

FpgaL Studio

Yusuf Young

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
ybyoung@mit.edu

David Onadeko

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
donadeko@mit.edu

Abstract—We present a design for a Digital Audio Workstation (DAW) capable of recording, editing, mixing, and producing audio. There will be four audio tracks where the initial audio can be created. The tracks are capable of defining instrument patterns for synths, drums, flutes, and bass. Additionally, each audio track can be augmented with a customizable chain of effects such as a 4-band equalizer, compressor, delay, limiter, and gain. The audio editing process will be accompanied by visual pages that show progress. Our goal is to streamline the music production process with the FPGA. **REPOSITORY:**

Index Terms—Digital Audio Workstation.

I. INTRODUCTION

A Digital Audio Workstation commonly referred to a DAW is an electronic device capable of audio editing, recording, and production. Although DAW's can have a variety of uses, one of the most common applications is in music production, with many DAWs being specialized for this use case. For our project, we aim to do just this and produce a DAW specialized for music production on the Real Digital Urbana FPGA board.

II. PHYSICAL CONSTRUCTION & PERIPHERALS

The DAW consists of three potentiometer voltage dividers connected to an ADC, a 24 bit speaker, and a SD card for storage of samples for playback. The three potentiometers are how the user navigates through the various screens and interacts editors. The purpose of the knobs changes depending on what page the user is on. For example, if the user is creating a beat pattern the first knob controls the measure, another knob controls the pitch of the note, and the last one controls the position of the note. However, if the user is on an edits effect page and adjusting the compression of a track, the knob will be used to control the threshold.

The user also uses various switches and buttons on the FPGA board to interact with the program.

- Switches[15:12]: These switches, when active, are used to bypass the chain effects module for each of the four instrument tracks (Synth, Bass, Percussion, and Flute).
- Switches[11]: This switch, when active, is used to bypass the master chain effects module from the combined audio output.
- Switches[15:0]: When on the effects order page, all switches are used to specify the desired effect order.
- Switches[6:0]: When on the effects edit page, these switches in conjunction with the knobs control the individual effect parameters.

- Button[3]: Serves as our system reset
- Button[2]: Play/pause button
- Button[1]: Exit/previous button
- Button[0]: Enter/next button

III. AUDIO EDITING

The `audio_editor` module implements the user-interface state machine that controls both the 8×8 beat sequencer for each instrument track (percussion, synth, bass, flutes) and all global audio-effect settings. It receives debounced button presses, switch inputs, and three rotary-knob values driven by potentiometers, and translates them into read/write operations to instrument BRAM, parameter updates for the DSP effects, and navigation between UI pages.

a) Sequencer editing.: Instrument patterns are stored locally in a buffer `local_data[7:0][7:0]`, where each 8-bit entry represents note or trigger information for a particular measure and eighth note. When the user selects an instrument, the module loads 64 bytes from BRAM (`LOADINGDATA`) into `local_data`. In the instrument edit states (`DRUMEDIT`, `SYNTHEDIT`, etc.), pressing enter toggles a specific bit chosen by knob positions `measure`, `eighth_note`, and `pitch_select`. Pressing exit stores the edited pattern back to BRAM (`SAVINGDATA`). The array is continually copied into `display_data_out_all` for on-screen rendering.

b) Effect navigation.: From any instrument page, the user may enter the effects subsystem (`EFFECTEDIT`), which branches into:

- `EFFECTORDER`: reorder the six global effects,
- `EFFECTPARAMETER`: choose which effect to configure.

A combinational signal `update` pulses whenever the user confirms a change, allowing the DSP pipeline to latch new settings.

c) Effect ordering.: Five 3-bit registers (`order0..order4`) encode a permutation of the effect IDs (EQ, compressor, limiter, delay, gain). These are directly exported as `effects_order0..4`. The user adjusts them using the front-panel switches; the module enforces validity and recomputes the last position so that exactly one of each effect appears in the chain.

d) Effect parameter editing.: Each effect has its own configuration state, entered from `EFFECTPARAMETER`. The module updates the DSP parameters according to knob or switch positions:

- EQEDIT: sets 3-bit band gains band1 . . 4.
- COMPRESSOREDIT: sets compressor threshold and ratio.
- LIMITEREDIT: sets limiter threshold.
- DELAYEDIT: sets delay depth.
- GAINEDIT: selects discrete amplification levels.

