Auralization of the Visual World

Clark Della Silva, Gabriel Karpman, Adam Suhl

December 12th, 2012

Abstract

The goal of this project was to produce a system that produces some form of audio representation for the visual world. The system takes a video input, process it, and uses the data to control a music synthesis system. The visual field is processed, and a section the visual field is parsed into control parameters which are fed to the a sequencer. The sequencer allows selection and recording of tones, and the arrangement of previously-discovered tones into melodic patterns, and then feeds these tones to the audio output unit. The audio output uses the output of the sequencer to control a highly parametric synthesizer (IFFT), and uses separate user controls to control a drum machine. The outputs of the synthesizer, drum machine, and audio processing modules are routed through a shared, addressed bus, and the final outputs are passed out to speakers through the AC'97 as stereo 16 bit 48000 Hz PCM audio.

Overview

Our project aimed to explore the interplay between audible and visual information, by using video capture as a controlling medium for sound creation. There is a large discrepancy between the amount of information stored in video compared to audio. This means that for the reverse problem of representing auditory information as visual information it is much easier to produce a representation that contains intuitively identifiable results. However, the production of an audio representation of the visual field in a way that is identifiable is a much harder problem. In addition, we wanted to enhance the project in an artistic manner by allowing the control and manipulation of the auditory representation in order to create music. In particular, we process information from an NTSC XVGA camera, as well as hardware user controls, to produce a dynamic soundscape. This production consists of two major types of interaction.

1. Ambient analysis of a visual field: we analyze a region of the image, and extract average hue, saturation and brightness --- with temporal smoothing to account for several-bit noise in the feed of the NTSC video camera available.

2. User controls in the form of on-labkit buttons and switches, and a PS/2 keyboard.

To allow for the user to record sequences of either the ambient or intentioned streams, loop them, and rearrange them into different patterns, the interface between the control set extracted from the video feed and the sound-generating components is a step sequencer and looper. The soundstream produced is be passed through a final general, packet-controlled routing layer, through a collection of mixers, and to an AC'97-based output stage for playback. The routing layer, as implemented on the labkit, provides small modules for extracting self-addressed control and sound from the shared busses, and is fully configured to route modules on up to 122 additional channels, although due to time constraints the only units instantiated are a pair of multi-channel mixers and the AC'97 drivers for left and right channel.

To alleviate the otherwise substantial complexity of the project, and to allow for an extreme degree of design flexibility and isolated testing of modules, the flow of information into and out of each subunit was abstracted to a substantial degree. This was be done by implementing the project such that a 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. This allowed different components to operate correctly in separate builds, even during last minute panicking, editing, and code failure.

Primary Modules

The system consists of four major components, which are fully modular and able to function, given suitable input and output, wholly independent of the others. Further sub-divisions of these modules will be explained below (see page for diagram). These systems are :

1. *Video processing unit (VPU).* This unit takes an NTSC video stream, and extracts a set of Hue/Saturation/Brightness values representing a temporally-smoothed version of a fixed region of the screen.

2. *Control unit (CU).* This unit takes in the outputs of the VPU, as well as hardware user inputs, and generates 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. This proved useful near the end of the project, as we were able to add last-minute support for PS/2 keyboard controls in the drum machine in a matter of minutes, and were only prevented from adding PS/2 keyboard controls for the sequencer unit by an unrelated build error preventing compilation in the half hour before the project demo.

3. *Sequencer*. This unit records patterns containing parameters for the synthesizer. It allows 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 take consists of a frequency response (i.e., the Fourier Transform of the desired signal.) These are stored as Frequency/Value pairs, to enable extremely space efficient storage of sparse data; they are serialized into a train of spaced out pulses as necessary to feed to the synthesizer (IFFT.).

4. *Audio generation and effects*. This unit produces 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 are then mixed together to form channels, which are passed to the effects unit. This unit uses 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 are passed out of the system by routing to the AC'97 left and right channel data inputs, which are instantiated as an (internal) output-less effect unit in the routing matrix.

To promote modularity and ease of routing between these major components, data is passed on word-serial busses. Each bus contains 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) Goes high for one cycle at the beginning of packets. This is synchronous with the device ID for the shared control bus, and 2 cycles before data begins being written to the bus for the inter-module communication busses.
- 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 is generally be several orders of magnitude lower than that available on these buses, we provide a distinct bus for each channel of communication for the sake of simplicity, rather than defining a full packet protocol.

The system block diagram is located at the end of the report (Page 22)

Sub-Modules (Clark and Adam)

1) VPU Modules (Clark and Adam)

1.1) <u>Video Decoder</u> (Clark)

Description:

This module accepts the raw ntsc video data, extracts yCrCb data from it, converts this into RGB and then converts it to HSV.

Implementation:

This module used the provided adv7185, ntsc_decode, YCrCb2RGB, and rgb2hsv modules to get valid hsv data from the video camera. Output from the ntsc_decode module is first synchronized with the system clock using shift registers, and the data is passed through YCrCb2RGB and rgb2hsv. There is a 27 clock cycle delay for HSV data to become valid when passing through these two modules, so frame, h_sync, and v_sync signals are each passed through a 27 bit shift register to ensure they are timed correctly with the pixel data. A pixel and horizontal line count are then created using this data to allow for pixel data to have screen location data associated with it. PCount is incremented on each clock cycle, and is reset when H_sync is asserted. HCount is increment when H_sync is asserted, and is reset when V_sync is asserted. This pixel data (HSV and PCount, HCount) is then passed into the Chromatic Spectrographic Analysis module every clock cycle.

1.2) <u>Chromatic Spectrographic Analysis (CSA)</u> (Adam)

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. More specifically, it extracts the time-averaged hue, saturation, and value of a region of pixels. This is somewhat different from what we originally proposed: we had planned to find the peak (most frequent) hue value and the average saturation and value of pixels of that hue.

The spatial and time averaging were necessary because of the large amount of noise in the data coming from the camera.

In order to conserve resources, the analysis is done without ever storing the image -- the analyzer operates on a stream of pixels.

Data Inputs:

1. Hue [7:0]
2. Saturation [7:0]
3. Value [7:0]
4. Hcount [9:0]
5. Pcount [10:0]

Data Outputs:

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]

2) Control Unit Modules (Clark and Gabe)

2.1) <u>Hardware Input Translation</u> (Clark and Gabe)

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.

v. HSV address [7:0]
vi. Hue_peak [7:0]
vii. Saturation_peak [7:0]
viii. 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

Implementation:

This module was originally

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.

2.2) <u>Video Input Translation</u> (Clark)

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]

Data Outputs:

packet bus with N word packet, 24 bit word

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

N_max -> [16 osc] * [128 max] == 2048 words

24 bit word -> { frequency [15:0], value[7:0] }

Memory:

There will be a lookup table for the coefficients of the windowing function, which will be 2048 x 8 = 16384 bits which will use 1 bram. The coefficients of the gaussian will be scaled from 1-256 to allow for directly loading the coefficients into the frequency value rather than having to do any calculations.

Implementation:

This module was created using a primary state machine to control data flow and three minor state machines to handle input, output, and the calculations. The primary state machine sends control signals to the minor state machines, and is designed to lock the states of the machines so that they cannot run at the same time. This prevents data from being changed while it is being used. The primary state machine sends a start signal to a given state machine, and then locks its state until the busy signal from that state machine goes high, indicating it has started, and then has gone low again, indicating it has run to

completion.

The input and output state machines are standard across all of the modules. It waits until it is ready to transceive data, then sets a ready signal ihigh. The state machine then waits until it sees the ready signal from the module it is communicating with. Then both modules begin the transmission one clock cycle after both ready signals have gone high. The data is sent one word per clock cycle with a predefined number of packets to be transferred. After completing a transmission, the state machine returns to its idle state and sets its busy signal low.

After the input state machine finishes receiving new data, the main state machine starts the calculation state machine. This state machine does the calculations for mapping hsv to frequency and indexing into the gaussian coefficients. This is a pipelined operation to allow for multi-step calculations. Hue is mapped to the fundamental frequency with the equation: (Frequency = 256 + Hue). This means that over the complete range of hue (0-255) the fundamental frequency changes one complete octave. there are then 9 harmonics generated above this frequency, increasing linearly by the fundamental frequency. Saturation is used to determine the window across the harmonic series where the coefficients will be non zero, and brightness is used to determine a scaling factor for adjusting the step size, which is used to index into the gaussian window for each harmonic. Saturation from 0-255 mapped to 1-10 harmonics in the window, and brightness from 0-255 mapped from the default step size of 204 samples to ~700.

After these have been calculated, the state machine steps through each harmonic in the series. For each harmonic within the calculated window, it then indexes into the Rom storing the gaussian window and loads the value stored there into the coefficient space for that harmonic. The harmonic series is

stored as in 24 bit words, with the first 16 bits holding the frequency, and the final 8 bits holding the frequency value.

Upon completion of the calculation state machine, the output state machine is then started and passes the frequency information on to the sequencer in the manner described above with 10 packets of 24 bit words.

Testing:

This module was tested by using modelsim to input a series of predefined packets that span the range of HSV space to make sure the a given input will produce the expected output. Verificat of the output was easy as modelsim allows the input, output and lookup tables to be inspected, and the expected values are easily calculated.

3) Sequencer Modules **(Clark)**

3.1) <u>Sequencer</u> (Clark)

Description:

This module implements a pattern sequencer, which allows for audio samples to be stored and then played back and looped in user defined sequences. It implements a method for for up to 16 samples to be stored, and a 16 step sequencer. The module has three modes of operation. Live mode, which passes the current input directly to the output, allowing for real time audio feedback to the video input. Piano mode, which allows samples from memory to be played back, as if each sample was stored to a different key on a keyboard. And Sequencer mode, which uses the 16 step sequencer to allow for

programmed loops of samples. It moves from step to step based on the tempo input and each step of the sequencer can be assigned to a rest, the current input, or any of the stored memory samples.

Data Inputs:

packet bus with N word packet, 24 bit word

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

N_max -> [16 osc] * [128 max] == 2048 words

32 bit word -> { frequency [15:0], value[7:0] }

Control signals

sample_address [3:0] → the sample location in memory

save_sample → asserted when sample is to be saved

play → asserted when a sample is to be played from memory

tempo_clk → asserted when the sequencer step is to be incremented

step_address [3:0] → the step to be accessed

set_step → asserted when a step is to be assigned

mode → defines the mode of operation

Data Outputs:

packet bus with N word packet, 24 bit word

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

N_max -> [16 osc] * [128 max] == 2048 words

24 bit word -> { frequency [15:0], value[7:0] }

Memory:

Samples:

# of bits = (# of harmonics) * (24 bits) * (# of sequencer steps)

# of bits = 10 * 24 * 18 = 4320 bits

This is implemented using a Bram on the Virtex II, with a logsize of 8 and a width of 24 bits.

Implementation:

This module is also implemented using a main state machine and three minor state machines. The main state machine controls the input, output, and store sample minor state machines, as well as handling the logic for controlling each of the 3 modes of operation. The module handles inputs by storing the incoming signal to the 17th address in the Ram, allowing for no change of operation if the output is to be a stored sample from address 1-16 or the current sample. If save_sample is expressed, the current_sample is also written to the memory address indicated by the input sample_address[3:0]. By writing the sample to both locations, it allows for the sequencer and live modes to also output the live sample without having to keep track of where it was stored.

The input and output modules work as described in the Video Input Translation module, waiting for both ready signals to be set high and then assigning data to the bus one word per clock cycle.

The store module is triggered when the save_sample input is set high. When triggered, it writes the current sample into memory, addressed by the input sample_address[3:0]. This allows for samples to be stored and retrieved at will.

The Live mode of operation, the module merely accepts an input, and as soon as the input state machine has finished, it triggers the output state machine to send the data on. While in the Idle state, samples can be stored into memory.

In Piano mode, the module continues to accept input, but does not output anything unless a sample is triggered. When play is asserted, is triggers the output module, using the value from the input sample_address[3:0] to load the output data instead of defaulting to the live address.

In Sequencer mode, the module accepts input, but only outputs when the signal tempo_clk goes high. Each time tempo_clk goes high, the sequencer outputs the data using the value stored in the step array at the current step count to load the output data, and increments the step count by one. In this mode, when set_step is asserted, it loads the value from the input sample_address[3:0] into the array step_state indexed by the value of the input step_address[3:0]. Step_state is a 16x5bit array to allow an address for each step to be stored.

When output from any mode is triggered, the address for the desired sample is set, and then the output state machine is triggered.   When it is triggered, it uses the address as the first index into the memory, and then increments by one, up to the number of harmonics, to send the complete harmonic series.

Testing:

This module was 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. It was then connected to the Video Translation Module (2.2) and run through testing again to ensure proper communication between modules.

4) Audio Modules (Gabe)

4.1) <u>Sampler</u> (Gabe)

Description:

The sampler produces sounds by playing back and mixing pre-recorded samples (in this case, drum sounds) stored in ROM. Each Sample Unit consists

of a ROM and a collection of {active flag, counter} pairs. For each audio frame, each SU increments it's counters, look up the value stored in the ROM at the index pointed to by each active pointer, and accumulates these. It expresses its sample address and aggregated value.

When a sample start signal is received, any SU with matching address locates a non-active counter, if available, resest it, and marks 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 value and read enable, routed to the inter-module audio routing bus. The units are implemented using muxes and hardware shift registers, without relying on state machine paradigms and memory management for internal increment/accumulate. this allows for an exceptionally pipelined design with a sub 5ns maximum worst-case combinatorial path, and demonstrated operation up to 195MHz system clock. This clock has been verified for all core modules that I produced (IFFT, routing and sampler) in their fastest parameterized configurations. The versions instantiated in the labkit used for demo do not, in all cases, meet this specification, as many have had pipeline elements removed to simplify routing given the much slower clock we actually encountered.

4.2) <u>Channel Mapper/Aggregator</u> (Gabe)

Description:

This collection of modules implements a shared-bus control network and shared-bus audio routing layer. It comprises the following elements:

ser_par:

This module consists of a stage of clocked, word-wide parallel-in serial-out shift register. It is instantiated on the labkit as a 128 position serializer, with

individual output modules attached to different bus locations (for the labkit as used for the demonstrations, IFFT is on **BUS[0]**, drums are on **BUS[1**] through **BUS[4]**, and the mixer units for left and right channels are on **BUS[16]** and **BUS[17**] to allow for expansion to the full intended 15 piece drumkit.)

input_router

This module sits on the output of the **ser_par**, and stores the value expressed on the bus when the bus read enable is high, and the bus is currently driving the address bound to the particular **input_router**. The bind address for any particular router can be change dynamically at runtime using the shared control bus, by sending a packet of {device_id, control_addr, new_binding}. This allows for dynamic re-routing of sound arbitrarily, by instantiating effects units which are connected to **BUS** location and which have **input_router** units. On the labkit as used for demo, the feed-through path is only two units long (input -> mixer -> output) but additional units could be added at compile and then swapped in/out or rerouted dynamically at runtime without modifying any other code other than the control units choice of packets to send. (in the debug code with manual packet sending, no additional code changes would be needed to enable additional effects units at all.)

mixdown

This module is an effects unit which accepts a parametrically specified range of inputs, and accumulates them if, in the current read cycle, they are valid and readable. This would have been expanded to a complete mixer/light effects unit if time permitted.

packet_collector

This module sits on the shared control bus, is given a device id and address, and stores packets addressed to itself into a ram. It expresses a packet available pin to enable simple connection of new modules to the shared routing

bus.

