shattersample

Jun Hayakawa (jundh2) and Nathan Shin (nsshin2)

Introduction

Our project, named shattersample, was intended to function similar to a granular synthesizer. However, its design was modified to contain the added uniqueness of being a texture-based synthesizer / sampler / glitch machine with all of the available settings and randomization of those settings. It produces sound by using a library of samples stored on a microSD card, and chopping these samples into tiny segments (called grains) and reordering / changing them depending on the configuration of the settings. This allows for a wide range of timbres that can be played from a given audio sample. Additionally, the synthesizer features 3 playback modes (granular, sampler, and arpeggiator).

Written Description of shattersample

Navigating the user interface:

All parameters are interfaced using the buttons, switches, LEDs, and hex display on the FPGA. The hex display shows the name of the parameter or parameters to modify, and the LEDs above the switches show their current value. To change a parameter, set the switches to the desired value and press the leftmost button on the FPGA. The LEDs will update to reflect the new value of the parameter. To navigate between parameters, press the inner left and right buttons BTN2 and BTN1.

The hex display will show either one or two parameters at a time. Parameters which come in pairs are 8-bit values (except ARP, but more on that later) represented in binary. The parameter on the left hex display corresponds to LEDs / switches 15 – 8, while the parameter on the right corresponds to LEDs / switches 7 – 0. Solo parameters are reserved for those with a smaller range of values and range from 0 to 16. For these parameters, only one switch is needed to set its value. The highest-value switch selected indicates the parameter's value, and it, along with all LEDs to its right, light up to show this selection. In this way, the switches act as a quasi-slider.

Shattersample controls :

Position and Spray: Position and Spray, abbreviated as POS and SPr, are used to determine the starting position of a grain that is played. Both settings have 8-bit values that range from 0-255 (in decimal), where a position setting of 0 represents playing the sample from the beginning and 255 represents playing it at the end. The spray setting acts as a randomizer of the position setting, as it offsets the position per grain by a random value ranging from 0 - spray/2. For example, if the position setting is 150 and the spray setting is 200, the position of a single grain can range from 50 to 250.

Density and Density Randomization: These settings are abbreviated as dEnS and rAnd, and their purpose is to control the rate at which grains are played. Both settings have 8-bit values that range from 0-255 (in decimal). A higher density corresponds to more grains being played

while lower means less. The randomization function adds an offset to the density setting itself, where it offsets it by a random value ranging from 0 - rand/2.

Length and Length Randomization : LEn and rAnd are used to control the length of an individual grain. The length randomization setting functions the same way the other randomization settings. Both settings have 8-bit values that range from 0-255 (in decimal).

Arpeggiate and Flip: Arpeggiate and Flip are two settings that are independent from each other. On the UI, it is abbreviated as ArP. and FLIP. ArP can either be on or off, and if it is on, upon pressing multiple keys on the keyboard, the notes will randomly fluctuate back and forth. The flip setting has an 8-bit value ranging from 0-255 (in decimal), where the value represents the probability that a sample sound that will be played is read reverse, producing a reversed audio grain. The probability is determined by flip/255, so if the value is 255, every grain will play reverse audio.

Sound: This parameter, abbreviated as SOUnd., select which sample to play grains from. This setting only regards the most significant bit that is high. For example, if switch 15 is flipped, regardless of the other switches, the setting will return the decimal value 16 in bitwise form, which on the display would show all the LEDs lit. This setting is indexed 0-16, which means that a total of 17 samples can be chosen.

Loudness: In addition to a volume setting that may already exist within the external audio device being used to play grains, the loudness setting, abbreviated as LOUdnESS, is also indexed the same way as sound select. The value of this setting ranges from 0-16, where 0 means it is muted and 16 is full volume.