When the user presses `enter`, the chosen parameters are written into the corresponding output registers and propagated to the effects chain.

e) Summary.: Overall, `audio_editor` provides a complete UI abstraction layer: it manages pattern memory, effect configuration, and mode switching, while exposing clean, clock-synchronous outputs for the downstream audio-processing modules. All effect-order and effect-parameter logic is centralized here, enabling a fully modular and user-programmable audio pipeline.

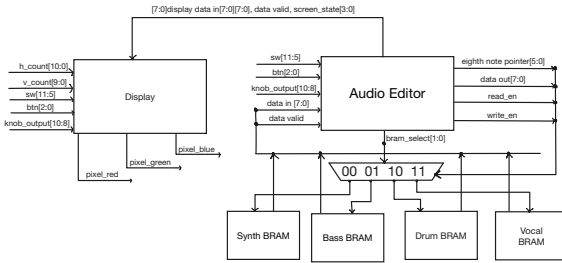


Fig. 1. Audio Editing and Display Pipeline

IV. AUDIO PLAYBACK

To playback audio from the DAW, samples must be selected and filtered based on user specification. This consists of 2 main steps:

- 1) Retrieval of Samples from the SD card based on the current state of the song
 - Retrieval of Indicator Variables from the Editor BRAM
 - Retrieval of Samples from the SD Card
 - Packing and Sending of reconstructed sample out to the chain effects
- 2) Processing and output of audio samples

A. SD Card Data Retrieval

The Data Readout module is responsible for retrieving audio data for playback. It interacts with both the Indicator BRAM and SD Card to retrieve and provide data for Audio Processing. The module stores an internal eight-note pointer that corresponds to which of the 64 eight-note segments of the song (8 eight-note heights of the 8 measures) it is currently retrieving. It also stores 4 FIFOs, each of which holds data for an instrument track. When enabled, it cycles through two

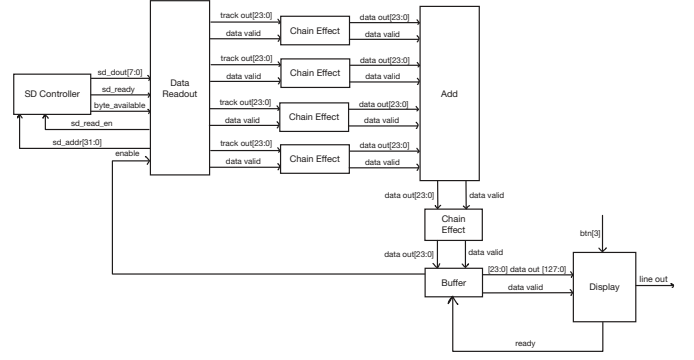


Fig. 2. Audio Playback Pipeline

stages, the BRAM_READ stage and the SAMPLE_FETCH stage.

- **BRAM_READ:** In the BRAM_READ stage, the module sends a read request to the BRAMs to retrieve the indicator variable stored for that eight-note. This determines which of the pitches or percussion instruments are active at the current eight-note for each of the 4 instrument tracks. It then increments the eighth note pointer to be ready for the next BRAM read and goes into the sample fetch stage.
- **SAMPLE_FETCH:** In the SAMPLE_FETCH, the module does a round robin read from the SD card into each of the FIFOs using the SD Controller IP. A round robin read is necessary to achieve our target rate. The address we read to is calculated based on the current instrument being read and the indicator variable retrieved for that instrument at this eighth note. Each of the 256 (2^8) possible combinations of pitches for an instrument is stored in adjacent locations of the SD card to facilitate this. Since we must read 512 bytes at a time, the address is incremented by 512 each round robin read cycle. Once all 96 packets are received for each instrument, the module returns to the BRAM_READ stage.

This process will continuously loop through all the eighth notes.

The final audio output we desire is comprised of 24-bit samples, so as we retrieve bytes from the SD card, we need to reconstruct these samples by continuously reading from the 4 FIFOs each cycle if every FIFO has data, and then, in parallel, grouping each instrument bytes into three and outputting 4 24-bit samples at a time, one for each instrument. We also send a 4-bit data out valid signal with this data for use by the Chain Effects module.