4.3) Audio Synthesis / IFFT (Gabe)

Description:

This module takes the frequency domain representation of the audio signal we want to produce and converts it into the time domain. It accepts a sparse list of frequency components and their amplitudes, and produces a time varying signal that contains all of these elements. This particular implementation avoids the standard compromise between high frequency resolution and high temporal resolution by using a high point count IFFT to produce very accurate frequency response, and swapping between two buffers to update the output at a rate many times (in this case, approximately 4000 times) the rate at which the audio output scans through the output buffer. This produces a 2^{14} point IFFT, giving 3 Hz frequency precision and 0.61ms temporal precision. The IFFT itself consists of a tree of submodules as follows:

R22x16kFFT

This module ties together control signals and 7 layers of radix 2^2 IFFT, to provide a serial-in, serial-out 2^{14} point IFFT. Inputs are 9-9 (9 bits real, 9 bits imaginary) and are expanded between layers, rather than truncating to maintain correct fixed point, to produce 16-16 imaginary outputs, with the implicit fractional point moved up 7 bits. We do this to save on inter-module bandwidth and memory requirements in the early stages of the IFFT, in which large feedback buffers would otherwise demand a very substantial fraction of the on-chip resources. This module is also responsible for generating all control signals used by the **R4FFT_section** subunits, and synchronizing reset and output read_enables/ clock enables and other related tasks

R4FFT_section

This module assembles two butterflies (**BFI** which provides a standard R2SDF FFT butterfly, and **BFII** which provides a standard R2SDF FFT butterfly, augmented with a trivial twiddle factor multiplier (TFM) which multiplies by 1 or *-j*, depending on where the butterfly is in the input cycle. This module also contains a **TS_CORDIC** instance which provides a selectable phase shift, when in the highest layer, and which serves as a high precision multiplierless TFM in the remaining modules. In practice, a similar effect can be produce by storing the output samples off shifted by one when converting from bit-reversed output order to natural order, but the inclusion in this unit means that driving the **R22x16kFFT** unit with the output of a larger pre-FFT would immediately generate a larger FFT implementation without needing any internal or external modification to this module --- which could not be accomplished in a straightforward fashion by modifying the bit reversal system. This also lets the system provide appropriate pre-delay, even when the output is to be used immediately in bit-reversed order. The unit **MUL3** is a highly optimized implementation of a 2-bit by n-bit multiplier, which is used for twiddle factor generation. It makes use of Virtex 2 logic slice primitives to implement the 2-bit by n-bit multiplier in a single lookup table per bit. Feedback is stored in the **smart_fifo** module.

BFI

This module implements a standard Radix 2 Single Delay Feedback butterfly. This alternates between cycling data into/out of the delay path fifos, and butterflying data with data coming out of the delay path, which enables proper data re-arranging with only one memory read/write and one simple pass-forward. The unit is implemented using the **AS_CE** modules, which provide hard coded selectable addition/subtraction with bypass, where bypass can select the first or second element. As such, the entire **BFI** is implemented using 1 slice per

word bit, which is as compact as it is possible to produce such a structure in the Virtex 2 architecture. Due to the extremely regular nature of the design, this is implemented on chip as a single local routing network, providing relatively fast routing and good propagation characteristics.

BFII

The **BFII** module augments the **BFI** module with a **muxIM** layer, which serves to selectably multiply the input by 1 or ***-j.*** This allows us to merge more general twiddle factor multiplications, which produces the radix 2^2 SDF structure, rather than the radix 2 SDF structure, halving the required number of complex multipliers

TS_CORDIC

This module serve to wrap a **CORDIC** module, providing a time-matched pipeline for additional information. This is used, in the FFT, to pass forward control parameters with samples being multiplied, so that the pipeline latency increase from each **CORDIC** unit is transparent to the modules using it. This allows us to vary the angular precision of the TFMs at compile time, and even have different layers have different angular precisions, without needing to modify any other code.

CORDIC

This module implements a selectably pipelined, parametrically defined vectoring mode **CORDIC** rotation engine. It currently contains coefficients to process words of up to 32 bit width, and up to 34 bit angular precision; SNR appears to be at least 120 dB for 16 bit word, 18 bit angular precision, but this has not been fully verified. The unit is highly optimized and written almost entirely in

Virtex 2 hardware primitives, like all other low level algebraic units in this FFT/IFFT implementation. Each cordic layer is one LUT per bit, and the initial rotation is 1 LUT per bit, for 0, 90, 180 or 270 degree rotations. When both buffering parameters are set, the unit is fully pipelined, with a latency equal to one less than the angular precision, a throughput of one. When only fine buffering is selected, the unit is pipelined only in the fine rotations, and has a latency of angular precision - 2. When no buffering is selected the unit has a latency of 1, but a very substantial combinatorial delay.

When implemented at 8 bit word precision and 10 bit angular precision (with variable vector inputs) the unit achieves 102 slices on the Virtex Xc2V6000-4, bettering the 117 slices achieved for a CORDIC sine/cosine generator by Prashar and Singh, on whose work the design is heavily based. When reconfigured to act as a sine/cosine generator (without variable inputs) the unit can be decreased to under 96 slices, and is optimal for ripple-carry based unrolled CORDIC engines in the Virtex 2 architecture.

mybram_flip

This module provides a dual port (one-read, one-write) read-before-write page flipping buffer. The page flip is implemented using an address toggle, so that the output is fully synchronous with the ram, and avoids glitching when flipping pages, regardless of the state of the reader/writer.

smart_fifo

This unit provides a wrapper for a variety of conditionally generated FIFO/ring buffer elements. Single element delays are implemented using the FDE (D flipflop with clock enable) primitive to provide extremely low latency synchronous output. elements with delays between 2 and 16 are implemented using the

SRL16E (16 element shift register with clock enable) primitive, to provide one-bit-per-LUT storage. In practice, it would have been cost effective to extend these up to 64 or 128 elements. However, in the interests of coding time, I chose to implement all larger FIFO elements using counters and BRAMs.

4.4) <u>Time-domain FX</u> (Gabe)

Description:

The time domain FX unit was intended to contain modular units to be swapped into the audio routing network. In final implementation, the framework was completed successfully, but the only individual FX units to be produced were a multi-channel mixer to combine sampler outputs, and the AC'97 connection.

4.5) <u>Audio Encoder</u> (Gabe)

Description:

The Audio encoder unit provides a wrapper for the AC'97, providing reset correction to enable system clocks up to 200MHz, and providing a wrapper to handle access to the shared audio and control busses.

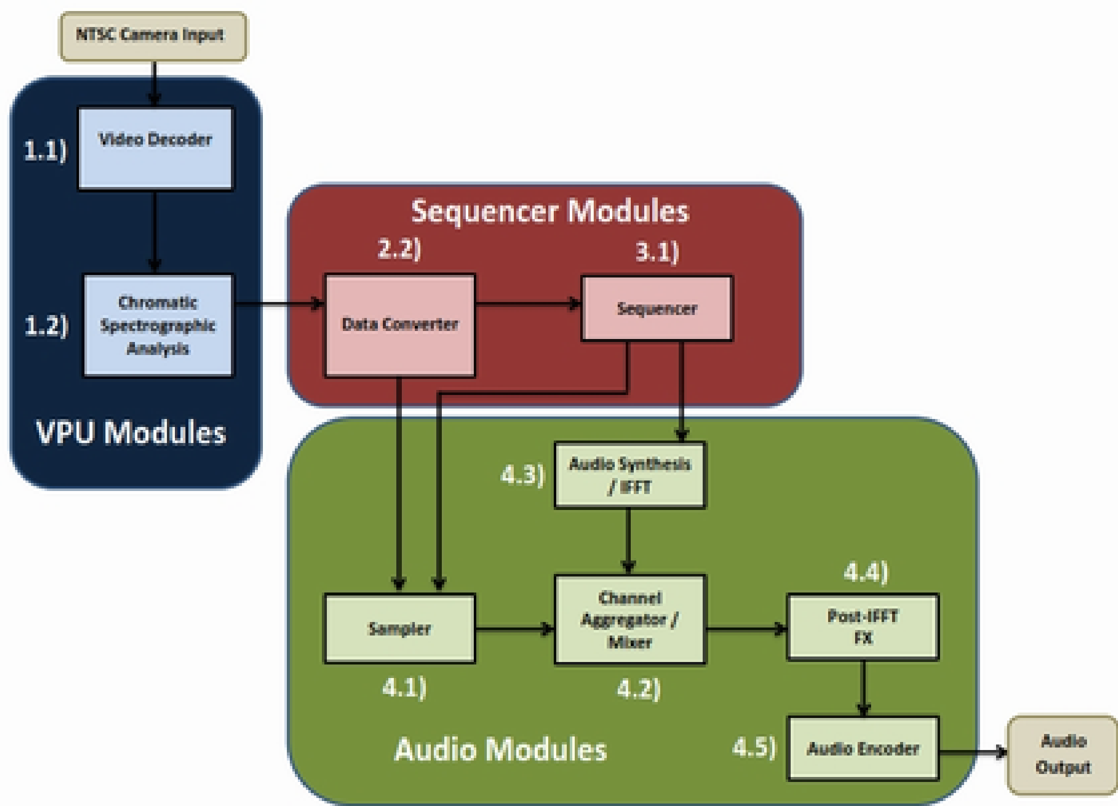
Future Work

Future extensions to this project would include finishing cleaning the PS/2 controls, finishing integrating video control into the project and providing video processing, and implementing additional modular effects. Further extensions could make use of the excessive availability of the FFT (currently generating one full 16k point frame ever 4 audio frames) to multiplex it between operations to allow additional functionality in FX, route-through, and equalization.

Conclusion

This project was certainly ambition, and provided a good deal of challenge. In the end, all core modules for the sequencer and sound generation portions of the design were completed to within the specifications set for them in the proposal. While we would have liked to get more done, the project eventually met or exceeded our minimal expectations in all regards other than video processing.

Block Diagram



CODE

//// LABKIT.v (provided by course staff; modified heavily by all)

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//        "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//        output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
```

```
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//        the data bus, and the byte write enables have been combined into the
//        4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//        hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2006-Mar-08: Corrected default assignments to "vga_out_red", "vga_out_green"
//                  and "vga_out_blue". (Was 10'h0, now 8'h0.)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//                  "disp_data_out", "analyzer[2-3]_clock" and
//                  "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//                  actually populated on the boards. (The boards support up to
//                  256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//                  value. (Previous versions of this file declared this port to
//                  be an input.)
//
```

```
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//                 actually populated on the boards. (The boards support up to
//                 72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////
module labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
        ac97_bit_clock,

        vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
        vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
        vga_out_vsync,

        tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
        tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
        tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

        tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
        tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
        tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
        tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

        ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
        ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

        ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
        ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,
```

clock_feedback_out, clock_feedback_in,

flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
flash_reset_b, flash_sts, flash_byte_b,

rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

mouse_clock, mouse_data, keyboard_clock, keyboard_data,

clock_27mhz, clock1, clock2,

disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
disp_reset_b, disp_data_in,

button0, button1, button2, button3, button_enter, button_right,
button_left, button_down, button_up,

switch,

led,

user1, user2, user3, user4,

daughtercard,

systemace_data, systemace_address, systemace_ce_b,
systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

```
analyzer1_data, analyzer1_clock,
	analyzer2_data, analyzer2_clock,
	analyzer3_data, analyzer3_clock,
	analyzer4_data, analyzer4_clock);

output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input  ac97_bit_clock, ac97_sdata_in;

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output	vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
  vga_out_hsync, vga_out_vsync;

output [9:0] tv_out_ycrcb;
output	tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
  tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
  tv_out_subcar_reset;

input [19:0] tv_in_ycrcb;
input	tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
  tv_in_hff, tv_in_aff;
output	tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
  tv_in_reset_b, tv_in_clock;
inout	tv_in_i2c_data;

inout [35:0] ram0_data;
output [18:0] ram0_address;
```

output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b,
ram0_we_b;

output [3:0] ram0_bwe_b;

inout [35:0] ram1_data;

output [18:0] ram1_address;

output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b,
ram1_we_b;

output [3:0] ram1_bwe_b;

input clock_feedback_in;

output clock_feedback_out;

inout [15:0] flash_data;

output [23:0] flash_address;

output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;

input flash_sts;

output rs232_txd, rs232_rts;

input rs232_rxd, rs232_cts;

input mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input clock_27mhz, clock1, clock2;

output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;

input disp_data_in;

output disp_data_out;

```
input       button0, button1, button2, button3, button_enter, button_right,
  button_left, button_down, button_up;
input [7:0]   switch;
output [7:0]  led;

inout [31:0]  user1, user2, user3, user4;

inout [43:0]  daughtercard;

inout [15:0]  systemace_data;
output [6:0]  systemace_address;
output       systemace_ce_b, systemace_we_b, systemace_oe_b;
input       systemace_irq, systemace_mpbrdy;

output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
  analyzer4_data;
output       analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

////////////////////////////////////////////////////////////////////////////
//
// I/O Assignments
//
////////////////////////////////////////////////////////////////////////////

// Audio Input and Output
assign beep= 1'b0;
//assign audio_reset_b = 1'b0;
```

```
//assign ac97_synch = 1'b0;
//assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// VGA Output
assign vga_out_red = 8'h0;
assign vga_out_green = 8'h0;
assign vga_out_blue = 8'h0;
assign vga_out_sync_b = 1'b1;
assign vga_out_blank_b = 1'b1;
assign vga_out_pixel_clock = 1'b0;
assign vga_out_hsync = 1'b0;
assign vga_out_vsync = 1'b0;

// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

// Video Input
//assign tv_in_i2c_clock = 1'b0;
```

```
assign tv_in_fifo_read = 1'b1;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b1;
//assign tv_in_reset_b = 1'b0;
assign tv_in_clock = clock_27mhz;
//assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
```

```
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
assign flash_we_b = 1'b1;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;
// flash_sts is an input

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs

// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

// LED Displays
/*assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
assign disp_data_out = 1'b0;*/
```

```
// disp_data_in is an input

// Buttons, Switches, and Individual LEDs
assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
assign user1[31:1] = 31'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16'hZ;
assign systemace_address = 7'h0;
assign systemace_ce_b = 1'b1;
assign systemace_we_b = 1'b1;
assign systemace_oe_b = 1'b1;
// systemace_irq and systemace_mpbrdy are inputs

// Logic Analyzer
//assign analyzer1_data = 16'h0;
//assign analyzer1_clock = 1'b1;
//assign analyzer2_data = 16'h0;
```

```
//assign analyzer2_clock = 1'b1;
//assign analyzer3_data = 16'h0;
//assign analyzer3_clock = 1'b1;
//assign analyzer4_data = 16'h0;
//assign analyzer4_clock = 1'b1;

////////////////////////////////////////////////////////////////////////////
//
// Reset Generation
//
// A shift register primitive is used to generate an active-high reset
// signal that remains high for 16 clock cycles after configuration finishes
// and the FPGA's internal clocks begin toggling.
//
////////////////////////////////////////////////////////////////////////////
wire    reseta;
wire    START;
wire reset=reseta;
SRL16E #(.INIT(16'hFFFF)) reset_sr(.D(1'b0), .CLK(clock_27mhz),.CE(1'b1), .Q(reseta),
                          .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
SRL16E #(.INIT(16'hFFFF)) start_sr(.D(reseta),.CLK(clock_27mhz),.CE(1'b1),.Q(START),
                          .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
 wire              ready;

defparam reset_sr.INIT = 16'hFFFF;
localparam ADDR_LEN=7;
localparam DATA_LEN=16;
////////////
```

```
// setup for video stuff
////////////
wire [63:0]   display;
wire [3:0]      disp[0:16];
wire [29:0]   ycrcb;    // video data (luminance, chrominance)
wire [2:0]      fvh;    // sync for field, vertical, horizontal
wire       dv;    // data valid
wire [7:0]      red, green, blue, hue, sat, val;
reg  [9:0]      y, cr , cb;
reg [2:0]      dv_d;
reg [29:0]      y_d, cr_d, cb_d;
reg [2:0]      f_d, v_d, h_d;
reg [9:0]      hcount;
reg [10:0]      pcount;
reg [27:0]      frame_d;
reg [27:0]      hz_d;
reg [27:0]      vert_d;
wire       hi;
wire       chroma_start_write;
wire [7:0]     outhsv;

initial begin
  y = 10'd0;
  cr = 10'd0;
  cb = 10'd0;
  dv_d = 3'd0;
  y_d = 30'd0;
  cr_d = 30'd0;
```

```
cb_d = 30'd0;
f_d = 3'd0;
v_d = 3'd0;
h_d = 3'd0;
hcount = 10'd0;
pcount = 11'd0;
frame_d = 28'd0;
hz_d = 28'd0;
vert_d = 28'd0;
    end // initial begin

    //////////////////
    // Setup for sequencer
    //////////////////

    reg     ready_read = 0;
    wire            ready_write ;
    wire     spectro_start_read, spectro_start_write;
    reg [7:0]     data_in = 8'd0;
    wire [23:0]   spectro_data_out;
    reg [23:0]     data = 0;

    wire     tempo_clk;
    reg     [1:0] tempo_clk_;
    wire     seq_start_read, seq_start_write;
    wire [23:0]   seq_data_in, seq_data_out;
    reg     next_mode, prev_mode, save_sample, set_step, play;
```

```
wire    button_up_, button_down_, button0_, button1_, button2_, addr_ex;
reg     button_up0, button_down0, button00, button10, button20;
reg [4:0]    sample_address;
reg [3:0]    step_address;
reg [1:0]    mode;

reg [15:0]    counter = 0;



