Auralization of the Visual World

Clark Della Silva, Gabriel Karpman, Adam Suhl

November 5th, 2012

Abstract

The goal of this project is to produce a system that produces some form of audio representation for the visual world. The system will take a video input, process it, and use various aspects of the data to control a music synthesis system. The visual field will be split into a number of vertical channels, and each channel will be processed independently. Then the results of processing these channels will be fed to a sequencer that will send information to the audio system one channel at a time, as well as implement recording and playback options. The information output by the sequencer will then be fed into a multichannel audio synthesizer where the number of oscillators, pitch, harmonic content, waveform, and amplitude of the sound will be determined by the processed video data.

Overview

Our project aims to explore the interplay between audible and visual information, by using video capture as a controlling medium for sound creation. In particular, we will process information from an NTSC XVGA camera, as well as hardware user controls, to produce a dynamic soundscape. This production will consist of two major types of interaction.

1. Ambient analysis of a visual field: we subdivide an image into multiple vertical bands. Each of these vertical bands will be subdivided into 16 horizontal bands, making a grid out of the image overall. Each of these grid sections will then be analyzed to extract HSV peak information, and the 16 sections of each vertical band will be used to control a parametric synthesizer. The vertical band which is controlling the synthesizer will cycle across the image at regular intervals, so that each band defines one polyphonic note of a musical phrase.

   Information about the rate of change of the bands will also be stored, and used to control effects on the overall sound, allowing a reverberant and expansive soundscape for slowly-varying scenes, and a tight, crisp sound for rapidly changing scenes.

2. Analysis of intention in the visual field: we will apply blob detection --- most likely based on the scaled difference of gaussians approach --- to identify light sources (LEDs) of pre selected hue. This information will be used to produce control objects which respond to either the position or motion of blobs including, in particular, controls which trigger when an LED is moved across a pre-defined line in one particular direction. These will be used to control a sample playback style drum machine, allowing the user to overlay intentioned and structured rhythmic content onto the passively controlled sounds described above.

The interface between the control set extracted from the video feed, and the sound generating components, will be through a step sequencer and looper, which will allow the user to record sequences of either the ambient or intentioned streams, loop them, and rearrange them into different patterns. The soundstream produced will be passed through a final layer of modular filters, gain controls, mixers, and effects, before being passed to an AC’97 based output stage for playback.

To alleviate the overall complexity of the project, the flow of information into and out of each subunit will be abstracted to a substantial degree. This will allow us to test each unit fully in isolation, and will provide an extreme degree of overall design flexibility --- as the subset of the available functionality that is actually desired in any given instantiation may be selected at runtime, and alternate versions of any given module --- or external devices providing the same interface --- may be substituted without having to modify any other components.

Primary Modules

The system will consist of four major components, which will be self contained and able to function, given suitable input and output, wholly independent of the others. Further sub-divisions of these modules will be explained below. These systems are:

Video processing unit (VPU). This unit will take input from the video camera, and extract information about the chromatic content of each region of the image, as well as about the presence of easily identifiable blobs (LEDs) and their motion.

Control unit (CU). This unit will take in the outputs of the VPU, as well as hardware user inputs, and will generate control parameters for the sequencing and audio generation systems. This isolates all user input from those systems, and makes the overall structure easily expandable --- for example, adding computer control, midi, Open Sound Control or other interface would require only adding support in the CU: These changes would be transparent to the entire rest of the system. We will initially be implementing a selection of hardware controls using the User IO pins, buttons, and PS/2 connector on the labkit. If time allows, we may also integrate midi control to allow the use of professionally produced music controllers.

Sequencer. This unit will record patterns containing parameters for the synthesizer and triggers for the sampler playback units. It will allow playback from patterns as they are recorded, as well as playback, re-arrangement and resequencing of pre-recorded patterns, allowing the incremental production of more complex and layered pieces. The format that synthesizer parameters will take will be that of a frequency response (i.e., the Fourier Transform of the desired signal,) allowing the specification of multiple notes and of their harmonic contents directly and efficiently. This unit will also contain a parameterized reverb effect unit, to take advantage of the fact that the frequency responses --- as generated --- will have a fairly sparse support, and so will be able to be stored and manipulated efficiently within the sequencer.