B. Audio Processing Output

1) Audio Buffer: The `Audio_buffer` module implements a ping-pong audio buffer between a fast producer (an SD card in our state) and a slower consumer running at the same clock but effectively lower sample rate. It stores

incoming 24-bit audio samples in on-chip BRAM and then replays them later at a controlled rate, allowing the system to read audio from memory in bursts while streaming it out smoothly to the audio pipeline. The output signal `begin_audio_production` is sent out when the `Audio_buffer` is ready for the next batch of data.

Internally, the module instantiates a Xilinx true dual-port RAM configured for 24-bit data and `DEPTH` locations. Port A is used for writing incoming samples, addressed by `input_sample_pointer`, while Port B is used for reading samples, addressed by `output_sample_pointer` plus an offset of `DEPTH/2`. This offset effectively splits the RAM into two halves and realizes a double-buffer (ping-pong) scheme: while one half is being written with new audio data, the other half can be read out for playback. Both pointers are $\lceil \log_2(\text{DEPTH}) \rceil$ -bit counters and naturally wrap around modulo `DEPTH`.

On every clock cycle, if `data_valid_in` is asserted, the current `audio_in` sample is written into the BRAM at address `input_sample_pointer`, and the write pointer is incremented. On the read side, a 9-bit counter `counter` runs continuously; whenever it rolls over to zero, it generates a strobe that controls when the next output sample is produced. If `request_data` is asserted at that instant, the module increments `output_sample_pointer`, captures the BRAM output `temp_audio_out` into `audio_out`, and sets `data_valid_out` high to indicate a valid audio sample. If `request_data` is low, the read pointer is held, `audio_out` is driven to zero, and `data_valid_out` still pulses, indicating that the consumer has been serviced with a zero sample instead of advancing the buffer. The strobe period (here determined by the width of `counter`) effectively sets the output sample rate relative to the system clock.

The signal `begin_audio_production` is asserted whenever the write and read pointers are equal. In the intended usage, this condition indicates that enough data has been buffered (the writer has caught up to the current read half), so the rest of the system can safely begin or continue audio playback without underflow. On reset, all internal counters and pointers are cleared, and BRAM writes are disabled until new valid input data arrives.

2) *Audio Driver*: The `audio_driver` module implements a simple serial audio interface driver (I²S-style) that converts 24-bit parallel audio samples into a bitstream suitable for our speaker. It generates the bit clock (BCLK), the left/right channel clock (LRCLK), and the serial data line (DIN), while also driving fixed control pins `SD_MODE` and `GAIN` for the codec configuration.

The module is parameterized by the audio sample rate (`SAMPLE_RATE`), the bits per sample (`BITS_PER_SAMPLE`, set to 24), and the system clock frequency (`CLK_FREQ`). Internally, a local parameter `BCLK_DIV` controls the division between the system clock `clk` and the generated bit clock BCLK. A 5-bit counter `bclk_counter` counts system clock cycles; when it reaches `BCLK_DIV - 1`, it resets and toggles the internal bit clock `bclk_int`, which is then exposed as

BCLK. On every rising transition of `bclk_int` (detected inside the same always block), a 6-bit `bit_counter` is incremented. After 32 bit periods, `bit_counter` rolls over to zero and the internal left/right clock `lrclk_int` toggles, so LRCLK changes state once every 32 bits: one 32-bit word per half-frame.

The data path is implemented using a 32-bit shift register `shift_reg`. On each high-to-low transition of LRCLK (detected by comparing `lrclk_prev` and `lrclk_int`), the module loads the current 24-bit signed `audio_data` into the upper bits of `shift_reg` with 8 trailing zero bits:

$$\text{shift_reg} \leftarrow \{\text{audio_data}, 8'd0\},$$

so that a 24-bit sample is padded to a 32-bit word. This transition marks the start of a new left-channel word in the serial stream.

To serialize the data, the module samples the falling edge of BCLK. An edge detector (using `bclk_prev` and `bclk_int`) asserts `bclk_falling` when BCLK transitions from high to low. On each falling edge, as long as `shift_bit_count` is less than 32, the most-significant bit of `shift_reg` is driven onto DIN and the register is shifted left by one bit:

$$\text{DIN} \leftarrow \text{shift_reg}[31],$$

$$\text{shift_reg} \leftarrow \{\text{shift_reg}[30:0], 1'b0\}.$$

The `shift_bit_count` counter tracks the number of bits transmitted for the current word. After 32 bits have been sent, DIN is forced low until the next load at the following LRCLK transition.