/////////////
///  General modules
/////////////
debounce
deb1(.reset(reset), .sys_clock(clock_27mhz), .noisy(~button_up), .clean(button_up_));
debounce
deb2(.reset(reset), .sys_clock(clock_27mhz), .noisy(~button_down), .clean(button_down_));
debounce
deb3(.reset(reset), .sys_clock(clock_27mhz), .noisy(~button0), .clean(button0_));
debounce
deb4(.reset(reset), .sys_clock(clock_27mhz), .noisy(~button1), .clean(button1_));
debounce
deb5(.reset(reset), .sys_clock(clock_27mhz), .noisy(~button2), .clean(button2_));
debounce
deb6(.reset(reset), .sys_clock(clock_27mhz), .noisy(~button3), .clean(addr_ex));

display_16hex display1(.reset(reseta), .clock_27mhz(clock_27mhz),
        .data(display), .disp_blank(disp_blank),
        .disp_clock(disp_clock), .disp_rs(disp_rs),
```

```
                    .disp_ce_b(disp_ce_b), .disp_reset_b(disp_reset_b),
                    .disp_data_out(disp_data_out));
    wire ascii_ready;
    wire[7:0] ascii;
    ps2_ascii_input ps2(clock_27mhz, reset, keyboard_clock, keyboard_data, ascii,
ascii_ready);

        ///////////////////////
        // Video Modules
        ///////////////////////

        // ADV7185 NTSC decoder interface code
        // adv7185 initialization module
        adv7185init adv7185(.reset(reseta), .clock_27mhz(clock_27mhz),
            .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
            .tv_in_i2c_clock(tv_in_i2c_clock),
            .tv_in_i2c_data(tv_in_i2c_data));

        // NTSC video decoder module
        ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reseta),
            .tv_in_ycrcb(tv_in_ycrcb[19:10]),
            .ycrcb(ycrcb), .f(fvh[2]),
            .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

        //YCrCb to RGB conversion module
        YCrCb2RGB
ycrcb2rgb1( .R(red), .G(green), .B(blue), .clk(clock_27mhz), .rst(reseta), .Y(y), .Cr(cr), .Cb(cb));
```

```
		//RGB to HSV conversion module
		rgb2hsv
rgb2hsv1(.clock(clock_27mhz), .reset(reseta), .r(red), .g(green), .b(blue), .h(hue), .s(sat), .v(val));

		//HSV to output for data-converter module
		//chroma #(.CENTER_X(100), .CENTER_Y(400), .LOG_W(3), .LOG_H(3))
chroma_conv(.sys_clock(clock_27mhz), .reset(reseta),
		chroma chroma_conv(.sys_clock(clock_27mhz), .reset(reseta),
			.hue(hue), .sat(sat), .val(val),
			.hi(hi),
			.hcount(hcount), .pcount(pcount), .frame_d(frame_d[27]),
			.start_write(chroma_start_write), .ready_write(spectro_start_read),
			.outhsv(outhsv),
			.sw(switch[2:0]));



		//////////////////
		// Sequencer modules
		//////////////////

		wire [2:0]	in_state;
		wire [7:0]	hueue, sata, vala;

	/*  spectro_conv spectral(.sys_clock(clock_27mhz), .reset(reseta),
			.ready_read(chroma_start_write), .start_read(spectro_start_read),
			.ready_write(seq_start_read), .start_write(spectro_start_write),
			.data_in(outhsv), .data_out(spectro_data_out), .in_stata(in_state),
			.hueue(hueue), .sata(sata), .vala(vala));
```

```
            sequencer
sequencer1( .sys_clock(clock_27mhz), .reset(reseta), .tempo_clk(tempo_clk),
                    .ready_read(spectro_start_write), .start_read(seq_start_read),
                    .ready_write(ready_write), .start_write(seq_start_write),
                    .data_in(spectro_data_out), .data_out(seq_data_out),
                    .sample_address(sample_address), .step_addres(step_addres),
                    .save_sample(save_sample), .set_step(set_step),
                    .play(play), .mode(mode));*/

            assign hi = ((hcount == 10'd150) && (pcount == 11'd400) && (frame_d[27] == 1));
            //assign hi = frame_d[27] == 1;

            ////////////////
            //  Video Logic
            ////////////////

            always @(posedge clock_27mhz) begin
        f_d <=  {f_d[1:0], fvh[2]};
        h_d <=  {h_d[1:0], fvh[0]};
        v_d <=  {v_d[1:0], fvh[1]};
        dv_d <= {dv_d[1:0], dv};
        y_d  <= {y_d[19:0], ycrcb[29:20]};
        cr_d <= {cr_d[19:0], ycrcb[19:10]};
        cb_d <= {cb_d[19:0], ycrcb[9:0]};

        frame_d <= {frame_d[26:0] , f_d[2]}; //use frame_d[27]
        hz_d <= {hz_d[26:0], h_d[2]};  //use hz_d[27]
```

```
vert_d <= {vert_d[26:0], v_d[2]};  //use vert_d[27]
pcount <= (hz_d[27] == 1) ? 0 : pcount + 1;
hcount <= (vert_d[27] == 1) ? 0 : (hz_d[27] == 1) ? hcount + 1 : hcount;

if (dv_d[2]) begin
        y <= y_d[29:20];
        cr <= cr_d[29:20];
        cb <= cb_d[29:20];
end
        end

        //////////////////////
        ////// Sequencer logic
        //////////////////////
        assign tempo_clk=tempo_clk_[0]&~tempo_clk_[1];
        always @(posedge clock_27mhz) begin
if(ready)
         counter <= (counter == 12000) ?  0 : counter + 1;

tempo_clk_ <= {tempo_clk_[0],(counter == 11'd2000) ? 1 : 0};
sample_address <= {addr_ex, switch[7:4]};
step_address <= switch[3:0];

button_up0 <= button_up_;
button_down0 <= button_down_;
button00 <= button0_;
button10 <= button1_;
button20 <= button2_;
```

```
next_mode <= (button_up_ && ~button_up0);
prev_mode <= (button_down_ && ~button_down0);
save_sample <= (button0_ && ~button00);
set_step <= (button1_ && ~button10);
play <= (button2_ && ~button20);

if (counter == 10'd180 || counter == 10'd642) begin
        ready_read <= 1;
end
else if (counter == 10'd185 || counter == 10'd647)  begin
        ready_read <= 0;
end



        end // always @ (posedge clock_27mhz)
        //////////////////////
        ////// Audio
        //////////////////////
        reg [7:0] control_DEV=0;
        reg [7:0] control_ADDR;
        reg [7:0] control_packet[0:4];
        reg [2:0] control_state;
        reg [7:0] control_reg;
reg[7:0]        control;
reg             control_RIE,control_start;
always @(posedge clock_27mhz)
begin
```

```
if(ascii_ready)
begin
case(ascii)
8'h0d:    //begin
          //if (next_mode) begin
          mode <= (mode == 2'd2) ? 0 : mode + 1;
          //end
          //else if (prev_mode) begin
          //mode <= (mode == 2'd0) ? 2 : mode - 1;
          //end
          //end

8'h4F:    begin
          control_DEV<=0;
          control_ADDR<=0;
          control_packet[0]<=0;
          control_state<=3'h6;
          end
8'h50:    begin
          control_DEV<=0;
          control_ADDR<=1;
          control_packet[0]<=0;
          control_state<=3'h6;
          end
8'h5B:    begin
          control_DEV<=0;
          control_ADDR<=2;
          control_packet[0]<=0;
```

```
                control_state<=3'h6;
                end
8'h5D:    begin
                control_DEV<=0;
                control_ADDR<=3;
                control_packet[0]<=0;
                control_state<=3'h6;
                end
endcase
end else
begin
        case(control_state)
        3'h6:begin
                        control<=control_DEV;
                        control_start<=1;
                        control_RIE<=1;
                        control_state<=3'h7;
                        end
        3'h7:begin
                        control<=control_ADDR;
                        control_start<=0;
                        control_state<=0;
                        end
        3'h5:begin
                control_RIE<=0;
                end
        default:begin
                control_state<=control_state+1;
```

```
				control<=control_packet[control_state];
				end
		endcase
	end
	if(ascii_ready)
	begin
		case(mode)
			2'h0:	begin
				save_sample<=1;
				//sample_address<=button_addr
				case(ascii)
					8'h31:		sample_address<=0;
					8'h32:	sample_address<=1;
					8'h33:		sample_address<=2;
					8'h34:		sample_address<=3;
					8'h35:		sample_address<=4;
					8'h36:		sample_address<=5;
					8'h37:		sample_address<=6;
					8'h38:		sample_address<=7;

					8'h51:		sample_address<=8;
					8'h57:		sample_address<=9;
					8'h45:		sample_address<=10;
					8'h52:		sample_address<=11;
					8'h54:		sample_address<=12;
					8'h59:		sample_address<=13;
					8'h55:		sample_address<=14;
					8'h49:		sample_address<=15;
```

```
            default: sample_address<=sample_address;
            endcase
            end
2'h1:   begin
            save_sample<=0;
            play<=1;
            //sample_addres<=button_addr
            case(ascii)
                8'h31:          sample_address<=0;
                8'h32:   sample_address<=1;
                8'h33:          sample_address<=2;
                8'h34:          sample_address<=3;
                8'h35:          sample_address<=4;
                8'h36:          sample_address<=5;
                8'h37:          sample_address<=6;
                8'h38:          sample_address<=7;

                8'h51:          sample_address<=8;
                8'h57:          sample_address<=9;
                8'h45:          sample_address<=10;
                8'h52:          sample_address<=11;
                8'h54:          sample_address<=12;
                8'h59:          sample_address<=13;
                8'h55:          sample_address<=14;
                8'h49:          sample_address<=15;
                default:sample_address<=sample_address;
            endcase
            end
```

```
2'h2:   begin
        //if(button in step): set sample_step, button_step<
        // if button in sample -> save_sample
        case(ascii)
            8'h31:      begin sample_address<=0; end
            8'h32:  begin sample_address<=1;  end
            8'h33:      begin sample_address<=2;  end
            8'h34:      begin sample_address<=3;  end
            8'h35:      begin sample_address<=4;  end
            8'h36:      begin sample_address<=5;  end
            8'h37:      begin sample_address<=6;  end
            8'h38:      begin sample_address<=7;  end

            8'h51:      begin sample_address<=8;end
            8'h57:      begin sample_address<=9;  end
            8'h45:      begin sample_address<=10;  end
            8'h52:      begin sample_address<=11; end
            8'h54:      begin sample_address<=12;end
            8'h59:      begin sample_address<=13; end
            8'h55:      begin sample_address<=14;  end
            8'h49:      begin sample_address<=15; end
            8'h4F:      begin sample_address<=17; end
            default:sample_address<=sample_address;
        endcase
            case(ascii)
            8'h41:      begin step_address<=0;
set_step<=1; end
```

```
8'h53:		begin step_address<=1;
set_step<=1; end
8'h44:		begin step_address<=2;
set_step<=1; end
8'h46:		begin step_address<=3;
set_step<=1; end
8'h47:		begin step_address<=4;
set_step<=1; end
8'h48:		begin step_address<=5;
set_step<=1; end
8'h4A:		begin step_address<=6;
set_step<=1; end
8'h4B:		begin step_address<=7;
set_step<=1; end

8'h5A:		begin step_address<=8;
set_step<=1; end
8'h58:		begin step_address<=9;
set_step<=1; end
8'h43:		begin step_address<=10;
set_step<=1; end
8'h56:		begin step_address<=11;
set_step<=1; end
8'h42:		begin step_address<=12;
set_step<=1; end
8'h4E:		begin step_address<=13;
set_step<=1; end
```

```
                                        8'h4D:          begin step_address<=14;
set_step<=1; end
                                        8'h2C:          begin step_address<=15;
set_step<=1; end
                                        default:sample_address<=sample_address;
                                    endcase
                                    end
                        endcase
                    end else
                    begin
                        play<=0;
                        save_sample<=0;
                        set_step<=0;
                    end
                end
                    always @(posedge clock_27mhz) begin
                if (seq_start_write) begin
                    data <= seq_data_out;
                end
                    end

                wire [DATA_LEN-1:0]    data_s;
                wire [ADDR_LEN-1:0]    addr_s;
                wire                   RE_s;
                reg [ADDR_LEN-1:0]     addr_rec;
                reg                    WE_rec=1;
                wire[15:0]             debug;
                wire[DATA_LEN-1:0]     BUS [0:(1<<ADDR_LEN)-1];
```

```
wire                                VALID[0:(1<<ADDR_LEN)-1];
wire[DATA_LEN:0]    SER[0:(1<<ADDR_LEN)];
assign data_s   =   SER[0][DATA_LEN-1:0];
assign addr_s   =   addr_rec;
assign RE_s   =   SER[0][DATA_LEN];
assign SER[(1<<ADDR_LEN)]=0;
genvar i;
generate
for(i=0;i<(1<<ADDR_LEN);i=i+1)
begin: bus_ser_par
        ser_par  #(.WIDTH(DATA_LEN+1))b_ser_par(
                .C(clock_27mhz),
                .CE(~reset),
                .LOAD(&addr_rec),
                .SER_IN(SER[i+1]),
                .PAR_IN({VALID[i],BUS[i]}),
                .OUT(SER[i])
                );
end
endgenerate
wire C,CE,R;
assign C=clock_27mhz;
assign CE=~reseta;
assign R=reseta;
mixdown #(.MAX(16),.ADDR_LEN(ADDR_LEN),.DATA_LEN(DATA_LEN)) mixdown_left(
        .C(C),
        .CE(CE),
        .R(R),
```

```
        .ready(ready),
        .data_in(data_s),
        .addr_in(addr_s),
        .RE_in(RE_s),
        .data_out(BUS[16]),
        .RE_out(VALID[16])
        );
    mixdown #(.MAX(16),.ADDR_LEN(ADDR_LEN),.DATA_LEN(DATA_LEN))
mixdown_right(
        .C(C),
        .CE(CE),
        .R(R),
        .ready(ready),
        .data_in(data_s),
        .addr_in(addr_s),
        .RE_in(RE_s),
        .data_out(BUS[17]),
        .RE_out(VALID[17])
        );
    always@(posedge clock_27mhz)
        addr_rec<=addr_rec+1;

    wire[13:0] CIN;
    wire[13:0] COUT_;
    wire[13:0] COUT;
    reg s=0;
    wire[7:0] COEF;
    reg[13:0] output_count=0;
```

```
wire[15:0] RO,IO,RO_,IO_,RI,II;
wire RE;
reg[13:0] count=14'h0;
assign RI=COEF;
assign II=0;
always @(posedge clock_27mhz)
begin
	count<=count+1;
	if(&COUT)
		s<=~s;
	if(ready)
		output_count<=output_count+1;
end
FFT_coef uut (
	.C(clock_27mhz),
	.CE(~reseta),
	.R(reseta),
	.CIN(count),
	.COEF(COEF),
	.WOE(ready_write),
	.RE(seq_start_write),
	.FREQ(seq_data_out[23:10]),
	.DATA(seq_data_out[7:0])
	);

mybram_flip #(.LOGSIZE(14),.WIDTH(32)) frame_buffer
	(.addr_in(COUT_),
	.addr_out(output_count),
```

```
		.s(s),
		.clk(clock_27mhz),
		.din({RO,IO}),
		.dout({RO_,IO_}),
		.we(RE));
	R22x16kFFT FFT_inst(
		.C(clock_27mhz),
		.CE(~reseta),
		.R(reseta),
		.START(&count),
		.RI(RI),
		.II(II),
		.RO(RO),
		.IO(IO),
		.RE(RE),
		.CIN(CIN),
		.COUT(COUT),
		.RCOUT(COUT_)
		);
	assign BUS[0] = {RO_[9:0],6'b0};
	assign VALID[0] = 1'b1;
	fast_ac97
#(.IN0(16),.IN1(17),.DEVICE_ID(8'h1),.ADDR_LEN(ADDR_LEN),.DATA_LEN(DATA_LEN),.LOG_DB(
2),.LOG_RST(18))
	ac97(
		.C(clock_27mhz),.CE(~reseta),.R(reseta),.ready(ready),
		.addr_in(addr_s),.data_in(data_s),.RIE_in(RE_s),
	// control wires
```

```
.control(control),.control_start(control_start),.control_RIE(control_RIE),

// ac97 interface (passthrough)

.audio_reset(audio_reset_b),.sdata_out(ac97_sdata_out),.sdata_in(ac97_sdata_in),.syn
c(ac97_synch),.bit_clock(ac97_bit_clock)
,.debug(debug)
);
sampler_4 sampler_inst(C,CE,R,ready,
    control,control_start,control_RIE,
    BUS[1],VALID[1],BUS[2],VALID[2],BUS[3],VALID[3],BUS[4],VALID[4]);