Audio generation and effects. This unit will produce sounds specified by the sequencer in the time domain. This takes two forms: parametric synthesis in the form of an IFFT and Hilbert Transformer, and playback of recorded samples. The audio streams will then be mixed together to form channels, which will be passed to the effects unit. This unit will use a patch-bay like interface to connect channels --- both inputs from the sound sources and outputs of effects units --- to the inputs of effects units, allowing the construction of mixed and layered outputs. Sound will be passed out of the effects unit and into the AC’97 based output stage by patching to a particular two-input effects unit which will serve as stereo output.

To promote modularity and ease of routing between these major components, data will be passed on word-serial busses. Each bus will contain wires for

- WOE (1 bit), high when the receiver is ready to accept new data (has already read what is on the bus.)
- RE (1 bit), high for one clock when new data is written. (high continually during burst writes.) If WOE is high and new data is not being written, RE should be low to prevent the data already on the bus from being read twice.
- Start (1 bit), high for the first clock of each packet. Resets the receiving FSM.
- Data (1 word) Contains data words being transferred. The first word should be available when RE goes high, and the word should not change once RE has gone high until WOE is expressed.

While the required system-to-system bandwidth will generally be several orders of magnitude lower than that available on these buses, we intend to provide a distinct bus for each channel of communication for the sake of simplicity, rather than defining a full packet protocol.

Sub-Modules

VPU Modules (adam)

Video Decoder

Description:

This module accepts the raw ntsc video data, extracts yCrCb data from it and then converts it to HSV. Uses the ntsc_decode and rgb2hsb modules.

Data Inputs:

Digital NTSC video input

Data Outputs:

i	Hue	[7:0]
ii	Saturation	[7:0]
iii	Brightness	[7:0]
iv	pixel_x	
v	pixel_y	
vi	hsync	
vii	vsync	

Memory:

Video data does not need to be stored, merely converted to HSV and passed to the CSA and SD modules.

Testing:

This module will be tested by also implementing a module that stores an incoming frame to a file as both yCrCb and HSV data, and then the two can be compared with a simple script to ensure that the module is processing the video correctly.

Chromatic Spectrographic Analysis (CSA)

Description:

This module accepts the HSV outputs from the Video Decoder and processes them to extract information about the most occurring colors in each frame and the average saturation and brightness of that hue. The peak detection algorithm for each subdivision of the frame is described below:

```
Peak detection:
    Iterate over every pixel in the sub-section
        add 1 to address(hue) in list Count  (2 brams)
        add Saturation[7:0] to address(hue) in list Sat (1 bram)
        add Brightness[7:0] to address(hue) in list Bright (1 bram)
    Window accumulate across list Count (find window with highest value)
    Use the center of the max window as the max hue value
    Take the average of the window centered at the max hue for the lists
         Sat and Bright and use these for the Saturation and Brightness
        Peaks
```

Data Inputs:

i	Hue	[7:0]
ii	Saturation	[7:0]
iii	Brightness	[7:0]
iv	pixel_x	
v	pixel_y	
vi	hsync	
vii	vsync	

Data Outputs:

packet bus with 4 word packet, 8 bit word.

1. HSV address	[7:0]
2. Hue_peak	[7:0]
3. Saturation_peak	[7:0]
4. Brightness_peak	[7:0]

Memory:

There are 1024*768 = 786432 pixels to be tracked. [Count] has 256 bins each with 20 bits, for a total of 5120 bits of memory (1 bram). [Saturation] and [Brightness] each store 256 bins with 28 bits (max value of the sum of the pixel values if they are 256 is 201.3million which fits in 28 bits) for a total of 2 x 7168 bits so they use 1 bram total. The windowing function has a max width of 32, so for [count] it must store 2x 25 bits, one for the current window, and one for the current maximum.

Testing:

This module will be tested by giving it predefined frames of video as a file of stored HSV values, and we will compare the output to the expected output (expected output will be calculated by a python script and stored as a file).