V. AUDIO PROCESSING

A. ChainEffects

1) *ChainEffects*: The `ChainEffects` module implements a fully reconfigurable audio effects chain. It takes a single 24-bit audio stream and a set of user-programmable effect parameters (EQ, compressor, limiter, delay, gain) and routes the signal through these effects in an order specified at run time by the user interface (`effects_order0..4`). The module therefore serves as the central hub that connects all individual effect blocks into a flexible, user-defined processing pipeline.

Effect order configuration. The six inputs `effects_order0..5` encode a permutation of six effect IDs:

ID	Effect
0	EQ
1	Compressor
2	Limiter
3	Delay
4	Gain

These IDs are collected into the internal array `effects_order[0..4]`. As a safety check, the module verifies that the user-provided order is valid by testing

$$\text{effects_order0} + \dots + \text{effects_order4} = 10,$$

which is the sum of $\{0, 1, 2, 3, 4\}$. If this condition fails, the effect order is reset to the default sequence

```
effects_order = {0, 1, 2, 3, 4},
```

so that the chain still operates in a known, linear order. In either case, index 5 is reserved for the “output stage” (no effect), and `audio_out` is always taken from `staged_audio[5]`.

Staging buffers and control signals. To support arbitrary reordering, the module uses a set of per-effect staging registers:

- `staged_audio[0..5]`: 24-bit audio buffers for each logical stage.
- `staged_valid_in[0..5]`: `data_valid` flags for each stage input.
- `all_current_stages[0..5]`: small integers indicating the current “stage index” associated with the data at each effect.

At the beginning of each processing step, the incoming sample and its validity flag are injected into the first effect in the chain:

```
staged_audio[effects_order[0]] ← audio_in,
staged_valid_in[effects_order[0]] ← data_valid_in,
all_current_stages[effects_order[0]] ← 0.
```

The downstream effects then pass audio and metadata from one stage index to another according to their individual `next_stage` outputs.

Individual effects and parameter inputs. The module instantiates one instance of each effect:

- **EQ** (EQ): takes per-band gains `band1..4`, applies a multi-band equalizer, and outputs `EQ_out_audio`, `valid_EQ`, and a 3-bit `stage_EQ_out`.
- **Compressor** (compressor): takes `threshold_comp` and `ratio`, performs dynamic range compression, and returns `Comp_out_audio`, `valid_Comp`, and `stage_Comp_out`.
- **Limiter** (limiter): clips or limits the signal based on `threshold_lim`.
- **Delay** (delay): delays the signal by a number of samples given by `delay_amount`.
- **Gain** (gain): applies a programmable amplification factor from `amplify_in`.

Each effect receives:

```
audio_in = staged_audio[id],
data_valid_in = staged_valid_in[id],
cur_stage = all_current_stages[id],
```

where `id` is the effect’s fixed index (0 for EQ, 1 for compressor, etc.). Each effect, in turn, produces:

- a processed audio sample (`X_out_audio`),
- a validity flag `valid_X`,
- a 3-bit `stage_X_out` indicating the next *logical stage index* this data should move to.

Dynamic routing through the chain. The core of `ChainEffects` is the routing logic that maps logical “stage indices” to concrete module instances using `effects_order`. For each effect, a `next_stage` index is interpreted as:

“Send my output to the effect at chain position `stage_X_out`”.

This is implemented by writing the effect’s output into:

```
staged_audio[effects_order[stage_X_out]] ← X_out_audio,
staged_valid_in[effects_order[stage_X_out]] ← valid_X,
all_current_stages[effects_order[stage_X_out]] ← stage_X
```

for $X \in \{\text{EQ, Comp, Lim, Delay, Gain}\}$. Thus the combination of `stage_X_out` and `effects_order[]` determines which physical effect module will receive the next sample, allowing arbitrary permutations of the chain.