Sampler Mode= Abbreviated as Sannple, this setting disables all other settings apart from sound select, and in this setting, the entirety of a sample can be played as opposed to the maximum of 1.5 seconds in “synth” mode. (When length is at max value)

Playing Notes :

In order to produce sound from a given sound sample indicated by sound select, a keyboard must be connected to the FPGA device. Typical models that are compatible are the Dell Wired Keyboard or the AmazonBasics Keyboard. Keycodes within the keyboard are mapped to certain notes, similar to on a piano or any other music instrument. The figure below depicts the mapped keys to notes. Keep in mind that C4 will play the original sample’s pitch/speed. The range of the keyboard is C1 - B5, where the octave up key will increase the octave by one and octave down will decrease it by one.

Loading custom sounds:

Loading custom sounds onto the microSD card is a straightforward albeit somewhat tedious process and requires the free programs HxD and Audacity. Audio is stored on the SD card in 10-second-long clips of 8-bit unsigned stereo PCM data with a 44.1kHz sample rate. To format your desired samples, load them into Audacity and export the project. Make sure to export each track separately and specify the audio format to be raw headerless 8-bit stereo PCM at 44.1kHz.

The starting sectors in memory of sounds are selected as a function of the sound index (from 0 to 16) where sector = 2800 x sound# + 350. To store data directly into specific sectors of memory, the program HxD is required to bypass the SD card's file system. First, open your audio data exported from Audacity using Notepad for Windows or TextEdit for macOS. It will appear as a massive string of ASCII characters. Copy this string and then reopen HxD. Select a block at least 3800 sectors away from your target address and then go to Edit -> Select Block and then choose the starting address to be your target address. You can now paste your audio data into the memory block and it will be able to be accessed on the FPGA.

Provided Modules

Although most of the project files were made completely from scratch, some files were used as a template and were then modified to produce the desired outcome.

SDCard.vhd

This file was provided and its purpose is to give the necessary signals within the state machine of the SD card to read the data that is stored. It was slightly modified to be able to work on SD cards of 32GBs. It is in charge of sending/receiving SPI signals to/from the SD card, and returns bytes of data to the main SD FSM.

Sdcard_init.sv

This file is a state machine for the SD card, which handles retrieving and transmitting a requested block of SD card memory. In its original state when we downloaded it, the FSM was configured to read the entirety of SD card memory, from address 0x0 to the maximum RAM address. Because this module was so heavily altered from its original state, its functionality will be discussed in the module description section of the report.

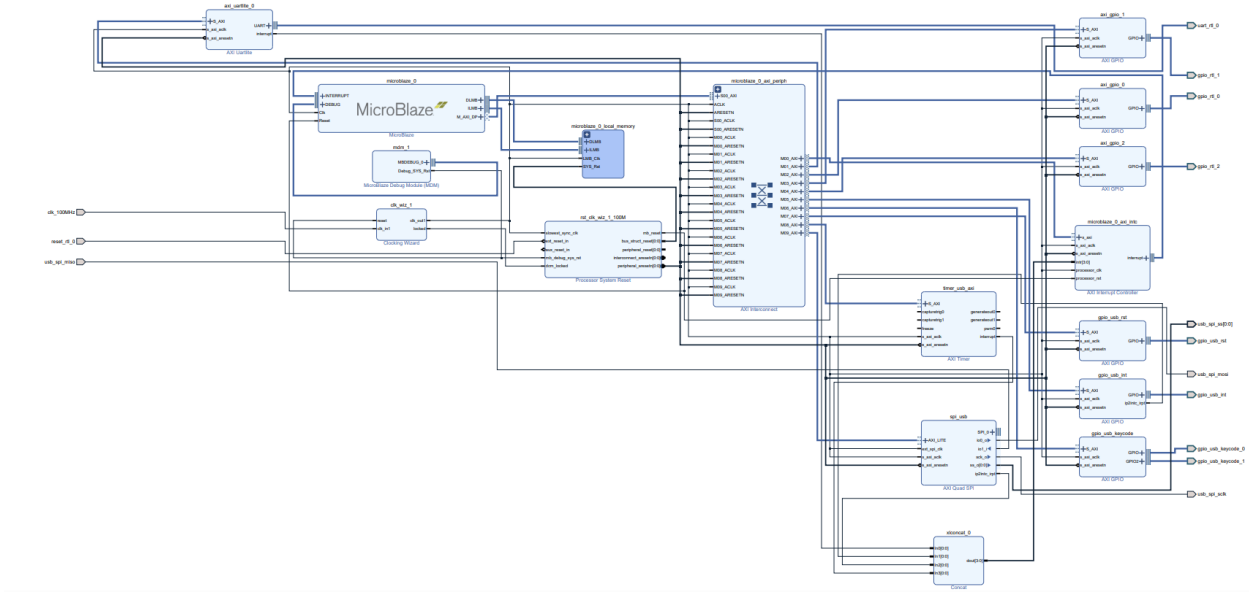
design_1.bd
This block diagram is very similar to the one that was created in lab 6.2 with the only difference being that the HDMI signals were all removed, as well as the addition of GPIO's that would input and output registers that corresponded to certain settings that would be randomized using the MicroBlaze.