Sprite Detection

Description:

This module accepts the HSV outputs from the Video Decoder and processes them to extract information about objects of predefined types. In particular, a Difference of Gaussians method will be used to find blobs, using downsampling to approximate gaussian scaling. This allows us to progressively downsample the section of image buffer used, drastically saving memory. This module then uses frame-to-frame correlation to determine the trajectories and velocities of the objects found. This location and trajectory information is compared to a set of pre-configured rules, and produces a set of outputs specifying values for rules which are active. Rule types will include “object present in area” and “object crossed line in a specific direction.”

Data Inputs:

i	Hue	[7:0]
ii	Saturation	[7:0]
iii	Brightness	[7:0]
iv	pixel_x	
v	pixel_y	
vi	hsync	
vii	vsync	

Data Outputs:

packet bus with 3 word packet, 8 bit word

1	Rule address	[7:0]
2	Rule value	[13:8]
3	Rule value	[7:0]

Memory:

12 BRAMs, SDRAM

Testing:

This module will be tested in a similar manner to the CSA module, with predefined frame inputs and an expected output, however this module will be tested on multiple correlated frames in order to test the calculation of trajectory and velocity of objects.

Control Unit Modules (clark and gabe)

Hardware Input Translation

Description:

Convert user inputs (User IO, buttons, switches) into control parameters. This will be used to generate the control signals for the sequencer module and for the Fx modules. User input will be mapped to the sequencer inputs which control sample saving, step assignment, and playback. For the FX modules, user input will be mapped to the parameters controlling the effects, such as which effect is enable and the wet/dry mix.

Data Inputs:

packet bus with 4 word packet, 8 bit word.

i	HSV address	[7:0]
ii	Hue_peak	[7:0]
iii	Saturation_peak	[7:0]
iv	Brightness_peak	[7:0]

packet bus with 3 word packet, 8 bit word

1	Rule address	[7:0]
2	Rule value	[13:8]
3	Rule value	[7:0]

Data Outputs:

Sequencer:

memory_address [3:0]
store_sample -->asserted when sample to be saved
clear_sample -->asserted when saved sample to be cleared
play_sample → plays a sample from memory without assigning it
tempo -->asserted when the sequencer step is to be incremented
step_address [3:0]
store_step → asserted when a step is to be assigned
clear_step → asserted when a step assignment is to be cleared
mode_select → low for continous mode, high for sequencer mode

FX modules:

effect enable → low when audio is passed through, high when effect is inserted
effect select → which effect is utilized
wet/dry mix → ratio of original audio to audio with effects
effects parameters → to be determined

Memory:

No memory intended to be used, this module is a input to control mapping

Testing:

This module will be tested by loading the module onto the fpga with all of the user controls connected, and then outputting their corresponding control signals to the logic analyzer to make sure the are mapping correctly.

<u>Video Input Translation</u>

Description:

This module accepts the outputs from both the video processing submodules and converts the data into samples and triggers for the sequencer. The output from the CSA sub-module will be mapped from HSV into a frequency domain signal, where the Hue is used to define the fundamental frequency for a finite list of harmonic oscillators. The brightness and saturation values are then used to control the width and center frequency of a windowing function that filters the oscillators. This creates a sample where color controls frequency and the vibrancy and brightness of the sample controls how colorful and rich the sound is. The output from the Sprite Detection Submodule is then used to generate the tempo input for the sequencer and sample triggers for the sampler.

Data Inputs:

packet bus from CSA with 4 word packet, 8 bit word.

1. HSV address [7:0]
2. Hue_peak [7:0]
3. Saturation_peak [7:0]
4. Brightness_peak [7:0]

packet bus from Sprite Detection with 3 word packet, 8 bit word.

1. Rule address [7:0]
2. Rule value [13:8]
3. Rule value [7:0]

Data Outputs:

packet bus with N word packet, 32 bit word

N -> (# of oscillators) * (number of harmonics)

N_max -> [16 osc] * [128 max] == 2048 words
32 bit word -> { frequency [15:0], value[15:0] }
Sample_address [7:0]

Memory:

There will be a lookup table to convert rule addresses and values into sample triggers, this will be 8 x 14 x 8 = 896 bits. There will also be a lookup table for the coefficients of the windowing function, which will be 1024 x 8 = 8192 bits which will use 1 bram. The lookup table for mapping hue to frequency will be 256 x 16 = 4096 bits and will fit in the same bram as the windowing function coefficients.