The final “sink” of the chain is the reserved index 6. Once an effect outputs `next_stage = 3'b101` (logical stage 5), its audio is placed into `staged_audio[5]`, which is then exposed as the module output:

```
audio_out ← staged_audio[5],
data_valid_out ← staged_valid_in[5].
```

Overall behavior. On each clock cycle (when not in reset), `ChainEffects`:

- 1) Injects the new input sample into the first effect in the user-specified order.
- 2) Lets each effect consume its input from the corresponding `staged_audio` entry, apply its internal processing (using the current EQ/compressor/limiter/delay/gain parameters), and emit an output sample with an associated `next_stage`.
- 3) Routes each effect’s output to the next module in the chain using `effects_order` and the logical stage index.
- 4) Collects the final processed sample at stage 5 and drives it onto `audio_out` with the corresponding `data_valid_out`.

This design cleanly separates *what* each effect does from *where* it sits in the signal chain. By encoding the effect order and next-stage indices in small integers, the same instantiations of EQ, compressor, limiter, delay, and gain can be wired together in any order at run time, providing a flexible and efficient effects rack entirely in hardware.

B. Equalizer (EQ)

The EQ module implements a four-band finite-impulse-response (FIR) equalizer. Given a 24-bit input sample and four signed 3-bit gain controls (`band1_amp` to `band4_amp`), it splits the signal into low, low-mid, high-mid, and high frequency bands, scales each band independently, and sums them back together.

Internally, each band is realized as a 128-tap FIR filter (as outlined in slide 107 of lecture 12) with

hard-coded precomputed coefficients (`low*`, `lowMid*`, `highMid*`, `high*`). For each incoming sample, the module performs an accumulated multiply-add across these taps, storing the partial sums in four 34-bit pipelines (`audio_pipe_low`, `audio_pipe_low_mid`, `audio_pipe_high_mid`, `audio_pipe_high`). In parallel, it maintains `valid_pipe` and `stage_pipe` shift registers to delay the input `data_valid_in` flag and `cur_stage` value by the filter latency (`STAGES_NEEDED` = 128).

At the final tap, the band outputs are scaled by their respective gains and normalized via a right shift:

```
adjusted_low = band1_amp(audio_pipe_low[127]
                        >> 10),
adjusted_low_mid = band2_amp(audio_pipe_low_mid[127]
                             >> 10),
adjusted_high_mid = band3_amp(audio_pipe_high_mid[127]
                              >> 10),
adjusted_high = band4_amp(audio_pipe_high[127]
                          >> 10).
```

The final 24-bit output is the sum of all four bands:

```
audio_out = adjusted_low[23:0] +
            adjusted_low_mid[23:0] +
            adjusted_high_mid[23:0] +
            adjusted_high[23:0].
```

The module asserts `data_valid_out` when the filtered sample emerges from the pipeline and increments `next_stage = cur_stage + 1` so that the `ChainEffects` block can route the sample to the next effect.

C. Compressor

The compressor module reduces the dynamic range of the signal by attenuating samples that exceed a programmable threshold. A signed 24-bit threshold `threshold` defines the onset of compression, and a 3-bit parameter `ratio` controls the strength of compression via a right shift (division by 2^{ratio}). For input samples $|x| \leq T$, the output is unchanged; for samples above the threshold, the amount exceeding T is compressed:

$$y = \begin{cases} T + \frac{x - T}{2^r}, & x > T, \\ x, & -T \leq x \leq T, \\ -T + \frac{x + T}{2^r}, & x < -T, \end{cases} \quad (1)$$

where x is the input sample, T is the threshold, and r is the compression ratio parameter. The module asserts `data_valid_out` and increments `next_stage` for each valid input, allowing compressed samples to flow directly into the next effect in the chain.

D. Limiter

The limiter module prevents clipping by hard-clamping the signal magnitude to a programmable threshold. For each 24-bit signed input sample, the module computes the absolute value and compares it against a positive threshold `threshold`. If the magnitude exceeds this threshold, the output is forced to $\pm T$ with the same sign as the input; otherwise, the sample passes through unchanged:

$$y = \begin{cases} -T, & x < -T, \\ x, & -T \leq x \leq T, \\ T, & x > T, \end{cases} \quad (2)$$

where x is the input and T is the limiter threshold. On each valid input, the module produces a limited output sample, asserts `data_valid_out`, and increments `next_stage` so that the signal can continue through the effect chain without exceeding the allowed amplitude range.