hex_driver.sv
The code to convert hex values was provided but was modified to include the letters S, r, L, U, n, P, P., d., and "NULL" , and h. Originally it only used numbers 0-F in hexadecimal.

sync.sv
Used to synchronize the buttons. This code was provided and was not modified.

Block Diagrams
Due to the fact that the block diagram of the top level design within Vivado is so large, a single picture could not encapsulate the entirety of the diagram. The block diagram for the top level instantiations (excluding the microblaze block diagram) is shown in this link.

Below is an image of the block diagram of the MicroBlaze which is used to randomize the position, length, and density settings as well as interface with the USB keyboard.



Module Descriptions

SV Modules

eight_to_one_MUX.sv

Inputs: [15:0] A, [15:0] B, [15:0] C, [15:0] D, [15:0] E, [15:0] F, [15:0] G, [15:0] H, [3:0] S

Outputs: [15:0] Z, [31:0] letters

Description: The 8-1 MUX serves as a display so that the user can know which setting they are modifying and what setting they are currently on. The output “letters” of the mux is used as an input in the modified hex display module (that was provided) so that the appropriate setting can be displayed on the hex grids, while the output “Z” represents the values that are to be displayed on the LED’s so that the user can know what the current configuration for a given setting is.

Purpose: To display the current setting as well as the current values of the setting.

sync.sv

Inputs: Clk, d

Outputs: q

Description: Used to synchronize buttons 0-3.

Purpose: To reduce the static hazards that occur when selecting between different settings and modes within the synthesizer. Overall, it does not change the functionality but makes the interface much smoother.

setting_state_machine.sv

Inputs: Clk, sampler_mode, [3:0] buttons,

Outputs: [3:0] grid_in, LD1,LD2,LD3,LD4,LD5,LD6,LD7,LD8,LD9,LD10, initial_clear

Description: The control unit for the setting interface, note that for this project, a different state machine was created for both reading of the microSD card and for the UI of the synthesizer. The control unit contains a state for each window of settings along with the different modes such as the sampler_mode. Each window has a setting state, a change state, and right and left states for when the user wants to navigate between different settings. The state machine assigns load values to the corresponding settings depending on the current window, along with assigning which letters that are to be displayed on the hex grids. The control unit also takes inputs of the buttons, where btn[0] is used to switch between sampler_mode and synth mode, btn[1] and btn[2] are used to switch between different windows, and btn[3] is used to confirm a configured setting.

Purpose: To assign the appropriate load signals as well as display to be shown on hex grids and to also incorporate the buttons 1 and 2 that are used to switch between settings.

bit8_register.sv

Inputs: Clk, Reset, Write_En,[7:0] D, [3:0] reset_init, sampler_mode

Outputs:[7:0] Data_Out

Description: A modified 8-bit register that takes in values of either Switches[15:8] or [7:0], depending on the setting, as well as hardcoded reset_init values, and sampler_mode to determine the output value of the setting register which will be used to implement modification to the settings. Sampler_mode indicates whether or not to set the settings to a preset setting, and reset_init sets a certain binary value to each register in a unique way on each reset, because if every setting was set to bitwise 8’b0 for each setting, the quality of the sound would be diminished.