Testing:

This module will be tested by using modelsim to input a series of predefined packets that span the range of the lookup tables, to make sure the a given input will produce the expected output. This is easy to verify as modelsim allows the input, output and lookup tables to be inspected.

Sequencer Modules **(clark)**

Sequencer

Description:

The sequencer implements a bank of 16*n (n=1,2,3,4)memory locations where incoming samples and triggers can be stored. It also has a 16*n (n=1,2,3,4) step sequencer that moves from step to step based on the tempo input. Each step of the sequencer can be assigned to a rest, the current input, or any of the stored memory samples. Before sending the frequency samples to the output, it is passed through a basic FX unit that implements simple reverb and filters. This is easy because we have a finite frequency representation of each sample. To implement reverb, the module also stores the last N played samples, and adds them to the current sample with decaying coefficients.

Data Inputs:

packet bus with N word packet, 32 bit word
N -> (# of oscillators) * (number of harmonics)
N_max -> [16 osc] * [128 max] == 2048 words
32 bit word -> { frequency [15:0], value[15:0] }
Sampler_address [7:0]

Control signals
memory_address [3:0]
store_sample -->asserted when sample to be saved
clear_sample -->asserted when saved sample to be cleared
play_sample → plays a sample from memory without assigning it
tempo -->asserted when the sequencer step is to be incremented
step_address [3:0]
store_step → asserted when a step is to be assigned
clear_step → asserted when a step assignment is to be cleared
mode_select → low for continous mode, high for sequencer mode

Data Outputs:

packet bus with N word packet, 16 bit word (for 2N point IFFT)

1	IFFT_data	[15:0]	(real component only, positive coefficients.)

packet bus with 1 word packet, 8 bit word

i	sampler_address	[7:0]

Memory:

Samples: (using zbt ram)

# of bits = (# of oscillators) * (# of harmonics) * (32 bits) * (# of sequencer steps)

max_bits = [16] * [128] * 32 * [64] =~ 4.2mbits

Reverb: (using zbt ram)

# of bits = (# of oscillators) * (# of harmonics) * (32 bits) * (# of samples for reverb)

max_bits = [16] * [128] * 32 * [64] =~ 4.2mbits

In total, this will use 8.2Mbits of 1 zbt ram for the stored samples, as well as a lookup table of size (16 x 18 = 288 bits) to store the assignments of each step of the sequencer.

Testing:

This module will be tested by feeding it pre-defined packets as input, and then storing and clearing each address in the sample memory and then repeatedly changing the assignments of the steps. For a given step assignment and known memory storage, the expected output for each of the 16 steps is known.

Frequency-Domain FX

Description:

The Frequency-Domain FX unit will be responsible for applying parametric effects to the inputs of the synthesizer which are inexpensive to compute in frequency domain, but expensive in time domain. These consist of Reverb and arbitrary filtering/equalization. Filters will be applied by multiplying the frequency output by a desired frequency response. Reverb will be applied by evaluating an FIR on the desired frequency response, using the saved values in the Reverb memory. This acts on streaming data and appears as a pass through to the Sequencer and IFFT.

Data Inputs:

packet bus with N word packet, 16 bit word (for 2N point IFFT)

1	IFFT_data	[15:0]	(real component only, positive coefficients.)

Data Outputs:

packet bus with N word packet, 16 bit word (for 2N point IFFT)

1	IFFT_data	[15:0]	(real component only, positive coefficients.)

Memory:

Accesses the ZBT RAM space used by the sequencer; and uses lookup tables to store the coefficients of various filters.

Testing:

This module will be tested by feeding the input simple IFFT lists and then applying each of the filters to it. The expected output for a given filter and input is easily computed.