E. Delay

The delay module creates an echo effect by mixing the current sample with a delayed copy stored in block RAM. A 12-bit write pointer `cur_ptr` advances through a circular buffer, writing the incoming 24-bit sample while simultaneously reading a past sample at an address offset by `delay_amount`. When `delay_amount` is zero, the output is simply the (pipelined) input; otherwise, the module forms

$$y_{\text{delay}}[n] = \begin{cases} x[n], & D = 0, \\ x[n] + x[n - D], & D > 0, \end{cases} \quad (3)$$

where $x[n]$ is the input sample and D is the programmable delay in samples. The design includes a small pipeline to align the read-out delayed sample with the original input, then asserts `data_valid_out` and increments `next_stage` so the echoed signal can be passed to subsequent effects.

F. Gain

The gain module applies a simple linear amplification to the 24-bit signed audio sample and performs saturation to avoid overflow. The 2-bit control input `amplify_in` is zero-extended to a signed 3-bit scalar `amplify`, which is multiplied by `audio_in` to form a 32-bit product. On each clock cycle, if `data_valid_in` is asserted, the module checks whether the product lies within the representable 24-bit range $[-2^{23}, 2^{23} - 1]$; if so, the lower 24 bits of the product are passed through, otherwise the output is clipped to the positive or negative limit. The `next_stage` counter is incremented by one to indicate that the sample has passed through the gain stage, and `data_valid_out` simply forwards `data_valid_in`.

VI. INTERACTIVE EDITOR DISPLAY

The `editor_display` module is responsible for generating the visual user interface of the audio editor. It converts the current editor state, effect parameters, and sequencer data into per-pixel RGB values (`pixel_red`,

pixel_green, pixel_blue) based on the current screen coordinates `h_count` and `v_count`. Internally, it stores an 8×8 grid of pattern data (`local_data`), updated whenever `display_data_in_valid` is asserted, and uses this grid to visualize which eighth notes are active for the selected measure, instrument, and pitch. The current state and `instrument_state` are cast into a page-state enum to control what is drawn on screen.

Two main sprite generators are instantiated. The `measure_sprite` block draws the active measure, note, and pitch region using the pipelined pixel counters, highlighting the currently selected eighth note and pitch from the knob inputs. The `screen_sprites` block draws higher-level UI “nodes” such as instrument selection boxes, effect blocks with a selector (`screen_nodes_sel`) choosing which layout to display based on the current page (e.g. HOME, DRUM, SYNTH, EFFECTEDIT, etc.) and displaying the respective sprites by retrieving their pixel values from a BRAM. The mem file specifying the pixel values are generated by a script which at a given address, stores a one or zero for each of the 16 possible sprites depending on if that address’ corresponding pixel is black or white. Depending on what sprites should be displayed on a given screen, the `screen_sprites` module indexes into the respective bits in the retrieved BRAM data. The top portion of the screen is also filled with a color-coded header bar (red for drums, green for synth, blue for vocals, etc.) to indicate the currently active instrument or master channel.

For effect-edit pages, the module provides simple visualizations of the effect parameters: in `EQEDIT` it draws four vertical regions whose brightness encodes the gains `band1-band4`; in `COMPRESSOREDIT` and `LIMITEREDIT` it renders a vertical “threshold” box whose height depends on `threshold_comp` or `threshold_lim`, together with a ratio indicator based on `ratio`; in `DELAYEDIT` it draws a bar whose size depends on `delay_amount`; and in `GAINEDIT` it highlights one of several stacked regions depending on `amplify_in`. The final pixel color at each location is a bitwise combination of the measure sprite, instrument header, and screen-node overlays, masked by simple “active” flags so that only the relevant layers are visible on a given page.”