////////////////////////
//////// Test Stuff
////////////////////////

assign analyzer1_data = {seq_data_out[23:10]};
assign analyzer1_clock = clock_27mhz;
assign analyzer2_data = {seq_data_out[7:0], 8'h0};
assign analyzer2_clock = clock_27mhz;
assign analyzer3_data = {seq_start_write, ready_write, 14'h0};
assign analyzer3_clock = ready;
assign analyzer4_data = {data_s};
assign analyzer4_clock = clock_27mhz;
```

```
assign display = {disp[4'hf],disp[4'he],disp[4'hd],disp[4'hc],
        disp[4'hb],disp[4'ha],disp[4'h9],disp[4'h8],
        disp[4'h7],disp[4'h6],disp[4'h5],disp[4'h4],
        disp[4'h3],disp[4'h2],disp[4'h1], disp[4'h0]};

assign disp[0] = switch[7:4];
assign disp[1] = {3'b000, addr_ex};
assign disp[2] = switch[3:0];
assign disp[3] = mode;
assign disp[4] = hue[3:0];
assign disp[5] = hue[7:4];
assign disp[6] = seq_data_out[11:8];
assign disp[7] = seq_data_out[15:12];
assign disp[8] = seq_data_out[19:16];
assign disp[9] = seq_data_out[23:20];
assign disp[10] = spectro_data_out[11:8];
assign disp[11] = spectro_data_out[15:12];
assign disp[12] = spectro_data_out[19:16];
assign disp[13] = spectro_data_out[23:20];
assign disp[14] = outhsv[3:0];
assign disp[15] = outhsv[7:4];

endmodule // labkit

//// CHROMA.v (Clark. modified by asuhl)
```

```
module chroma(sys_clock , reset, hue, sat, val, hi, start_write, ready_write,  outhsv,
pcount, hcount, frame_d, sw);

        input           sys_clock;
        input           reset;
        input           hi;
        input           ready_write;
        input [7:0]     hue, sat, val;
    input wire [10:0] pcount;
    input wire [9:0] hcount;
    input wire frame_d;
    input wire [2:0] sw;

        output reg      start_write = 0;
        output [7:0]    outhsv;

        localparam S_Idle = 3'd0;
        localparam S_Start_transmit = 3'd1;
        localparam S_hsvaddr = 3'd2;
        localparam S_hue = 3'd3;
        localparam S_sat = 3'd4;
        localparam S_val = 3'd5;

        reg [2:0]       video_state = 3'd0;
        reg [7:0]       hsv_addr  = 8'd0;
        reg [7:0]       hue_out = 8'd0;
        reg [7:0]       sat_out = 8'd0;
        reg [7:0]       val_out = 8'd0;
```

```
	parameter LOGW = 6;
	parameter LOGH = 6;
	parameter ENDX = 399;
	parameter ENDY = 150;
	reg [7+LOGW+LOGH:0] avged_hue = 0;
	reg [7+LOGW+LOGH:0] avged_sat = 0;
	reg [7+LOGW+LOGH:0] avged_val = 0;
		wire[11:0] hue_sum = sw[0] ? (hue_out*15 + (sw[1] ?
avged_hue[7+LOGW+LOGH:LOGW+LOGH] : hue)) : sw[1] ?
avged_hue[7+LOGW+LOGH:LOGW+LOGH] : 16 * hue;
		wire[11:0] sat_sum = sw[0] ? (sat_out*15 + (sw[1] ?
avged_sat[7+LOGW+LOGH:LOGW+LOGH] : sat)) : sw[1] ?
avged_sat[7+LOGW+LOGH:LOGW+LOGH] : 16 * sat;
		wire[11:0] val_sum = sw[0] ? (val_out*15 + (sw[1] ?
avged_val[7+LOGW+LOGH:LOGW+LOGH] : val)) : sw[1] ?
avged_val[7+LOGW+LOGH:LOGW+LOGH]: 16 * val;

	wire resetavg = (hcount == ENDY-2**LOGH) && (pcount == ENDX-2**LOGW - 1) &&
(frame_d == 1);
	wire shouldavg = (hcount >= ENDY-2**LOGH) && (hcount <= ENDY) && (pcount >=
ENDX-2**LOGW) && (pcount <= ENDX) && (frame_d == 1);
	wire myhi = (hcount == ENDY) && (pcount == ENDX+1) && (frame_d == 1);
	wire usedhi = myhi;


	reg[10:0] x_, x__;
```

```
always @(posedge sys_clock) begin
         x_ <= pcount;
         x__ <= x_;
end
assign x_changed = pcount != x__;

        always @(posedge sys_clock) begin
if (reset) begin
        video_state <= S_Idle;
        hue_out <= 8'd0;
        sat_out <= 8'd0;
        val_out <= 8'd0;
        start_write <= 0;
end
else case(video_state)
        S_Idle: begin
         video_state <= usedhi ? S_Start_transmit : S_Idle;
         hsv_addr <= 0;
         hue_out <= usedhi ? hue_sum[11:4] : hue_out;
         sat_out <= usedhi ? sat_sum[11:4] : sat_out;
         val_out <= usedhi ? val_sum[11:4] : val_out;
         start_write <= 0;
         //outhsv <= 8'd0;

         if (resetavg) begin
                 avged_hue <= 0;
                 avged_sat <= 0;
                 avged_val <= 0;
```

```
 end
 if (shouldavg && (x_changed || sw[2])) begin
        avged_hue <= avged_hue + hue;
        avged_sat <= avged_sat + sat;
        avged_val <= avged_val + val;
 end
end

S_Start_transmit: begin
 video_state <= (ready_write == 1) ? S_hsvaddr : S_Start_transmit;
 start_write <= 1;
 //outhsv <= 8'd0;
end

S_hsvaddr: begin
 video_state <= S_hue;
 start_write <= 0;
 //outhsv <= hue_out;
 //outhsv <= 8'd0;
end

S_hue: begin
 video_state <= S_sat;
 //outhsv <= sat_out;
end

S_sat: begin
 video_state <= S_val;
```

```
            //outhsv <= val_out;
            end

            S_val: begin
             video_state <= S_Idle;
             //outhsv <= 0;
            end

            default: video_state <= S_Idle;
        endcase // case (video_state)

            end // always @ (posedge sys_clock)

            assign outhsv = (video_state == S_hsvaddr) ? hue_out : (video_state == S_hue) ?
sat_out : (video_state == S_sat) ? val_out : 0;

    endmodule
    /// spectro conv (clark)
    module spectro_conv ( sys_clock, reset, ready_read, start_read,
ready_write,start_write, data_in, data_out, in_stata, hueue, sata, vala);


            /// Inputs and Outpus
            input                   sys_clock, reset;
            input                   ready_read, ready_write;
            input [7:0]         data_in;
            output reg              start_read = 0;
            output reg              start_write = 0;
```

```
output [23:0]           data_out;// = 23'd0;
output [2:0]     in_stata;
output [7:0]     hueue, sata, vala;



//parameters for instantiation
parameter num_harmonics = 10;
parameter step = 8'd204;
parameter num_hsv_addrs = 1;



///  Wire and regs

reg [15:0]              frequency[(num_hsv_addrs*num_harmonics)-1:0]; //n
16bit frequencies
reg [7:0]               frequency_value[(num_hsv_addrs*num_harmonics)-1:0];
//n 16bit frequency values *corresponding to frequency

reg [num_hsv_addrs-1:0]     hsv_addr;
reg [7:0]               hue[num_hsv_addrs-1:0];
reg [7:0]               saturation[num_hsv_addrs-1:0];
reg [7:0]               brightness[num_hsv_addrs-1:0];
integer                 i;

initial begin
for (i = 0; i<1; i = i+1) begin
hue[i] =8'd0;
saturation[i] = 8'd0;
```

```
		brightness[i] = 8'd0;
	end
		end

		////////////////////////////////////////////
		//  bram storing the gaussian coefficients
		////////////////////////////////////////////

		reg [10:0]				address = 0;   //mem address to read gauss coef
		wire [7:0]				gaussian_coeff;  //the bram output value

		ram2048x8 coeffs(.addra(address),.clka(sys_clock),.douta(gaussian_coeff));

		// end Bram
		////////////////////////////////////////////

		//////////////////////////////////////////////////////////////////
		/////  Main Control FSM
		/////	Has 7 states (Idle, Start_input, inputting , Start_calc, Calculating,
Start_output, outputting)
		/////	Has 3 sub fsms (Input, Calculate, Output)
		/////	Control Signals (start_input, input_busy, start_calc, calc_busy,
start_output, output_busy)
		/////	External Controls (ready_read, ready_write)
		/////////////////////////////////////////////////////////////////

		///States
		localparam S_Idle_main = 3'd0;
```

```
localparam S_Start_input = 3'd1;
localparam S_Inputting = 3'd2;
localparam S_Start_calc = 3'd3;
localparam S_Calculating = 3'd4;
localparam S_Start_output = 3'd5;
localparam S_Outputting = 3'd6;



// Variables
reg [2:0]                main_state = 0;
reg                start_input = 0;
reg                input_busy = 0;
reg                start_calc = 0;
reg                calc_busy = 0;
reg                start_output = 0;
reg                output_busy = 0;

always @(posedge sys_clock) begin
    if (reset) main_state <= S_Idle_main;
    else case (main_state)
        S_Idle_main: begin
            main_state <= ((ready_read) ? S_Start_input : S_Idle_main);   //waiting
for input to be available
            start_input <= start_input;
            start_output <= start_output;
            start_calc <= start_calc;
        end
```

```
                    S_Start_input: begin
                            main_state <= (input_busy ? S_Inputting: S_Start_input);  // wait for
input FSM to start then continue
                            start_input <= 1;
                            start_output <= start_output;
                            start_calc <= start_calc;
                    end

                    S_Inputting:  begin
                            main_state <= (input_busy ? S_Inputting: S_Start_calc);   //wait for input
fsm to finish
                            start_input <= 0; // waiting for input, do nothing
                            start_output <= start_output;
                            start_calc <= start_calc;
                    end

                    S_Start_calc: begin
                            main_state <= (calc_busy ? S_Calculating : S_Start_calc); // wait for calc
fsm to start
                            start_output <= start_output;
                            start_input <= start_input;
                            start_calc <= 1;
                    end

                    S_Calculating:  begin
                            main_state <= (calc_busy ? S_Calculating : S_Start_output); //wait for
calculate fsm to finish
```

```
            start_calc <= 0;  // waiting for calculations, do nothing
            start_output <= start_output;
            start_input <= start_input;
        end

        S_Start_output: begin
            main_state <= (output_busy ? S_Outputting : S_Start_output); //wait for
output fsm to start
            start_output <= 1;
            start_input <= start_input;
            start_calc <= start_calc;
        end

        S_Outputting: begin
            main_state <= (output_busy ? S_Outputting : S_Idle_main); //wait for
output fsm to finish then go to Idle
            start_output <= 0;  // waiting for output, do nothing
            start_input <= start_input;
            start_calc <= start_calc;
        end
        default: begin
            main_state <= S_Idle_main;
            start_input <= start_input;
            start_output <= start_output;
            start_calc <= start_calc;
        end
       endcase // case (main_state)
       end // always @ (main_state)
```

```
// End of Main FSM
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
////   Calculator  (Minor FSM)
////
////
////////////////////////////////////////////////////////////////////

//states
localparam S_Idle_calc = 0;
localparam S_Start_calculating = 1;
localparam S_Scale_factor = 2;
localparam S_Step_size = 3;
localparam S_Calculate = 4;

//Internal Variables
reg [2:0]          calc_state = 3'd0;
reg [4:0]      index  = 5'd0;
reg [8:0]      step_size =  8'd0;   //[9:0]
reg [7:0]      scale_factor = 8'd0;

wire      window;
wire [3:0]      len_index;   //[7:0]
wire [3:0]      start_index;  //[7:0]
wire [3:0]      end_index;  // [7:0]
```

```
	assign window = (((index-1) >= start_index) && ((index-1) <= end_index));
	assign start_index = ((num_harmonics * saturation[0]) >>9);
	assign len_index = (num_harmonics * scale_factor) >> 9;
	assign end_index = (start_index + len_index);

	wire[15:0] step_size_mul;
	assign step_size_mul=$unsigned(step)*$unsigned(scale_factor);

	always @(posedge sys_clock) begin
if (reset) calc_state <= S_Idle_calc;
else case(calc_state)
	 S_Idle_calc:  begin
		calc_state <= ((start_calc) ? S_Start_calculating : S_Idle_calc);
		calc_busy <= 0;
	 end

	 S_Start_calculating:begin
		calc_state <= S_Scale_factor;
		calc_busy <= 1;
	 end

	 S_Scale_factor: begin
		calc_state <= S_Step_size;
		scale_factor <= (8'h80 + (brightness[0] >> 1));
	 end

	 S_Step_size: begin
		calc_state <= S_Calculate;
```

```
            step_size <= step_size_mul[15:7];
    end

    S_Calculate: begin
            calc_state <= (index == num_harmonics-1) ? S_Idle_calc : S_Calculate;

            frequency[index] <= ((index+1) * ((18'd256+hue[0])));

            frequency_value[index] <= (window ? gaussian_coeff[7:0] : 8'd1);

            address <= (index * step_size);
            index <= (index == num_harmonics-1) ? 0 : index + 1;

    end

    default: calc_state <= S_Idle_calc;

   endcase // case (calc_state)
   end // always @ (posedge sys_clock)



   // End Calculator FSM
   ///////////////////////////////////////////////////////////////////////////

   ///////////////////////////////////////////////////////////////////////////
   //////  Input serializer   (minor FSM)
   //////             8bit word with 4 packets (hsv_addr, hue, sat, brightness)
   //////             Has 6 states (Idle, start, HSV_Addr, Hue, Sat, Bri)
```

```
////////////////////////////////////////////////////////////////////

//States
localparam S_Idle_in = 3'd0;
localparam S_Start_receiving = 3'd1;
localparam S_HSV_addr = 3'd2;
localparam S_Hue = 3'd3;
localparam S_Sat = 3'd4;
localparam S_Bri = 3'd5;

//Internal variables
reg [2:0]           in_state = 3'd0;      //keeps track of state for input

always @(posedge sys_clock) begin
if (reset) in_state <= S_Idle_in;
else begin
case(in_state)
 S_Idle_in: begin
    in_state <= (start_input) ? S_Start_receiving : S_Idle_in;
    start_read <= 0;
    input_busy <= 0;
 end

 S_Start_receiving: begin
    in_state <= S_HSV_addr;
    start_read <= 1;
    input_busy <= 1;
 end
```

```
S_HSV_addr:begin
        in_state <= S_Hue;
        start_read <= start_read;
        input_busy <= input_busy;
        hsv_addr <= data_in;
end

S_Hue:    begin
        in_state <= S_Sat;
        start_read <= start_read;
        input_busy <= input_busy;
        hue[hsv_addr[0]] <= data_in;
end

S_Sat: begin
        in_state <= S_Bri;
        start_read <= start_read;
        input_busy <= input_busy;
        saturation[hsv_addr[0]] <= data_in;
end

S_Bri: begin
        in_state <= (hsv_addr == (num_hsv_addrs -1)) ? S_Idle_in : S_HSV_addr;
        start_read <= start_read;
        input_busy <= input_busy;
        brightness[hsv_addr[0]] <= data_in;
end
```

```
default: begin
        in_state <= S_Idle_in;
        start_read <= 0;
        input_busy <= 0;
 end
endcase // case (in_state)
end
end

// End of Input FSM
//////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////
///// Output serializer
/////    32bit word, with N packets, where N = no_osc
/////    has 3 states (Idle, start, Transmitting)
//////////////////////////////////////////////////////////

//States
localparam S_Idle_out = 2'd0;
localparam S_Start_transmit = 2'd1;
localparam S_Transmitting = 2'd2;

//Internal Variables
reg [1:0]                    out_state = 2'd0;      //keeps track of state for
output
```

```
        reg [3:0]                   out_counter = 4'd0;  //which frequency osc is being
output
        wire [15:0]          freq;
        wire [7:0]                  freq_v;

        assign freq = frequency[out_counter][15:0];
        assign freq_v = frequency_value[out_counter][7:0];

        always @(posedge sys_clock) begin
    if (reset) out_state <=S_Idle_out;
    else case(out_state)
        S_Idle_out: begin
            out_state <= (start_output) ? S_Start_transmit : S_Idle_out;
            //data_out <= 0;
            output_busy <= 0;
            start_write <= 0;
        end

        S_Start_transmit: begin
            out_state <= (ready_write) ? S_Transmitting : S_Start_transmit;
            start_write <= 1;
            output_busy <= 1;
        end

        S_Transmitting: begin
            out_state <= (out_counter == (num_harmonics)) ? S_Idle_out :
S_Transmitting;
            out_counter <=  (out_counter == (num_harmonics)) ? 0: out_counter + 1 ;
```