Audio Modules (gabe)

Sampler

Description:

The sampler produces sounds by playing back and mixing pre-recorded samples (in this case, drum sounds) stored in ROM. Each Sample Unit will comprise a ROM and a collection of {active flag, counter} pairs. For each audio frame, each SU will increment it's counters, look up the value stored in the ROM at the index pointed to by each active pointer, and accumulate these. It will express it's sample address and aggregated value.

When a sample start signal is received, any SU with matching address will locate a non-active counter (or the oldest active counter if no free counter is available,) reset it, and mark it as active. When a counter reaches the length of the sample, it is freed (marked as inactive.)

Output is in the form of a serialized string of address/value pairs.

Data Inputs:

packet bus with 1 word packets, 8 bit words

1	Sample Address	[7:0]

Data Outputs:

Raw output of N 16-bit words, one for each SU

Memory:

As needed by samples. Expected to be no more than 32 BRAMs (may choose to downsample and reconstruction-filter some samples without substantial high frequency content to save on memory.)

Testing:

Testing will primarily consist of the use of SE-Verilog/Modelsim testbenches. Additionally, since the module accepts asynchronous start signals, and generates addressed 16-bit

Channel Mapper/Aggregator

Description:

Takes the outputs of the synthesizer (IFFT) and sampler, aggregates values, and outputs a unified serial stream of values assigned to channels. In this way the fact that these are two sound sources (IFFT and sampler) is transparent to the following unit, and we can make the address space of effects unit channels independent of the details of the sampler/IFFT.

Data Inputs:

IFFT output [15:0]
Sampler output [15:0][0:N-1] (for N samplers. Not actually an array type, as Verilog does not support arrays as inputs and outputs.)
ready

Data Outputs:

packet bus of 2 word packets, 16 bit words

1. {8'b0, Channel Number [7:0]}
2. Value [15:0]

Memory:

Distributed Ram for an address-> channel lookup table. No dedicated memory resources.

Testing:

The unit will be tested by running pre-generated test vectors in a SE-Verilog/Modelsim testbench.

Audio Synthesis / IFFT

Description:

A CORDIC based Radix 2^2 Single Path Delayed Feedback IFFT. In order to get good frequency response, the design will be an N point IFFT with N expected to be 2^{15}-2^{16}. For a R2^2SDF architecture, this requires 4 complex multipliers, 16 butterfly stages, and N complex words of FIFOs. Additionally, we will require N complex words of input buffer, and some amount of output buffer in which to do sample reordering for natural-order output.

In order to handle the extreme memory requirements this design imposes for FIFOs, we will time multiplex one ZBT RAM to serve all of the FIFOs of more than 32kb (1024 complex words, 2 BRAMs) Since this necessarily limits our throughput (to 2 clock cycles per FIFO using ZBT RAM) and we are using a fully pipelined CORDIC, we will be able to time-share one CORDIC for all of our internal multiplications, and still be at under 50% utilization.

We make use of this by using idle cycles to compute a Hilbert Transformer on the input to the IFFT. In other words, we choose some fixed timing window that is longer than the time required to compute the IFFT and compute the number of audio samples that will be required for that period. We then rotate the inputs to the IFFT in phase space by that amount of time, so that we can switch between the middle of one output window and the beginning of the next without inducing inconsistency in the phase of each harmonic component.

Since our expected system clock is at least 68 MHz, our expected audio output is at most 48kHz, and our throughput should be no worse than 1/24 during operating cycles, we can therefore use a new set of harmonic components every 60th of a second with some room for additional delays. This gives us the frequency precision of a 2^{16} point IFFT (.73Hz/bin) with the timing precision of an 800 point IFFT, effectively escaping the frequency/timing precision tradeoff at the cost of increased processing power requirements.

In order to ensure phase consistency in the presence of potentially variable input timing, the IFFT unit will initially buffer all inputs, and will only calculate the necessary phase lag, correct, and process a new set of parameters when that set of parameters is complete. This will allow the system to be robust to potential packet cancellation (restart) or delays in input from the sequencer. In the event of a substantial delay (e.g., disabling the upstream datasource) this will cause continuous looping at all frequencies that are harmonics of the output buffer length, and phase discontinuities at frequencies which are not, with period equal to the buffer length. For a buffer length of at least 2 BRAMs, this would be below 25 Hz, and primarily subaudible, so that windowing increases our temporal accuracy and decreases memory usage, without sacrificing functionality or robustness.