VII. DESIGN EVALUATION

The following is a breakdown of the physical resources that the FPGA uses to build our project:

- Total LUTs: 26659
- Logic LUTs: 25710
- LUT Rams: 832
- SRLs: 117
- FFs: 17705
- RAMB36: 47
- RAMB18: 0
- DSP Blocks: 88

We can consider what sort of latency that our project might experience. The SD card infrastructure and the chain effect modules both run on a 25MHz clock. Additionally, once the

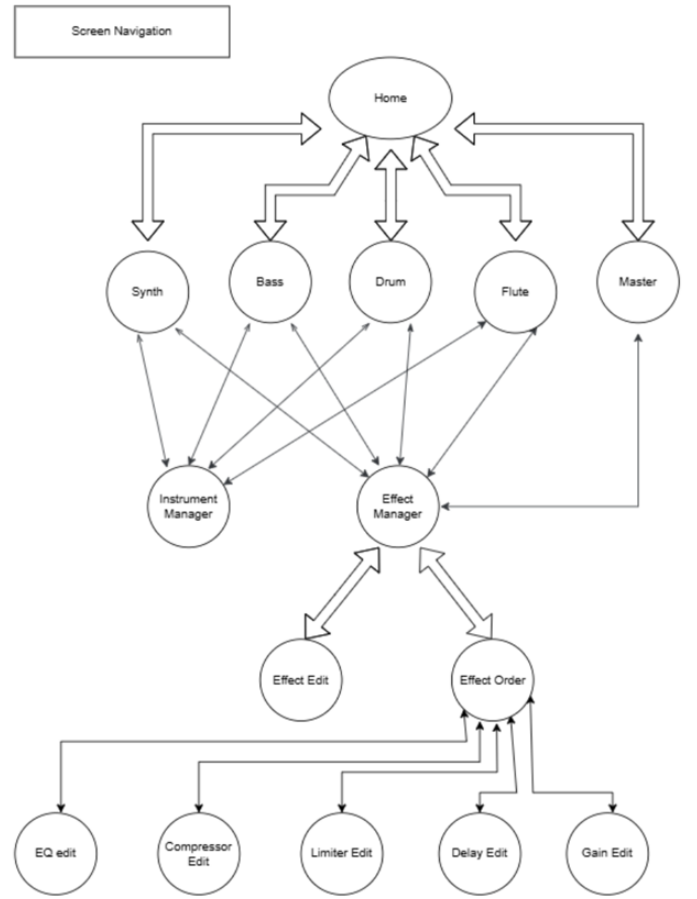


Fig. 3. Graph of Screen Connections

SD card modules start outputting data, we receive a valid 24 bit audio sample once every three cycles, as the SD card outputs bytes at a time. The total latency of the chain effects modules is 135 cycles. The majority of this latency comes from audio samples flowing through the pipeline in the EQ. We then store these audio samples in the output buffer. We produce audio one eighth note at a time, which corresponds to 16384 audio samples. However, since we are only outputting data at 44.1khz, there is no scenario where the next eighth note of audio is not ready when the current eighth note has been output.

The only time a user would experience noticeable latency is when utilizing the play/pause button. Audio output can only be started/stopped at the beginning of an eighth note. Our audio playback is at 80 bpm, so a singular eighth note lasts for a duration of 0.665 seconds, which is the maximum delay between pressing the pause button and audio output stopping.

VIII. IMPLEMENTATION INSIGHTS

A. Different EQ Implementation

Currently, the EQ computes a 128 sample long dot product for each of the four frequency bands. These operations are done in parallel so there is not critical path issue, however,

this utilizes a lot of resources. In the future, it would be more efficient to split up the computation of these dot products between multiple cycles. The chain effects modules only receive valid audio data once every 3 cycles at most. So if some of that dead space was use for computation, it would only require approximately a third of the resources that it currently does.

IX. AUTHOR CONTRIBUTION

Yusuf:

- Yusuf was responsible for handling the audio effect logic as well as the final data output buffer and addition logic. He also was responsible for handling the editor display logic for the effect edit pages as well as the audio editor logic for the effect state transitions and effect editing. He was responsible for communication with the potentiometers, and speaker. He also helped write the Physical Construction and Peripherals, Audio Editing, Audio Processing Output, Audio Processing, Design Evaluation, and Implementation insights sections of the paper as well as handled the final editing of the Final Report.

David:

- David was responsible for handling the data readout logic as well as the BRAM interactions in the audio editor and editor display logic design. He was also responsible for handling the selection sprite and instrument edit display logic in the editor display as well as the audio editor logic for instrument state transitions and instrument editing. he also helped write the Audio Playback, Audio Editing, and Author Contribution sections of the paper and the Code Provenance as well as handled the final editing of the Final Video.