```
            //data_out <= {freq,freq_v};
            start_write <= 0;
            output_busy <= (out_counter == (num_harmonics)) ? 0 : 1;
                    end
      default: out_state <= S_Idle_out;
    endcase
    end // always @ (posedge sys_clock)

    assign data_out = (out_state == S_Transmitting) ? {freq, freq_v} : 0;

    // End of Output FSM
    ///////////////////////////////////////////////////////////////////////////////



    assign in_stata = in_state;
    assign hueue = hue[0];
    assign sata =  saturation[0];
    assign vala =  brightness[0];

endmodule // spectro_conv





/// sequencer.v (clark)
module sequencer( sys_clock, reset, tempo_clk, ready_read, start_read, ready_write,
start_write, data_in, data_out, sample_address, step_address, save_sample, set_step, play,
mode);
```

```
input           sys_clock, reset;
input           ready_read, ready_write;
input [23:0]    data_in;
input           tempo_clk;

output reg              start_read = 0;
output reg              start_write = 0;
output [23:0]   data_out;

input [4:0]     sample_address;
input [3:0]     step_address;
input           save_sample,set_step,play;
input [2:0]     mode;

parameter num_harmonics = 10;



///bram decs
wire [7:0]              address; // clogb2(num_harmonics*18)
reg [7:0]               address_in = 8'd0;
reg [7:0]               address_out = 8'd0;
reg [7:0]               address_st = 8'd0;
wire            we;
reg             we1 = 0;
reg             we2 = 0;
reg [23:0]              sample_in2 = 23'd0;
reg [23:0]              sample_in1 = 23'd0;
wire [23:0]             sample_in;
```

```
wire [23:0]                 sample_out;



////////////////////////////////////////////////////////////////////////////////
////   Main FSM
////     Has 3 sub fsm's  (input, output, store_sample)
////     Control Signals  (start_input, input_busy, start_output, output_busy, start_store,
store_busy)
////////////////////////////////////////////////////////////////////////////////


        localparam S_Live_idle = 4'd0;
        localparam S_Piano_idle = 4'd1;
        localparam S_Seq_idle = 4'd2;
        localparam S_Start_receiving = 4'd3;
        localparam S_Receiving = 4'd4;
        localparam S_Start_transmitting = 4'd5;
        localparam S_Transmitting = 4'd6;
        localparam S_Start_storing = 4'd7;
        localparam S_Storing = 4'd8;

        localparam M_Live = 3'd0;
        localparam M_Piano = 3'd1;
        localparam M_Seq = 3'd2;

        reg [4:0]          step_state [0:15];       //16 x 5 bit array to store state of each step
(0-15 = stored, 16 = live, 17 = rest)
```

```
reg [3:0]      step_count = 4'd0;     //what step the sequencer is on
reg [3:0]      seq_state = 4'd0;      // state for the piano player
reg [1:0]      seq_store = 2'd0;      // stores the idle state to return after input
or store

reg [4:0]      address_index = 5'd0;   // the address index that gets passed to
output module

reg            start_output = 1'd0;
reg            output_busy = 1'd0;
reg            start_input = 1'd0;
reg            input_busy = 1'd0;
reg            start_store = 1'd0;
reg            store_busy = 1'd0;

initial begin
step_state[0] = 5'b10000;
step_state[1] = 5'b10000;
step_state[2] = 5'b10000;
step_state[3] = 5'b10000;
step_state[4] = 5'b10000;
step_state[5] = 5'b10000;
step_state[6] = 5'b10000;
step_state[7] = 5'b10000;
step_state[8] = 5'b10000;
step_state[9] = 5'b10000;
step_state[10] = 5'b10000;
step_state[11] = 5'b10000;
```

```
step_state[12] = 5'b10000;
step_state[13] = 5'b10000;
step_state[14] = 5'b10000;
step_state[15] = 5'b10000;
        end

        always @(posedge sys_clock) begin
if (reset) begin
        step_count <= 0;
end

else case(seq_state)
          S_Live_idle: begin
                    case(mode)
                      M_Live: begin
                                if (ready_read) begin
                                  seq_store <= seq_state;
                                  seq_state <= S_Start_receiving;
                                  address_index <= 16;
                                end

                                else if (save_sample) begin
                                  seq_store <= seq_state;
                                  seq_state <= S_Start_storing;
                                end
                      end // case: M_Live

                      M_Piano: seq_state <= S_Piano_idle;
```

```
            M_Seq: seq_state <= S_Seq_idle;
            default: seq_state <= S_Live_idle;
          endcase // case (mode)
end // case: S_Live_idle

S_Piano_idle: begin
          case(mode)
            M_Piano: begin
                    if (ready_read) begin
                      seq_store <= seq_state;
                      seq_state <= S_Start_receiving;
                    end

                    else if (save_sample) begin
                      seq_store <= seq_state;
                      seq_state <= S_Start_storing;
                    end

                      else if (play) begin
                              seq_store <= seq_state;
                              seq_state <= S_Start_transmitting;
                              address_index <= sample_address;
                      end
            end

            M_Live: seq_state <= S_Live_idle;
            M_Seq: seq_state <= S_Seq_idle;
            default: seq_state <= M_Piano;
```

```
        endcase // case (mode)
    end // case: S_Piano_idle

    S_Seq_idle: begin
        case(mode)
         M_Seq: begin
            if (ready_read) begin
             seq_store <= seq_state;
             seq_state <= S_Start_receiving;
            end

            else if (save_sample) begin
             seq_store <= seq_state;
             seq_state <= S_Start_storing;
            end

              else if (tempo_clk) begin
                seq_store <= seq_state;
                seq_state <= S_Start_transmitting;
                step_count <=  step_count + 1;
                address_index <= step_state[step_count];
              end
         end // case: M_Seq

         M_Live: seq_state <= S_Live_idle;
         M_Piano: seq_state <= S_Piano_idle;
         default: seq_state<= S_Seq_idle;
        endcase // case (mode)
```

```
end // case: S_Seq_idle

S_Start_receiving: begin
    start_input <=1;
    seq_state <= ( input_busy ? S_Receiving : S_Start_receiving );
    start_store <= 0;
    start_output <= 0;
end

S_Receiving: begin
    start_input <= 0;
    case(seq_store)
     S_Live_idle: seq_state <= (input_busy) ? S_Receiving :
S_Start_transmitting;
     S_Piano_idle: seq_state <= (input_busy) ? S_Receiving : S_Piano_idle;
     S_Seq_idle: seq_state <= (input_busy) ? S_Receiving : S_Seq_idle;
    endcase
end

S_Start_transmitting: begin
    seq_state <= ( output_busy ? S_Transmitting : S_Start_transmitting);
    start_output <= 1;
    start_input <= 0;
    start_store <= 0;
end

S_Transmitting: begin
    seq_state <= (output_busy ? S_Transmitting : seq_store);
```

```
            start_output <= 0;
        end

        S_Start_storing: begin
            seq_state <= ( store_busy ? S_Storing : S_Start_storing);
            start_store <= 1;
            start_input <= 0;
            start_output <= 0;
        end

        S_Storing: begin
            seq_state <= ( store_busy ? S_Storing : seq_store);
            start_store <= 0;
        end

        default: seq_state <= S_Live_idle;
      endcase // case (seq_state)
      end // always @ (posedge sys_clock)


      always @(posedge sys_clock) begin
  if (set_step) step_state[step_address] <= sample_address;
      end


////////////////////////////////////////////////////////////////////////////////
////   Sample Storer (minor FSM)
////     Stores sample in bram when save_sample pressed
```

```
////////////////////////////////////////////////////////////////////////////////

        // States
        localparam S_Idle_store = 1'd0;
        localparam S_Storing_s = 1'd1;

        // Variables
        reg store_state = 1'd0;
        reg [3:0] store_count = 4'd0;
        reg [23:0]  new_sample [0:9];   //incoming sample from previous module



        always @(posedge sys_clock) begin
    case (store_state)
        S_Idle_store: begin
         store_state <= (start_store) ? S_Storing_s : S_Idle_store;
         store_count <= 0;
         store_busy <= 0;
         address_st <= 0;
        end
        S_Storing_s: begin
         store_state <= (store_count == num_harmonics) ? S_Idle_store : S_Storing_s;
         address_st <= (sample_address * num_harmonics) + store_count;
         we1 <= (store_count == num_harmonics) ? 0 : 1;
         sample_in1 <= new_sample[store_count];
         store_count <= (store_count == num_harmonics) ? 0 : store_count +1;
         store_busy <= 1;
        end
```

```
        default store_state <= S_Idle_store;
  endcase // case (store_state)
        end // always @ (posedge sys_clock)


// End Storing fsm
////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////
////   Input Serializer (minor FSM)
////      24 bit word with num_harmonics packets
////      Has 3 states (Idle, start, Receiving)
////////////////////////////////////////////////////////////////////////////////
        // States
        localparam S_Idle_in = 2'd0;
        localparam S_Start_receive = 2'd1;
        localparam Si_Receiving = 2'd2;

        // Variables
        reg [1:0]     in_state = 2'd0;     //keeps track of input state
        reg [3:0] in_counter = 0;   //keeps track of which point is being input



        always @(posedge sys_clock) begin
  if (reset) in_state <= S_Idle_in;
```

```
else case(in_state)
        S_Idle_in: begin
                in_state <= (start_input) ? S_Start_receive : S_Idle_in;
                in_counter <= 0;
                input_busy <= 0;
                start_read <= 0;
                address_in <= 0;
                we2 <= 0;
        end

        S_Start_receive: begin
                in_state <= (input_busy == 1) ? Si_Receiving : S_Start_receive;
                input_busy <= 1;
                address_in <= 0;
                start_read <= 1;
                in_counter <= 0;
                we2 <= 0;
        end

        Si_Receiving:  begin
                in_state <= (in_counter == num_harmonics) ? S_Idle_in : Si_Receiving;
                address_in <= 8'd160 + in_counter;
                we2 <= (in_counter == num_harmonics) ? 0 : 1;
                sample_in2 <= data_in;
                new_sample[in_counter] <= data_in;
                start_read <= 0;
                in_counter <= (in_counter == num_harmonics) ? 0 : in_counter + 1;
```

```
  end
  default: in_state <= S_Idle_in;
endcase // case (in_state)
end // always @ (posedge sys_clock)

// end Input fsm
////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////
///// Output serializer
/////    24bit word, with N packets, where N = no_osc
/////    has 3 states (Idle, start, Transmitting)
////////////////////////////////////////////////////////////

    //States
    localparam S_Idle_out = 2'd0;
    localparam S_Start_transmit = 2'd1;
    localparam So_Transmitting = 2'd2;

    //Internal Variables
    reg [1:0]                    out_state = 2'd0;       //keeps track of state for
output
    reg [3:0]  out_counter;  //which frequency osc is being output



    always @(posedge sys_clock) begin
```

```
		if (reset) out_state <=S_Idle_out;
		else case(out_state)
			S_Idle_out: begin
				out_state <= (start_output) ? S_Start_transmit : S_Idle_out;
				out_counter <= 0;
				output_busy <= 0;
				start_write <= 0;
				address_out <= 0;
			end

			S_Start_transmit: begin
				out_state <= (ready_write) ? So_Transmitting : S_Start_transmit;
				output_busy <= 1;
				start_write <= 1;
				address_out <= address_index * num_harmonics;
				if(ready_write)
				  out_counter<=1;
			end

			So_Transmitting: begin
				out_state <= (out_counter == (num_harmonics + 1)) ? S_Idle_out :
So_Transmitting;
				address_out <= (out_counter <= (num_harmonics)-1) ?
(address_index*num_harmonics) + out_counter : 0;
	//			data_out <= sample_out;
				out_counter <=  (out_counter == (num_harmonics + 1)) ? 0: out_counter
+ 1;
				start_write <= 0;
```

```
          end
          default: out_state <= S_Idle_out;
        endcase
        end // always @ (posedge sys_clock)

        assign data_out = (out_state == So_Transmitting) ? sample_out : 0;



    // end Output fsm
    ////////////////////////////////////////////////////////////////////////////////



     ////////////////////////////////////////////////////////////////////////////////
    /////  Bram for Sample storage
    /////    384 * num_harmonics bits needed;  width clogb2(24*num_harmonics) bits,
height 16
    ////////////////////////////////////////////////////////////////////////////////

        assign we = ( store_state == S_Storing_s) ? we1 : (in_state == Si_Receiving) ?
we2 : 0;
        assign address = (in_state == Si_Receiving) ? address_in : (out_state ==
So_Transmitting) ? address_out : (store_state == S_Storing_s) ?  address_st : 0;
        assign sample_in =  (store_state == S_Storing_s) ? sample_in1 : sample_in2;

        mybram #(.LOGSIZE( 8 ), .WIDTH(24))
sample_bram(.addr(address), .clk(sys_clock), .we(we), .din(sample_in), .dout(sample_out));

    // End Bram section
```

```
////////////////////////////////////////////////////////////////////////////////
endmodule
//////////////////////////////////////////////////////////////////////////////
//
// Verilog equivalent to a BRAM, tools will infer the right thing!
// number of locations = 1<<LOGSIZE, width in bits = WIDTH.
// default is a 16K x 1 memory.
//
//////////////////////////////////////////////////////////////////////////////
/*
module mybram #(parameter LOGSIZE=14, WIDTH=1)
        (input wire [LOGSIZE-1:0] addr,
        input wire clk,
        input wire [WIDTH-1:0] din,
        output reg [WIDTH-1:0] dout,
        input wire we);
  // let the tools infer the right number of BRAMs
  (* ram_style = "block" *)
  reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
  always @(posedge clk) begin
        if (we) mem[addr] <= din;
   dout <= mem[addr];
  end

////////////////////////////////////////////////////////////////////////////////
```

```
endmodule*/

//// FFT.v (gabriel)

`timescale 1ns / 1ps
`default_nettype none
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    07:03:24 11/01/2012
// Design Name:
// Module Name:    FFT
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module R4FFT_section#(parameter WHICH=0,parameter WIDTH=12,parameter
CWIDTH=14,parameter logn=8 )(
```

```
    input wire              C,
    input wire              CE,
    input wire              R,
    input wire[CWIDTH-1:0]  PHASE,
    input wire[logn-1:0]    CIN,
    input wire[WIDTH-1:0]   RI,
    input wire[WIDTH-1:0]   II,
    output wire[logn-1:0]   COUT,
    output wire[WIDTH-1:0]  RO,
    output wire[WIDTH-1:0]  IO
    );
    wire [WIDTH-1:0] tbI[0:1];
    wire [WIDTH-1:0] tbR[0:1];
    wire [WIDTH-1:0] fbI[0:1];
    wire [WIDTH-1:0] fbR[0:1];
    wire [WIDTH-1:0] passR[0:1];
    wire [WIDTH-1:0] passI[0:1];
    wire[0:1] C1,C2,C1_,C2_;
    wire[logn-1:0] CIN_;
    wire [logn-1:0] rot_product;
    SRL16xNE #(.WIDTH(logn)) CIN_HOLD(
            .C(C),
            .CE(CE),
            .D(CIN),
            .Q(CIN_),
            .A(CWIDTH-3)
            );
generate
```

```
        if(WHICH==0)
        begin: which_equals_zero
          reg [CWIDTH-1:0] PHASE_;
          reg [CWIDTH-1:0] PHASE_INC;
          always @(posedge C)
          begin
                if(&CIN)
                        PHASE_<=0;
                else if(~|CIN)
                begin
                        PHASE_<=PHASE;
                        PHASE_INC<=PHASE;
                end else
                        PHASE_<=PHASE_+PHASE_INC;
          end
          assign COUT=CIN_-({2'b11,{((logn-2)){1'b0}}});
        CORDIC
#(.WIDTH(WIDTH),.CWIDTH(CWIDTH),.BUFFER_CORSE(1'b0),.BUFFER_FINE(1'b1)) core(
          .C(C),
          .CE(CE),
          .R(R),
          .RI({RI[WIDTH-1],RI[WIDTH-1:1]}),
          .II({II[WIDTH-1],II[WIDTH-1:1]}),
          .ZI(PHASE),
          .RO(passR[0]),
          .IO(passI[0])
          );
        end else begin:which_greater_than_zero
```