Data Inputs:

packet bus with N/2 word packet, 16 bit words for N point IFFT

1	IFFT_data	[15:0]	(real part, positive frequencies only.)

Data Outputs:

packet bus with M word packets, 16 bit words (for window size M)

1	IFFT_output	[15:0]

Memory:

4 BRAMs for internal FIFOs, 2 BRAMs for output buffers, 2Mbits of ZBT RAM

Testing:

The module will be tested using SE-Verilog/Modelsim testbenches to run pre-generated test vectors, and strings of commands. Output will be correlated against evaluations of the same using floating-point IFFT in numpy, to verify functionality and quantify quantization noise imparted by the transform.

The module will then be tested on the labkit by using a suitably scale-down parameterization as a drop-replacement for the LogiCORE IFFT module used in the demo spectrum analyzer.

Time-domain FX

Description:

The Time-Domain FX unit will comprise two subsystems, one of which will itself consist of a collection of submodules sharing a common interface.

The first subsystem will implement a channel-based patchboard, routing output channels to input channels. Output channel 0 will be the parametric synthesizer (IFFT,) and channels 1-15 will be sampler outputs. The remainder of the output channels will be outputs of modular effects units. Input channels will comprise the inputs to effect units, as well as the outputs to the Audio Encoder. In this way, the effects units need only to understand that they take in some number of audio streams, and output an audio stream --- the process of signal routing and the order in which modules are applied to any given stream can be entirely external to their operation.

Values will be buffered, and the effect of each unit will be effectively “non-blocking.” Application of each module will iterate every time ready is asserted, so that the commulative latency of the unit will be N*(1/48kHz) for an N-module processing path. The worst case possible latency for this unit is therefore 5ms for a continuous chain of 246 one-input/one-output effects units. In practice, latency will generally be much lower, as a typical effects chain will be much shorter than that worst case.

The second subsystem consists of a parallel collection of effects units, with inputs tied to the input patchboard, and outputs tied to the output patchboard. These units will include

1. FIRs (high pass, low pass, and band pass. possibly including combination filters such as equalizers.)
2. Gain control.
3. Mixdown (sum signals with parameterizable weights.
4. Echo
5. (possibly other effects if time permits.)

TD-FX unit will also provide the output channels, selectably, as control channels, allowing --- for example --- the creation of a gain unit which uses a function of the RMS value of it’s control parameter as it’s gain. Feeding the input to that unit as the control parameter would produce a compressor or leveler, depending on other parameters. Feeding a different channel as the parameter would produce a side-chain compressor. Feeding a constant value would produce a standard gain adjusting unit.

Data Inputs:

packet bus of 2 word packets, 16 bit words

1	{8'b0, Channel Number [7:0]}
2	Value [15:0]

packet bus of 4 word packets, 8 bit words

1	Channel Number	[7:0]
2	Parameter Number	[7:0]
3	Value	[15:8]
4	Value	[7:0]

ready

Data Outputs:

TD-FX_left	[15:0]
TD-FX_right	[15:0]
TD-FX_re	

Memory:

1 BRAM to store patch-bay routings and input values. Additional memory for effects units, compile-time configurable. (substantial for memory-based effects like echo, less for short-buffering effects like FIRs and IIRs, much less for re-pan, re-gain, and other memoryless units.)

Testing:

The patchboard will be tested using a bench which imposes easily identifiable values onto the inputs (i.e., value[i]=i) and then produces a sequence of ready and control signals. The known values being routed will allow straight-forward verification that routing is happening correctly.

The effects units will be tested individually in their own testbenches --- in simulation and on the lab-kit. Since each accepts 16 bit audio-frequency data samples, and generates 16-bit audio-frequency samples, an effects unit can be implemented on the labkit tied to the Audio Encoder or some similar AC'97 wrapper, and directly evaluated without needing any of the rest of the system.

