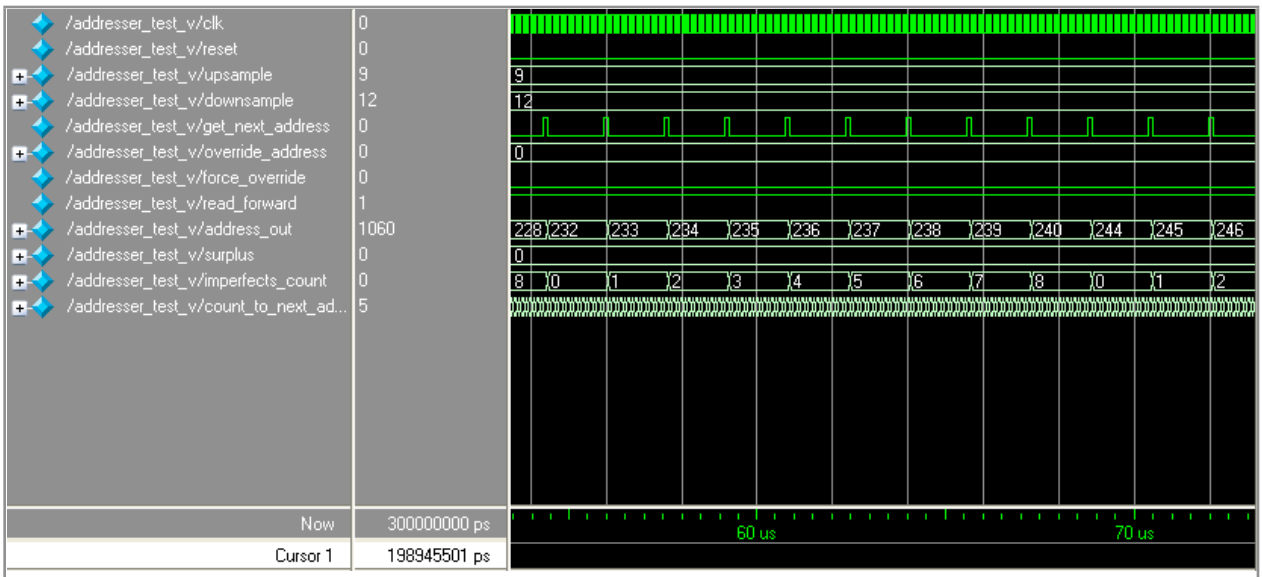


**Figure 20:** The working output of the Addresser in ModelSim. Note that each assertion of *get next address* produces a new *address out* value as expected.

## Audio Effects (Matthew Putnam)

The memory components of the echo and reverb modules were tested by running a test file on them in ModelSim. For the test, some inputs were given and the signal waveforms were inspected to see that the values were being stored at the correct locations. Upon this testing, some minor timing bugs were found. These were caused because some internal registers were not zeroed on reset. While the FPGA automatically zeroes out all registers on initialization, they were explicitly set to be safe.

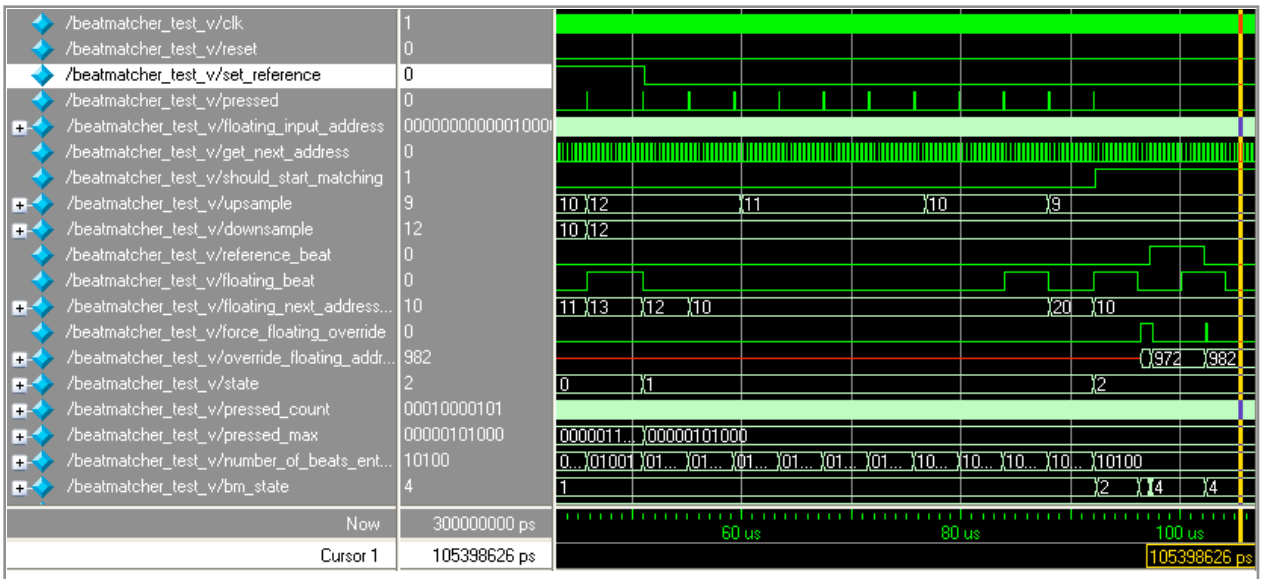


**Figure 21:** The working output of the Beatmatcher in ModelSim.

The echo and reverb effects themselves were tested by using a modified version of lab 4 to input and output actual audio samples through the AC97. We could then simply talk into the microphone to observe the effect. The biggest issue was with tuning the scale values of the reverb module. First, they have to add to very near 1024 so that the output is the right size. Second, they have to decay with the right shape so that the effect sounds authentic. The delay values need to be close enough together that the listener can't hear a discernible set of echoes, and far enough apart that the effect had noticeable sustain. Also, if the impulses were very close together, the adding in of signals at regular intervals would produce noise at a certain frequency. Usually, this signal had very little power, but if too many impulses were used, this started to be heard.

# Conclusions

The mixing setup features a robust memory foundation, specialized mixing algorithms, and innovative physical mixing components.

At a technical level, the memory modules are based on a series of different finite state machines, smoothly reading and writing based on user input of different switch selections. Moreover, the components were extremely successful in warping sounds in both the time and frequency domain.

First, a computer keyboard input, modeled as inputs to a piano, provided a unique method for discretized warping based on the distance between different ‘notes’. Second, a rotating motor allowed for continuous time and frequency changing via changes in speed, providing an excellent “scratching” sound.

While lower quality than the maximum 48kHz sound rates available, 6kHz samples permitted spectacular output from the echo and reverb modules decaying sound output. Integration of the components, especially the beat matcher and address modules was highly successful due to highly decoupled modules, which only were connected by two busses. Finally, the beat matching module was the climax of the project by matching user-inputted beats to play two songs synchronously  at the same beat.

Initially, problems with writing the Flash memory on the fly occurred, and a revision in design was created to use the ZBT as an intermediate. As stated, highly decoupled modules allowed for easier integration in the long run.

The design could be improved in a number of ways. First, improving the sound quality through both a higher sampling rate and higher precision in each sample would allow for a much cleaner output. Second, possibly utilizing more b-rams in the reverb module would allow it to extend further in time. Third, physical components such as the turntable could be implemented with a smoother motor to improve the quality and consistency of playback.

### Acknowledgements

We would like to extend our sincerest thanks to Gim, Alex, and Ben for their remarkable helpfulness during our numerous brainstorming and bug-diagnosing sessions.