```
if(logn-2>2*WHICH)
begin: tsk
   assign COUT=CIN_-({{WHICH{2'b0}},2'b11,{((logn-2-2*WHICH)){1'b0}}});
end else begin: tssk
   assign COUT=CIN_;
end
if(WHICH==1)
begin: WHICH_1
MUL3 #(.WIDTH(logn),.BUFFER(1'b0)) rot_mul(
   .C(C),
   .CE(CE),
   .R(R),
   .I0({2'b0,CIN[logn-1-2*WHICH:0]}),
   .P(CIN[logn+1-2*WHICH:logn-2*WHICH]),
   .O(rot_product)
   );
end else begin: WHICH_GREATER_THAN_1
MUL3 #(.WIDTH(logn),.BUFFER(1'b0)) rot_mul(
   .C(C),
   .CE(CE),
   .R(R),
   .I0({2'b0,CIN[logn-1-2*WHICH:0],{(WHICH-1){2'b0}}}),
   .P(CIN[logn+1-2*WHICH:logn-2*WHICH]),
   .O(rot_product)
   );
end
if(CWIDTH>logn)
begin: large_CWIDTH
```

```
        CORDIC
#(.WIDTH(WIDTH),.CWIDTH(CWIDTH),.BUFFER_CORSE(1'b0),.BUFFER_FINE(1'b1)) core(
          .C(C),
          .CE(CE),
          .R(R),
          .RI({RI[WIDTH-1],RI[WIDTH-1:1]}),
          .II({II[WIDTH-1],II[WIDTH-1:1]}),
          .ZI({-rot_product,{(CWIDTH-logn){1'b0}}}),
          .RO(passR[0]),
          .IO(passI[0])
          );
        end else begin: small_CWIDTH
        CORDIC
#(.WIDTH(WIDTH),.CWIDTH(CWIDTH),.BUFFER_CORSE(1'b0),.BUFFER_FINE(1'b1)) core(
          .C(C),
          .CE(CE),
          .R(R),
          .RI({RI[WIDTH-1],RI[WIDTH-1:1]}),
          .II({II[WIDTH-1],II[WIDTH-1:1]}),
          .ZI({-rot_product}),
          .RO(passR[0]),
          .IO(passI[0])
          );
        end
        end

        smart_fifo #(.WIDTH(2*WIDTH),.LOGN(logn-1-2*WHICH)) fifo_bfi_1(
                .C(C),
```

```
        .CE(CE),
        .D({tbR[0],tbI[0]}),
        .Q({fbR[0],fbI[0]})
        );
BFI #(.WIDTH(WIDTH),.BUFFER(1'b0)) bfi_1(
        .C(C),
        .CE(CE),
        .R(R),
        .frombufR(fbR[0]),
        .frombufI(fbI[0]),
        .tobufR(tbR[0]),
        .tobufI(tbI[0]),
        .inR(passR[0]),
        .inI(passI[0]),
        .outR(passR[1]),
        .outI(passI[1]),
        .C2(CIN_[logn-1-2*WHICH]),
        .C1(CIN_[logn-2-2*WHICH]),
        .C2_(C2_[0]),
        .C1_(C1_[0])
        );
smart_fifo #(.WIDTH(2*WIDTH),.LOGN(logn-2-2*WHICH)) fifo_bfii_1(
        .C(C),
        .CE(CE),
        .D({tbR[1],tbI[1]}),
        .Q({fbR[1],fbI[1]})
        );
BFII #(.WIDTH(WIDTH),.BUFFER(1'b0)) bfii_1(
```

```
.C(C),
.CE(CE),
.R(R),
.frombufR(fbR[1]),
.frombufI(fbI[1]),
.tobufR(tbR[1]),
.tobufI(tbI[1]),
.inR(passR[1]),
.inI(passI[1]),
.outR(RO),
.outI(IO),
.C2(C2_[0]),
.C1(C1_[0]),
.C2_(C2_[1]),
.C1_(C1_[1])
);

endgenerate

endmodule

module R22x16kFFT #(parameter FREQS=16,parameter OUTPUT_WIDTH=16,parameter CWIDTH=18,parameter logn=14,parameter INPUT_WIDTH=8,parameter LOG_BUF=10)(
    input wire                  C,
    input wire                  CE,
    input wire                  R,
    input wire                  START,
    input wire [CWIDTH-1:0]     PHASE,
```

```
    input wire [INPUT_WIDTH-1:0]    RI,
    input wire [INPUT_WIDTH-1:0]    II,
    output reg [OUTPUT_WIDTH-1:0]    RO,
    output reg [OUTPUT_WIDTH-1:0]    IO,
    output reg[logn-1:0]            CIN,
    output reg [logn-1:0]           COUT,
    output reg [logn-1:0]           RCOUT,
    output reg                      RE
    );
  reg HOLD=1;
  //reg [logn-1:0]    CIN=0;
  wire[logn-1:0]    CIN_[0:6];

  wire [0:6]    C1_;
  wire [0:6]    C2_;

  wire [OUTPUT_WIDTH-1:0] passR[0:logn+2];
  wire [OUTPUT_WIDTH-1:0] passI[0:logn+2];

  wire [logn-1:0]     revCOUT;
  wire [logn-1:0]    COUT_;
  reg [4:0] STATE=0;

  assign COUT_=CIN_[6]-3;
  assign
revCOUT={COUT_[0],COUT_[1],COUT_[2],COUT_[3],COUT_[4],COUT_[5],COUT_[6],COUT_[7],C
OUT_[8],COUT_[9],COUT_[10],COUT_[11],COUT_[12],COUT_[13]};
  localparam WAIT     =    4'h0;
```

```
localparam LOAD    =    4'h1;
localparam READ    =    4'h2;
always @(posedge C)
begin
   if(R)
   begin
          CIN<=0;
          HOLD<=1;
          RE<=0;
   end
   else
   begin
          if(START)
          begin
                  HOLD<=0;
          end
          if(!HOLD)
                  CIN<=CIN+1;
          COUT<=COUT_;
          RCOUT<=revCOUT;
          RO<=passR[6];
          IO<=passI[6];
          if(revCOUT==0)
                  RE<=1'b1;
   end
end

R4FFT_section #(.WHICH(0),.WIDTH(10),.CWIDTH(CWIDTH),.logn(logn)) R4FFT1(
```

```
		.C(C),.CE(CE),.R(HOLD),
		.CIN(CIN),.RI({1'b0,RI,{10-INPUT_WIDTH-1{1'b0}}}),.II({1'b0,II,{10-INPUT_WIDTH-
1{1'b0}}}),.PHASE(PHASE),
		.COUT(CIN_[0]),.RO(passR[0]),.IO(passI[0]));
		wire [logn-1:0] CIN2;
	R4FFT_section #(.WHICH(1),.WIDTH(11),.CWIDTH(CWIDTH),.logn(logn)) R4FFT2(
		.C(C),.CE(CE),.R(HOLD),
		.CIN(CIN_[0]),.RI({passR[0][9],passR[0][9:0]}),.II({passI[0][9],passI[0][9:0]}),.PHASE(0),
		.COUT(CIN_[1]),.RO(passR[1]),.IO(passI[1]));
	R4FFT_section #(.WHICH(2),.WIDTH(12),.CWIDTH(CWIDTH),.logn(logn)) R4FFT3(
		.C(C),.CE(CE),.R(HOLD),
		.CIN(CIN_[1]),.RI({passR[1][10],passR[1][10:0]}),.II({passI[1][10],passI[1][10:0]}
),.PHASE(0),
		.COUT(CIN_[2]),.RO(passR[2]),.IO(passI[2]));
	R4FFT_section #(.WHICH(3),.WIDTH(13),.CWIDTH(CWIDTH),.logn(logn)) R4FFT4(
		.C(C),.CE(CE),.R(HOLD),
		.CIN(CIN_[2]),.RI({passR[2][11],passR[2][11:0]}),.II({passI[2][11],passI[2][11:0]}
),.PHASE(0),
		.COUT(CIN_[3]),.RO(passR[3]),.IO(passI[3]));
	R4FFT_section #(.WHICH(4),.WIDTH(14),.CWIDTH(CWIDTH),.logn(logn)) R4FFT5(
		.C(C),.CE(CE),.R(HOLD),
		.CIN(CIN_[3]),.RI({passR[3][12],passR[3][12:0]}),.II({passI[3][12],passI[3][12:0]}
),.PHASE(0),
		.COUT(CIN_[4]),.RO(passR[4]),.IO(passI[4]));
	R4FFT_section #(.WHICH(5),.WIDTH(15),.CWIDTH(CWIDTH),.logn(logn)) R4FFT6(
		.C(C),.CE(CE),.R(HOLD),
		.CIN(CIN_[4]),.RI({passR[4][13],passR[4][13:0]}),.II({passI[4][13],passI[4][13:0]}
),.PHASE(0),
```

```
    .COUT(CIN_[5]),.RO(passR[5]),.IO(passI[5]));
R4FFT_section #(.WHICH(6),.WIDTH(16),.CWIDTH(CWIDTH),.logn(logn)) R4FFT7(
    .C(C),.CE(CE),.R(HOLD),
    .CIN(CIN_[5]),.RI({passR[5][14],passR[5][14:0]}),.II({passI[5][14],passI[5][14:0]}
),.PHASE(0),
    .COUT(CIN_[6]),.RO(passR[6]),.IO(passI[6]));

endmodule

`ifndef _smart_fifo_
`define _smart_fifo_
module smart_fifo #(parameter WIDTH=16,parameter LOGN=0)(
    input  wire C,
    input  wire CE,
    input  wire [WIDTH-1:0] D,
    output wire [WIDTH-1:0] Q
    );
genvar i;
generate
if(LOGN==0)
begin: fdre_case
FDRExN #(.WIDTH(WIDTH)) fdre(
    .C(C),
    .CE(CE),
    .R(1'b0),
    .D(D),
    .Q(Q)
    );
```

```
end
if(LOGN<5 && LOGN>0)
begin: srl_case
SRL16xNE #(.WIDTH(WIDTH)) srl(
   .C(C),
   .CE(CE),
   .A((1<<LOGN)-1),
   .D(D),
   .Q(Q)
   );
end
if(LOGN>=5)
begin: BRAM_case
reg [LOGN-1:0] out=1;
reg [LOGN-1:0] in =0;
 (* ram_style = "block" *)
 reg [WIDTH-1:0] mem[0:(1<<LOGN)];
 reg [WIDTH-1:0] regout;
 always @(posedge C)
 begin
   if(CE)
   begin
          out<=(out==(1<<LOGN)-1) ? 0 : out+1;
          in <=(in ==(1<<LOGN)-1) ? 0 : in+1;
          mem[in]<=D;
          regout<=mem[out];
   end
 end
```

```
 assign Q=regout;
end
endgenerate
endmodule
`endif
`ifndef _FDRExN_
`define _FDRExN_
module FDRExN #(parameter WIDTH=16)(
   input wire C,
   input wire CE,
   input wire R,
   input wire [WIDTH-1:0] D,
   output wire [WIDTH-1:0] Q
   );
genvar i;
generate
for(i=0;i<WIDTH;i=i+1)
begin: FDRE_block
FDRE fdre(
   .C(C),
   .CE(CE),
   .R(R),
   .D(D[i]),
   .Q(Q[i])
   );
end
endgenerate
endmodule
```

```
`endif
`ifndef _SRL16xNE_
`define _SRL16xNE_
module SRL16xNE #(parameter WIDTH=16)(
   input wire C,
   input wire CE,
   input wire [WIDTH-1:0] D,
   input wire [3:0] A,
   output wire [WIDTH-1:0] Q
   );
genvar i;
generate
for(i=0;i<WIDTH;i=i+1)
begin: SRL_block
SRL16E #(.INIT(16'h0)) srl16(
   .CLK(C),
   .CE(CE),
   .A0(A[0]),
   .A1(A[1]),
   .A2(A[2]),
   .A3(A[3]),
   .D(D[i]),
   .Q(Q[i])
   );
end
endgenerate
endmodule
`endif
```

//// CORDIC.v (gabriel)

```
`ifndef _CORDIC_
`define _CORDIC_
/*
   Implements a WIDTH word length, CWIDTH angular ward length vectoring mode cordic.
   if BUFFER_CORSE, the initial rough rotation is pipelined. if BUFFER_FINE, the CORDIC stages
are pipelined.

*/
module CORDIC #(parameter WIDTH=8,parameter CWIDTH=10,parameter
BUFFER_CORSE=0,parameter BUFFER_FINE=1)(
   input  wire C,
   input  wire CE,
   input  wire R,
   input wire [WIDTH-1:0] RI,
   input wire [WIDTH-1:0] II,
   input wire [CWIDTH-1:0] ZI,
   output wire [WIDTH-1:0]RO,
   output wire [WIDTH-1:0]IO
   );
parameter CONST = {
32'b001000000000000000000000000000000,
32'b000100101110010000000101000111011,
32'b000010011111101100111000010110110,
32'b000001010001000100010001110101000,
32'b000000101000101100001101010000110,
```

```
32'b00000101000101110101111111000010,
32'b00000010100010111101100000111100,
32'b00000000101000101111100010101010,
32'b00000000010100010111110010100110,
32'b00000000001010001011111001011101,
32'b00000000000101000101111100110000,
32'b00000000000010100010111110011000,
32'b00000000000001010001011111001100,
32'b00000000000000101000101111100110,
32'b00000000000000010100010111110011,
32'b00000000000000001010001011111001,
32'b00000000000000000101000101111100,
32'b00000000000000000010100010111110,
32'b00000000000000000001010001011111,
32'b00000000000000000000101000101111,
32'b00000000000000000000010100010111,
32'b00000000000000000000001010001011,
32'b00000000000000000000000101000101,
32'b00000000000000000000000010100010,
32'b00000000000000000000000001010001,
32'b00000000000000000000000000101000,
32'b00000000000000000000000000010100,
32'b00000000000000000000000000001010,
32'b00000000000000000000000000000101,
32'b00000000000000000000000000000010,
32'b00000000000000000000000000000001,
32'b00000000000000000000000000000000};
```

```
genvar i;
wire [WIDTH-1:0] RT1;
wire [WIDTH-1:0] IT1;

wire [CWIDTH-2:0] Z[CWIDTH-1:0];
wire [CWIDTH-1:0]   ZT1;
/*
   Rough rotation
*/
CORDIC_CORSE #(.WIDTH(WIDTH),.CWIDTH(CWIDTH),.BUFFER(BUFFER_CORSE)) cordic_corse(
   .C(C),
   .CE(CE),
   .R(R),
   .RI(RI),
   .II(II),
   .ZI(ZI),
   .RO(RT1),
   .IO(IT1),
   .ZO(ZT1)
   );

generate
if(CWIDTH<=WIDTH+2)
begin: narrow
/*
   if CWIDTH+2<=WIDTH, then we don't need to explicitly expand our word length
into "fractions" below the implied point
*/
```

```
wire [WIDTH-1:0] RT[CWIDTH+1:0];
wire [WIDTH-1:0] IT[CWIDTH+1:0];
assign RT[0]=RT1;
assign IT[0]=IT1;
for(i=0;i<CWIDTH-2;i=i+1)
begin: cordic_stages
CORDIC_STAGE #(.WIDTH(WIDTH),.BUFFER(BUFFER_FINE),.SHIFT(i),.CWIDTH(CWIDTH-
1),.CONST(CONST[32*32-1-i*32:32*32-1-i*32-CWIDTH+2])) c_stage
  (
  .C(C),
  .CE(CE),
  .R(R),
  .RI(RT[i]),
  .II(IT[i]),
  .ZI(Z[i]),
  .RO(RT[i+1]),
  .IO(IT[i+1]),
  .ZO(Z[i+1])
  );
end
assign RO=RT[CWIDTH-2][WIDTH-1:0];
assign IO=IT[CWIDTH-2][WIDTH-1:0];
end else
begin: wide
/*
  if CWIDTH-2>WIDTH, we need to explicitly expand our word length in order to have non-zero
terms in all stages.
```

```
   strictly speaking, we should probably do this anyway in order to maintain precision, but i'm
willing to accept crap precision in order to have a smaller design
*/
wire [CWIDTH-3:0] RT[CWIDTH+1:0];
wire [CWIDTH-3:0] IT[CWIDTH+1:0];
assign RT[0]={RT1,{(CWIDTH-2-WIDTH){1'b0}}};
assign IT[0]={IT1,{(CWIDTH-2-WIDTH){1'b0}}};
for(i=0;i<CWIDTH-2;i=i+1)
begin: cordic_stages
CORDIC_STAGE #(.WIDTH(CWIDTH-2),.BUFFER(BUFFER_FINE),.SHIFT(i),.CWIDTH(CWIDTH-
1),.CONST(CONST[32*32-1-i*32:32*32-1-i*32-CWIDTH+2])) c_stage
   (
   .C(C),
   .CE(CE),
   .R(R),
   .RI(RT[i]),
   .II(IT[i]),
   .ZI(Z[i]),
   .RO(RT[i+1]),
   .IO(IT[i+1]),
   .ZO(Z[i+1])
   );
end
assign RO=RT[CWIDTH-2][CWIDTH-3:CWIDTH-2-WIDTH];
assign IO=IT[CWIDTH-2][CWIDTH-3:CWIDTH-2-WIDTH];
end
endgenerate
assign Z[0]={1'b0,ZT1[CWIDTH-3:0]};
```