<u>Audio Encoder</u>

Description:

The audio encoder will take the processed sample stream from the Time-Domain FX unit's mixdown, and convert it into a stereo pair of analog outputs. This will consist of a driver for the AC'97, processing stereo 16 bit audio at 48kHz.

Data Inputs:

TD-FX_left	[15:0]
TD-FX_right	[15:0]
TD-FX_re	
Volume [4:0]	

Data Outputs:

ready

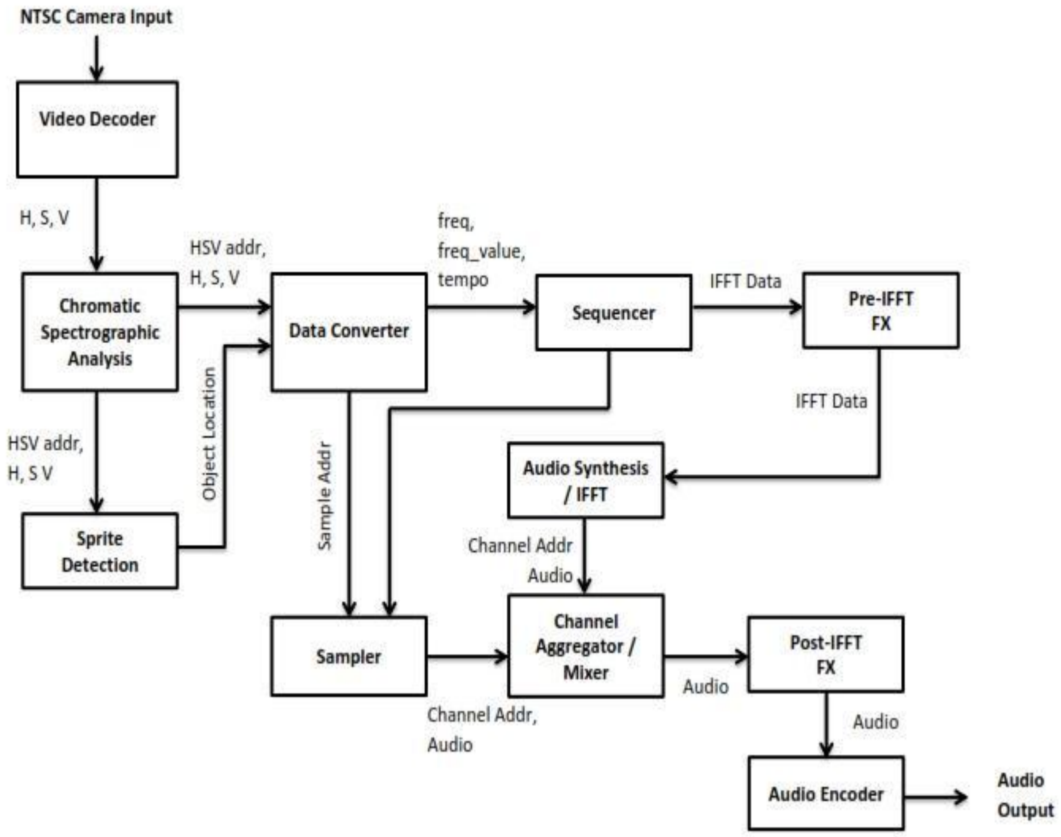
Memory:

SDRAM as needed, no dedicated memory resources

Testing:

A testbench will be created to feed pre-generated waveforms to the module. Output will be verified on the labkit with the oscilloscope.

Block Diagram


NTSC Camera Input
Video Decoder
H, S, V
Chromatic Spectrographic Analysis
HSV addr, H, S, V
Data Converter
Object Location
HSV addr, H, S V
Sprite Detection
freq, freq_value, tempo
Sequencer
IFFT Data
Pre-IFFT FX
IFFT Data
Sample Addr
Audio Synthesis / IFFT
Channel Addr Audio
Sampler
Channel Aggregator / Mixer
Channel Addr, Audio
Audio
Post-IFFT FX
Audio
Audio Encoder
Audio Output