```
endmodule
`endif

//// CORDIC_STAGE.v (gabriel)

`ifndef _CORDIC_STAGE_
`define _CORDIC_STAGE_
/*
   one stage of CORDIC operation. pipelined if BUFFER, shifts by SHIFT, uses angular constant
CONST
*/
module CORDIC_STAGE #(parameter WIDTH=16, parameter CWIDTH=16,parameter
CONST=0,parameter SHIFT=0,parameter BUFFER=1)(
   input C,
   input CE,
   input R,
   input [WIDTH-1:0] RI,
   input [WIDTH-1:0] II,
   input [CWIDTH-1:0] ZI,
   output [WIDTH-1:0] RO,
   output [WIDTH-1:0] IO,
   output [CWIDTH-1:0] ZO
   );
wire AS_I=~ZI[CWIDTH-1];
wire [2:0] OVERFLOW;
AS_CE #(.WIDTH(CWIDTH),.BUFFER(BUFFER)) angle_adder(
   .C(C),
   .CE(CE),
```

```
    .R(R),
    .I0(ZI),
    .I1(CONST[CWIDTH-1:0]),
    .AS(AS_I),
    .BYP(1'b1),
    .O(ZO),
    .OVERFLOW(OVERFLOW[0])
    );
generate
if(SHIFT>0)
begin: case_0
AS_CE #(.WIDTH(WIDTH),.BUFFER(BUFFER)) real_adder(
    .C(C),
    .CE(CE),
    .R(R),
    .I0(RI),
    .I1({{SHIFT{II[WIDTH-1]}},II[WIDTH-1:SHIFT]}),
    .AS(AS_I),
    .BYP(1'b1),
    .O(RO),
    .OVERFLOW(OVERFLOW[1])
    );
AS_CE #(.WIDTH(WIDTH),.BUFFER(BUFFER)) imaginary_adder(
    .C(C),
    .CE(CE),
    .R(R),
    .I0(II),
    .I1({{SHIFT{RI[WIDTH-1]}},RI[WIDTH-1:SHIFT]}),
```

```
    .AS(ZI[CWIDTH-1]),
    .BYP(1'b1),
    .O(IO),
    .OVERFLOW(OVERFLOW[2])
    );
end else
begin: case_1
AS_CE #(.WIDTH(WIDTH),.BUFFER(BUFFER)) real_adder(
    .C(C),
    .CE(CE),
    .R(R),
    .I0(RI),
    .I1(II),
    .AS(~ZI[CWIDTH-1]),
    .BYP(1'b1),
    .O(RO),
    .OVERFLOW(OVERFLOW[1])
    );
AS_CE #(.WIDTH(WIDTH),.BUFFER(BUFFER)) imaginary_adder(
    .C(C),
    .CE(CE),
    .R(R),
    .I0(II),
    .I1(RI),
    .AS(ZI[CWIDTH-1]),
    .BYP(1'b1),
    .O(IO),
    .OVERFLOW(OVERFLOW[2])
```

```
   );
end
endgenerate

endmodule
`endif
`ifndef _CORDIC_CORSE_
`define _CORDIC_CORSE_
/*
   initial CORDIC stage, doing rotation by 0,90,180,270

   name is because i can't spell COARSE. >_<


*/
module CORDIC_CORSE #(parameter WIDTH=16,parameter CWIDTH=17,parameter BUFFER=0)(
   input C,
   input CE,
   input R,
   input [WIDTH-1:0] RI,
   input [WIDTH-1:0] II,
   input [CWIDTH-1:0] ZI,
   output [WIDTH-1:0] RO,
   output [WIDTH-1:0] IO,
   output [CWIDTH-1:0] ZO
   );
wire [WIDTH-1:0] RT;
wire [WIDTH-1:0] IT;
```

```
genvar i;
generate
if(BUFFER)
begin:buffer_z
for(i=0;i<CWIDTH;i=i+1)
begin: Z_delay_block
FDCE flop(
   .C(C),
   .CE(CE),
   .CLR(R),
   .D(ZI[i]),
   .Q(ZO[i])
   );
end
end
if(!BUFFER)
begin: no_buffer_z
assign ZO=ZI;
end
endgenerate
wire [1:0] OVERFLOW;
ROT_CE #(.WIDTH(WIDTH),.IMG(0),.BUFFER(BUFFER)) real_rot(
   .C(C),
   .R(R),
   .CE(CE),
   .RI(RI),
   .II(II),
   .Z(ZI[CWIDTH-1:CWIDTH-2]),
```

```
   .O(RO),
   .OVERFLOW(OVERFLOW[0])
   );
ROT_CE #(.WIDTH(WIDTH),.IMG(1),.BUFFER(BUFFER)) img_rot(
   .C(C),
   .R(R),
   .CE(CE),
   .RI(RI),
   .II(II),
   .Z(ZI[CWIDTH-1:CWIDTH-2]),
   .O(IO),
   .OVERFLOW(OVERFLOW[1])
   );
endmodule
`endif


//// ADDERS.v (gabriel)

`ifndef _AS_CE_ //Adder/Subtractor with clear and enable
`define _AS_CE_

/*
   ripple carry adder/subtractor/bypass unit.

   returns I[BYPASS] when byp is low, returns I0+I1 when AS is low, I0-I1 when AS is high

   if BUFFER== 0, no pipelining, otherwise one internal pipeline delay.
```

```
*/
module AS_CE #(parameter WIDTH=16,parameter BYPASS=0,BUFFER=1)(
   input C,
   input R,
   input CE,
   input [WIDTH-1:0] I0,
   input [WIDTH-1:0] I1,
   input AS,
   input BYP,
   output [WIDTH-1:0] O,
   output OVERFLOW
   );
genvar i;
wire [WIDTH:0] CI;
assign CI[0]=AS&BYP;
generate
for(i=0;i<WIDTH;i=i+1)
begin: FAS_generate
FAS_CE #(.BYPASS(BYPASS),.BUFFER(BUFFER))fas(
   .C(C),
   .R(R),
   .CE(CE),
   .I0(I0[i]),
   .I1(I1[i]),
   .CI(CI[i]),
   .AS(AS),
   .BYP(BYP),
   .O(O[i]),
```

```
   .CO(CI[i+1])
   );
end
endgenerate
assign OVERFLOW=CI[WIDTH];
endmodule
`endif

`ifndef _MUL3_
`define _MUL3_
/*
   returns I0*P, where P is 2 bits. one pipeline stage if BUFFER.
   used for twiddle factor generation
*/
module MUL3 #(parameter WIDTH=16,parameter BUFFER=1'b0)(
   input wire C,
   input wire CE,
   input wire R,
   input wire [WIDTH-1:0] I0,
   input wire [1:0] P,
   output wire [WIDTH-1:0] O
   );
genvar i;
wire [WIDTH:0] CI;
assign CI[0]=~|P;
wire [WIDTH:0] I1;
assign I1[0]=0;
assign I1[WIDTH:1]=I0;
```

```
generate
for(i=0;i<WIDTH;i=i+1)
begin: FAS_generate
MUL3_FA #(.BUFFER(BUFFER))mul(
            .C(C),
            .R(R),
            .CE(CE),
            .I0(I0[i]),
            .I1(I1[i]),
            .CI(CI[i]),
            .ADD(~(^P)),
            .SEL(P[0]),
            .O(O[i]),
            .CO(CI[i+1])
            );
end
endgenerate
endmodule
`endif
`ifndef _MUL3_FA_
`define _MUL3_FA_
/*
   Full Adder for MUL3
*/
module MUL3_FA #(parameter BUFFER=1'b0)(
   input wire C,
   input wire CE,
   input wire R,
```

```
    input wire I0,
    input wire I1,
    input wire ADD,
    input wire SEL,
    input wire CI,
    output wire CO,
    output O
    );
wire lut_out;
wire multand_out;

LUT4 #(.INIT(16'b0111_1110_1000_1110)) lut(
        .I0(I0),
        .I1(ADD),   // &P
        .I2(SEL),   // P[0]
        .I3(I1),
        .O(lut_out)
        );
MULT_AND multand(
        .I0(I0),
        .I1(ADD),
        .LO(multand_out)
        );
MUXCY_L muxcy(
        .DI(multand_out),
        .CI(CI),
        .S(lut_out),
        .LO(CO)
```

```
	);
wire xorcy_out;
XORCY_L xorcy(
	.LI(lut_out),
	.CI(CI),
	.LO(xorcy_out)
	);
generate
if(BUFFER)
begin:buffer
FDCE D_flipflop(
	.C(C),
	.CE(CE),
	.CLR(R),
	.D(xorcy_out),
	.Q(O)
	);
end
else
begin:no_buffer
assign O=xorcy_out;
end
endgenerate
endmodule
`endif
/*
FAS is a full adder/subtractor.
```

AS is
   0 for add
   1 for subtract

this is a pipelined adder, to take explicit advantage of local routing (in my work on this, basically everything ever is going to be maximally pipelined, including the cordics, to enable maximum clocking. As such, don't bother trying to do unpipelined shit, and don't try to add other pipelining unless you mean it.

*/

```
`ifndef _FAS_CE_    //full adder/subtractor with clear and enable
`define _FAS_CE_
module FAS_CE #(parameter BYPASS=0,parameter BUFFER=1)(
   input C,
   input R,
   input CE,
   input I0,
   input I1,
   input CI,
   input AS,
   input BYP,
   output O,
   output CO
   );
wire lut_out;
generate
if(BYPASS==0)
```

```
begin: bypass_0
LUT4 #(.INIT(16'b1010_0110_0110_1010)) lut(
   .I0(I0),
   .I1(BYP),
   .I2(AS),
   .I3(I1),
   .O(lut_out)
   );
end
else begin: bypass_1
LUT4 #(.INIT(16'b1011_0111_0100_1000)) lut(
   .I0(I0),
   .I1(BYP),
   .I2(AS),
   .I3(I1),
   .O(lut_out)
   );

end
endgenerate
wire multand_out;
MULT_AND multand(
   .I0(I0),
   .I1(BYP),
   .LO(multand_out)
   );
MUXCY_L muxcy(
   .DI(multand_out),
```

```
   .CI(CI),
   .S(lut_out),
   .LO(CO)
   );
wire xorcy_out;
XORCY xorcy(
   .LI(lut_out),
   .CI(CI),
   .O(xorcy_out)
   );

generate
if(BUFFER)
begin: pipelin
FDCE D_flipflop(
   .C(C),
   .CE(CE),
   .CLR(R),
   .D(xorcy_out),
   .Q(O)
   );
end
else
begin: no_pipeline
assign O=xorcy_out;
end
endgenerate
endmodule
```

```
`endif
`ifndef _MUXim_
`define _MUXim_
/*
   MUXim provides selectable multiply by -j (bus reverse and reversal
*/
module MUXim #(parameter WIDTH=16)(
   input [WIDTH-1:0] I0,
   input [WIDTH-1:0] I1,
   input C1,
   input C2,
   output [WIDTH-1:0] O
   );
genvar i;
generate
for(i=0;i<WIDTH;i=i+1)
begin: gen_loop
LUT4 #(.INIT(16'hCAAA)) lut(
   .I0(I0[i]),
   .I1(I1[i]),
   .I2(C1),
   .I3(C2),
   .O(O[i])
   );
end
endgenerate
endmodule
`endif
```

```
`ifndef _ROT_CE_
`define _ROT_CE_
/*
   provides rotation by 0,90,180,270
   IMG=1 if imaginary part, 0 if not
   if(BUFFER) gives one pipeline stage
*/
module ROT_CE #(parameter WIDTH=16,parameter IMG=0,parameter BUFFER=0)(
   input C,
   input R,
   input CE,
   input [WIDTH-1:0] RI,
   input [WIDTH-1:0] II,
   input [1:0] Z,
   output[WIDTH-1:0] O,
   output OVERFLOW
   );
wire CI[WIDTH:0];
genvar i;
generate
if(IMG)
begin: IMG_block
   assign CI[0]=Z[1];
   for(i=0;i<WIDTH;i=i+1)
   begin: IMG_FA
   ROT1_CE #(.BUFFER(BUFFER))rot_ce(
           .C(C),
           .R(R),
```

```
            .CE(CE),
            .I0(II[i]),
            .I1(RI[i]),
            .CI(CI[i]),
            .CO(CI[i+1]),
            .Z(Z),
            .O(O[i])
            );
    end
end else
begin: IMG_block
    assign CI[0]=^Z;
    for(i=0;i<WIDTH;i=i+1)
    begin: IMG_FA
    ROT0_CE #(.BUFFER(BUFFER))rot_ce(
            .C(C),
            .R(R),
            .CE(CE),
            .I0(RI[i]),
            .I1(II[i]),
            .CI(CI[i]),
            .CO(CI[i+1]),
            .Z(Z),
            .O(O[i])
            );
    end
end
endgenerate
```

```
endmodule
`endif
`ifndef _ROT0_CE_
`define _ROT0_CE_
/*
   Full adder for real component rotation
*/
module ROT0_CE #(parameter BUFFER=0)(
   input C,
   input R,
   input CE,
   input I0,
   input I1,
   input CI,
   input [1:0]Z,
   output O,
   output CO
   );
wire lut_out;
LUT4 #(.INIT(16'hC53A)) lut(
   .I0(I0),
   .I1(I1),
   .I2(Z[0]),
   .I3(Z[1]),
   .O(lut_out)
   );
wire xorcy_out;
XORCY xorcy(
```

```
    .LI(lut_out),
    .CI(CI),
    .O(xorcy_out)
    );
MUXCY_L muxcy(
    .DI(1'b0),
    .CI(CI),
    .S(lut_out),
    .LO(CO)
    );
generate
if(BUFFER)
begin:buffer
FDCE D_flipflop(
    .C(C),
    .CE(CE),
    .CLR(R),
    .D(xorcy_out),
    .Q(O)
    );
end
else
begin:no_buffer
assign O=xorcy_out;
end
endgenerate
endmodule
`endif
```

```
`ifndef _ROT1_CE_
`define _ROT1_CE_
/*
   Full adder for imaginary part rotation
*/
module ROT1_CE #(parameter BUFFER=0)(
   input C,
   input R,
   input CE,
   input I0,
   input I1,
   input CI,
   input [1:0]Z,
   output O,
   output CO
   );
wire lut_out;
LUT4 #(.INIT(16'h35CA)) lut(
   .I0(I0),
   .I1(I1),
   .I2(Z[0]),
   .I3(Z[1]),
   .O(lut_out)
   );
wire xorcy_out;
XORCY xorcy(
   .LI(lut_out),
```

```
    .CI(CI),
    .O(xorcy_out)
    );
MUXCY_L muxcy(
    .DI(1'b0),
    .CI(CI),
    .S(lut_out),
    .LO(CO)
    );
generate
if(BUFFER)
begin:buffer
FDCE D_flipflop(
    .C(C),
    .CE(CE),
    .CLR(R),
    .D(xorcy_out),
    .Q(O)
    );
end
else
begin:no_buffer
assign O=xorcy_out;
end
endgenerate
endmodule
`endif
```

```
/// BFI.v

`ifndef _BFI_
`define _BFI_
/*
   Radix 2**2 butterfly type 1

   frombuf* are data from the feedback buffer
   tobuf* are data to the feedback buffer

   C1_ and C2_ are the input control signals suitably delayed for use by BFII
*/
module BFI #(parameter WIDTH=16,parameter BUFFER=1)(
   input C,
   input CE,
   input R,
   input [WIDTH-1:0] inR,
   input [WIDTH-1:0] inI,
   input [WIDTH-1:0] frombufR,
   input [WIDTH-1:0] frombufI,
   input C1,
   input C2,
   output [WIDTH-1:0] tobufR,
   output [WIDTH-1:0] tobufI,
   output [WIDTH-1:0] outR,
   output [WIDTH-1:0] outI,
   output C1_,
   output C2_
```

```
    );
generate
if(BUFFER)
begin: buffer_controls
FDRE byp_fdre(
    .C(C),
    .CE(CE),
    .R(R),
    .D(C1),
    .Q(C1_)
    );
FDRE twi_fdre(
    .C(C),
    .CE(CE),
    .R(R),
    .D(~C2),
    .Q(C2_)
    );
end else
begin: no_buffer_controls
assign C1_=C1;
assign C2_=~C2;
end
endgenerate
wire [3:0] OVERFLOW;
AS_CE #(.WIDTH(WIDTH),.BYPASS(1'b1),.BUFFER(1'b0)) as_tobufR(
    .C(C),
    .CE(CE),
```

```
    .R(R),
    .I0(frombufR),
    .I1(inR),
    .AS(1'b1),
    .BYP(C2),
    .O(tobufR),
    .OVERFLOW(OVERFLOW[0])
    );
AS_CE #(.WIDTH(WIDTH),.BYPASS(1'b1),.BUFFER(1'b0)) as_tobufI(
    .C(C),
    .CE(CE),
    .R(R),
    .I0(frombufI),
    .I1(inI),
    .AS(1'b1),
    .BYP(C2),
    .O(tobufI),
    .OVERFLOW(OVERFLOW[1])
    );
AS_CE #(.WIDTH(WIDTH),.BYPASS(1'b0),.BUFFER(BUFFER)) as_outR(
    .C(C),
    .CE(CE),
    .R(R),
    .I0(frombufR),
    .I1(inR),
    .AS(1'b0),
    .BYP(C2),
    .O(outR),
```

```
    .OVERFLOW(OVERFLOW[2])
    );
AS_CE #(.WIDTH(WIDTH),.BYPASS(1'b0),.BUFFER(BUFFER)) as_outI(
    .C(C),
    .CE(CE),
    .R(R),
    .I0(frombufI),
    .I1(inI),
    .AS(1'b0),
    .BYP(C2),
    .O(outI),
    .OVERFLOW(OVERFLOW[3])
    );
endmodule
`endif

//// BFII.v



`ifndef _BFII_
`define _BFII_
`default_nettype none
/*
    Radix 2**2 butterfly type II

    frombuf* are data from the feedback FIFO
```

```
    tobuf* are data to the feedback FIFO

   C1_ and C2_ are copies of C1 and C2 suitably delayed for use by the next stage (taking into
account the effective delay of this stage.
*/
module BFII #(parameter WIDTH=16,parameter BUFFER=0)(
   input wire C,
   input wire CE,
   input wire R,
   input wire [WIDTH-1:0] inR,
   input wire [WIDTH-1:0] inI,
   input wire [WIDTH-1:0] frombufR,
   input wire [WIDTH-1:0] frombufI,
   input wire C1,
   input wire C2,
   output wire [WIDTH-1:0] tobufR,
   output wire [WIDTH-1:0] tobufI,
   output wire [WIDTH-1:0] outR,
   output wire [WIDTH-1:0] outI,
   output wire C1_,
   output wire C2_
   );
generate
if(BUFFER)
begin: buffer_controls
FDRE byp_fdre(
   .C(C),
   .CE(CE),
```

```
   .R(R),
   .D(~C1),
   .Q(C1_)
   );
FDRE twi_fdre(
   .C(C),
   .CE(CE),
   .R(R),
   .D(C2),
   .Q(C2_)
   );
end else
begin: no_buffer_controls
assign C1_=~C1;
assign C2_=C2;
end
endgenerate
wire Cc;
assign Cc=(C2 & C1);
wire [3:0] OVERFLOW;
wire [WIDTH-1:0] MUXEDinR;
wire [WIDTH-1:0] MUXEDinI;
MUXim #(.WIDTH(WIDTH)) MUXinR(
   .I0(inR),
   .I1(inI),
   .C1(C1),
   .C2(C2),
   .O(MUXEDinR)
```

```
    );
MUXim #(.WIDTH(WIDTH)) MUXinI(
    .I0(inI),
    .I1(inR),
    .C1(C1),
    .C2(C2),
    .O(MUXEDinI)
    );
AS_CE #(.WIDTH(WIDTH),.BYPASS(1'b1),.BUFFER(1'b0)) as_tobufR(
    .C(C),
    .CE(CE),
    .R(R),
    .I0(frombufR),
    .I1(MUXEDinR),
    .AS(1'b1),
    .BYP(C1),
    .O(tobufR),
    .OVERFLOW(OVERFLOW[0])
    );
AS_CE #(.WIDTH(WIDTH),.BYPASS(1'b1),.BUFFER(1'b0)) as_tobufI(
    .C(C),
    .CE(CE),
    .R(R),
    .I0(frombufI),
    .I1(MUXEDinI),
    .AS(~Cc),
    .BYP(C1),
    .O(tobufI),
```

```
   .OVERFLOW(OVERFLOW[1])
   );
AS_CE #(.WIDTH(WIDTH),.BYPASS(1'b0),.BUFFER(BUFFER)) as_outR(
   .C(C),
   .CE(CE),
   .R(R),
   .I0(frombufR),
   .I1(MUXEDinR),
   .AS(1'b0),
   .BYP(C1),
   .O(outR),
   .OVERFLOW(OVERFLOW[2])
   );
AS_CE #(.WIDTH(WIDTH),.BYPASS(1'b0),.BUFFER(BUFFER)) as_outI(
   .C(C),
   .CE(CE),
   .R(R),
   .I0(frombufI),
   .I1(MUXEDinI),
   .AS(Cc),
   .BYP(C1),
   .O(outI),
   .OVERFLOW(OVERFLOW[3])
   );
endmodule
`endif

//// FFT_coef.v
```

```
module FFT_coef #(parameter LOGN=4,parameter WIDTH=8,parameter FREQLEN=14)(
  input wire              C,
  input wire              CE,
  input wire              R,
  input wire [FREQLEN-1:0]    CIN,
  output reg[WIDTH-1:0]       COEF,

  output reg              WOE,
  input wire              RE,
  input wire [FREQLEN-1:0]    FREQ,
  input wire [WIDTH-1:0]      DATA
  );

localparam S_WAIT=3'h0;
localparam S_SIGN=3'h1;
localparam S_READ=3'h2;
localparam S_DONE=3'h3;

reg   [3:0]in_ptr=0;
reg   [3:0]out_ptr=0;
reg  s=0;
reg   we=0;
wire    [WIDTH+FREQLEN-1:0] dout;
wire  [LOGN-1:0] out_ptr_mux;
//assign out_ptr_mux = (CIN+1==dout[WIDTH+FREQLEN-1:WIDTH]) ? out_ptr+1 : out_ptr;

mydram_flip #(.LOGSIZE(LOGN),.WIDTH(WIDTH+FREQLEN)) mem(
```

```
   .clk(C),
   .s(s),
   .addr_in(in_ptr),
   .din({FREQ,DATA}),
   .we(we),
   .addr_out(out_ptr),
   .dout(dout)
   );
reg [2:0] STATE=0;
always @(posedge C)
begin
   if(&CIN)
   begin
          out_ptr<=0;
   end else
   begin
          if(CIN==dout[WIDTH+FREQLEN-1:WIDTH])
          begin
                 out_ptr<=out_ptr+1;
                 COEF<=dout[7:0];
          end else
          begin
                 COEF<=0;
          end
   end
   case(STATE)
   S_WAIT:begin
          if(RE)begin
```

```
                STATE<=S_SIGN;
                WOE<=1;
                end
        end
S_SIGN:begin
                STATE<=S_READ;
                WOE<=0;
        end
S_READ:begin
        if(&in_ptr)begin
                        we<=0;
                        STATE<=S_DONE;
                        in_ptr<=0;
                end else
                begin
                        if(~we) begin
                        we<=1;
                        in_ptr<=0;
                        end else
                                in_ptr<=in_ptr+1;
                end
        end
default:begin
        if(&CIN)begin
                        s<=~s;
                        STATE<=S_WAIT;
                end
        end
```

```
   endcase
end
endmodule
module mydram_d #(parameter LOGSIZE=14, WIDTH=1)
         (input wire [LOGSIZE-1:0] addr_in,
         input wire [LOGSIZE-1:0] addr_out,
         input wire clk,
         input wire [WIDTH-1:0] din,
         output  [WIDTH-1:0] dout,
         input wire we);
  // let the tools infer the right number of BRAMs
  (* ram_style = "distributed" *)

  reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
  assign  dout = mem[addr_out];
  always @(posedge clk) begin
         if (we) mem[addr_in] <= din;
  end
endmodule
module mydram_flip #(parameter LOGSIZE=14, WIDTH=1)
         (input wire [LOGSIZE-1:0] addr_in,
         input wire [LOGSIZE-1:0] addr_out,
         input wire s,
         input wire clk,
         input wire [WIDTH-1:0] din,
         output wire  [WIDTH-1:0] dout,
         input wire we);
mydram_d #(.LOGSIZE(LOGSIZE+1), .WIDTH(WIDTH)) mem(
```

```
    .clk(clk),
    .addr_in({s,addr_in}),
    .addr_out({~s,addr_out}),
    .din(din),
    .dout(dout),
    .we(we)
    );
endmodule


/// SAMPLER.v
`default_nettype none

module SAMPLE_rom #(parameter DATA_WIDTH=8,parameter OUTPUT_WIDTH=16,parameter
ADDR_WIDTH=7, parameter SAMPLE_RANGE=14,parameter DEVICE_ID=8'h1,parameter
ADDR=7'h0,parameter INITIAL = 0)(
    input wire              C,
    input wire              CE,
    input wire              R,
    input wire              ready,
    inout wire[OUTPUT_WIDTH-1:0]    data_out,
    output wire             RE_out,



    input wire [7:0]        control,
    input wire              control_start,
    input wire              control_RIE
```

```
   );
    wire[SAMPLE_RANGE-1:0]    addr_sample;
    wire [DATA_WIDTH-1:0]    data_sample;
SAMPLE_mod
#(.DATA_WIDTH(DATA_WIDTH),.OUTPUT_WIDTH(OUTPUT_WIDTH),.ADDR_WIDTH(ADDR_W
IDTH),.SAMPLE_RANGE(SAMPLE_RANGE),.START(0),.DEVICE_ID(DEVICE_ID),.CADDR(ADDR))
sample(
   .C(C),
   .CE(CE),
   .R(R),
   .ready(ready),
   .data_out(data_out),
   .RE_out(RE_out),
   .addr_sample(addr_sample),
   .data_sample(data_sample),
   .control(control),
   .control_start(control_start),
   .control_RIE(control_RIE)
   );
mybrom #(.WIDTH(DATA_WIDTH),.LOGN(SAMPLE_RANGE-1),.INITIAL(INITIAL)) rom(
   .C(C),
   .addr(addr_sample),
   .data(data_sample)
   );
endmodule
module sampler_4(
   input wire C,
   input wire CE,
```

```
input wire R,
input wire ready,
input wire [7:0] control,
input wire control_start,
input wire control_RIE,

output wire [15:0]    data0,
output wire           re0,
output wire [15:0]    data1,
output wire           re1,
output wire [15:0]    data2,
output wire           re2,
output wire [15:0]    data3,
output wire           re3
);
wire[7:0] data_sample0;
wire[13:0] addr_sample0;

SAMPLE_mod #(.DEVICE_ID(8'h0),.SAMPLE_RANGE(13),.CADDR(7'b0)) bass_mod (
        .C(C),.CE(CE),.R(R),.ready(ready),
        .control(control),.control_start(control_start),.control_RIE(control_RIE),
        .addr_sample(addr_sample0),.data_sample(data_sample0),
        .data_out(data0),.RE_out(re0)
        );
bass bass_sample(
        .clka(C),
        .addra(addr_sample0),
        .douta(data_sample0)
```

```
	);
wire[7:0] data_sample1;
wire[12:0] addr_sample1;
SAMPLE_mod #(.DEVICE_ID(8'h0),.SAMPLE_RANGE(12),.CADDR(7'b1)) snare_mod (
	.C(C),.CE(CE),.R(R),.ready(ready),
	.control(control),.control_start(control_start),.control_RIE(control_RIE),
	.addr_sample(addr_sample1),.data_sample(data_sample1),
	.data_out(data1),.RE_out(re1)
	);
snare snare_sample(
	.clka(C),
	.addra(addr_sample1),
	.douta(data_sample1)
	);

wire[7:0] data_sample2;
wire[12:0] addr_sample2;
SAMPLE_mod #(.DEVICE_ID(8'h0),.SAMPLE_RANGE(13),.CADDR(7'd2)) open_mod (
	.C(C),.CE(CE),.R(R),.ready(ready),
	.control(control),.control_start(control_start),.control_RIE(control_RIE),
	.addr_sample(addr_sample2),.data_sample(data_sample2),
	.data_out(data2),.RE_out(re2)
	);
hatopen hatopen_sample(
	.clka(C),
	.addra(addr_sample2),
	.douta(data_sample2)
	);
```

```
   wire[7:0] data_sample3;
   wire[12:0] addr_sample3;
   SAMPLE_mod #(.DEVICE_ID(8'h0),.SAMPLE_RANGE(11),.CADDR(7'd3)) closed_mod (
        .C(C),.CE(CE),.R(R),.ready(ready),
        .control(control),.control_start(control_start),.control_RIE(control_RIE),
        .addr_sample(addr_sample3),.data_sample(data_sample3),
        .data_out(data3),.RE_out(re3)
        );
   hatclosed hatclosed_sample(
        .clka(C),
        .addra(addr_sample3),
        .douta(data_sample3)
       );
endmodule
module SAMPLE_mod #(parameter DATA_WIDTH=8,parameter OUTPUT_WIDTH=16,parameter
ADDR_WIDTH=7,parameter TRIG_WIDTH=1, parameter SAMPLE_RANGE=13,parameter
START=0,parameter END=4096,parameter DEVICE_ID=8'h1,parameter
CADDR=7'h0,RADDR=7'h1)(
   input wire           C,
   input wire           CE,
   input wire           R,
   input wire           ready,
   inout wire[OUTPUT_WIDTH-1:0]   data_out,
   output wire          RE_out,

   inout wire[SAMPLE_RANGE-1:0]   addr_sample,
   input wire [DATA_WIDTH-1:0]   data_sample,
```

```
    input wire [7:0]        control,
    input wire              control_start,
    input wire              control_RIE
    );
reg trig;
reg [7:0] packet[0:1];
reg [1:0] state=2'b10;
wire queue;
wire [SAMPLE_RANGE-1:0] mem_in;
wire [SAMPLE_RANGE-1:0] mem_out;
wire [SAMPLE_RANGE-1:0] mem_out_;
reg  [SAMPLE_RANGE-1:0] mem_hold;
reg  [15:0] acc;
reg  [15:0] out;
wire active_in,active_out,onehot_in,onehot_out;
reg [1:0] active_hold;
assign data_out=out;
assign RE_out=1'b1;
genvar i;
generate
for(i=0;i< SAMPLE_RANGE; i=i+1)
begin: mem_srls
SRL16E mem_srl(
    .CLK(C),
    .CE(trig),
    .D(mem_in[i]),
    .Q(mem_out_[i]),
    .A0(1'b0),.A1(1'b1),.A2(1'b1),.A3(1'b1)
```

```
   );
FDRE mem_latch(
   .C(C),
   .R(R),
   .CE(trig),
   .D(mem_out_[i]),
   .Q(mem_out[i])
   );
end
endgenerate
FDRSE queue_latch(
   .C(C),
   .CE(CE),
   .D(queue),
   .Q(queue),
   .S(((state==2'b01) && control_RIE) ? 1'b1 : 1'b0),
   .R(queue&~active_out&trig)
   );
SRL16E active_srl(
   .CLK(C),
   .CE(trig),
   .D(active_in),
   .Q(active_out),
   .A0(1'b1),.A1(1'b1),.A2(1'b1),.A3(1'b1)
   );
SRL16E onehot_srl(
   .CLK(C),
   .CE(CE),
```

```
   .D(ready),
   .Q(onehot_out),
   .A0(1'b1),.A1(1'b1),.A2(1'b1),.A3(1'b1)
   );
assign mem_in = (~active_out & queue) ? 0 : mem_out+1;
assign active_in = ((&mem_in) ? 1'b0 : active_out) | queue;
always @(posedge C)
begin
   if(ready)
   begin
         trig<=1'b1;
         acc<=0;
         out<={acc[8:0],7'b0};
   end else begin
         if(onehot_out)
         begin
               trig<=1'b0;
         end
         mem_hold<=mem_in;
         active_hold<={active_hold[0],active_out & trig};
         if(active_hold[1])
         begin
               acc<=acc+{{8{data_sample[7]}},data_sample};
         end
   end
end

assign addr_sample = mem_hold;
```

```
always @(posedge C)
begin
   if(R)
   begin
                              state<=2'b10;
   end
   else if(CE)
   begin
          if((control_start & control_RIE) | ((state==2'b11) && control_RIE))
          begin
                    if(&(DEVICE_ID^~control))
                              state<=2'b00;
          end else if(control_start) begin
                    state<=2'b11;
          end else if((state==2'b00) && control_RIE)
          begin
                    if(&(control[ADDR_WIDTH:0]^~CADDR))
                    state<=2'b01;
                    else
                    state<=2'b10;
          end else if((state==2'b01) && control_RIE)
          begin
                    state<=2'b10;
          end
   end
end
endmodule
```

//// testbenches submitted in verilog