Purpose: To store 8-bit values for settings that need to have 8-bit values, as well as modifying those values depending on the mode or the state described by the control unit for the settings.

bit16_register_modified.sv

Inputs: Clk, Reset, Write_En, [15:0] D,
Outputs: [4:0] Data_Out

Description: A modified register that takes in 16 bits but essentially returns the decimal value 0-16 in bitwise form, hence why Data_Out has a bit length of only 5 bits. It only regards the most significant bit that is high. For example, if switch 15 is flipped, regardless of the other switches, the data_out will return the decimal value 16 in bitwise form, but in the display all the LEDs will be lit. (this is taken care of in the eight_one_mux module)

Purpose: To store values of registers specifically for the settings of volume and to select which sound to be sampled.

sdcard_init.sv

Inputs: [7:0] length, [7:0] start, [4:0] sound_select, clk50, GO, reset, sampler_mode, reverse, ram_op_begun, miso_i

Outputs: [31:0] ram_address, [15:0] ram_data, [7:0] outs, ram_we, ram_init_error, ram_init_done, cs_bo, sclk_o, mosi_o

Description: This module is where the majority of processing action happens. sdcard_init is responsible for returning the requested block of data from the sd card in the proper fashion. 8-bit inputs start and length and 5-bit input sound_select control the position and length of data transmitted using a function we designed.

The function to calculate the starting position (in sectors of memory) is sector_offset = (ss * 2800) + 350 + (start_incr * 673 / 100), where ss is sound_select and start_incr is start, and the function to calculate the end point is MAX_RAM_ADDRESS = (length_incr * 250 / 100 + 15) + sector_offset, where length_incr is length. The reason these intermediate variables are used rather than the inputs themselves is to prevent changes in the input values from modifying the behavior of the FSM during audio playback. Whenever audio playback begins, sound_select, start and length are locked into ss, start_incr and length_incr, respectively.

If sampler mode is active, the function is applied differently. Start and length are ignored, and the function for the start position becomes sector_offset = (ss * 2800) + 350, while the maximum address becomes MAX_RAM_ADDRESS = 1720 + sector_offset, where 1720 is a magic number which corresponds to approximately 10 seconds of audio playback. Lastly, in the case that reverse is active, the start and end positions are swapped.

Data reads from the sd card are triggered by the GO signal, which is pulsed whenever a grain should be played. When GO is pulsed, the values of length, start, sound_select, sampler_mode, and reverse are locked in to intermediate variables, and the FSM goes through an initialization phase before looping through its block read/write states until it reaches the ending address, at which point it enters an idle state until the next trigger. In the initialization phase, the

VHDL SD SPI FSM is reset and its new starting address is loaded into it. In its read/write phase, it gathers 16 bits of data one byte at a time into its output variable ram_data, and then waits to receive a ready signal from the FIFO it writes to before making its write enable signal, ram_we, high. If playing forwards, it then increments the ram address, and if playing backwards it decrements. It then returns to the first state of this phase and repeats until it reaches the maximum ram address.

In the top level, the following connections were made. clk50 is connected to clk50, reset is connected to a constant zero signal called fake_reset, ram_we is connected to wren, the write enable of our FIFOs. ram_address is connected to a variable of the same name, but is never used because our FIFO based design had no need for it. Ram_data is connected to ram_data, ram_op_begun is connected to rready1 or rready2, the not full signals of our two FIFOs. Ram_init error and ram_init_done are unused.SPI signals mosi_o, miso_i, sclk, and cs_bo are all made external and connected to the corresponding pins of the SD card reader. GO is connected to a variable called trigger, which is the trigger signal of our design. Length and start are connected to variables length_mod and pos_mod, respectively, which are the values from the length and position registers after they have been affected by randomization. Lastly, sampler_mode and reverse are connected to variables of the same name.

Purpose: This module is instrumental to our design, and is how we are able to index the memory of the SD card at such a fine level.

edging.sv

Inputs: clk, signal, reset

Outputs: aligned

Description: This module is a positive edge detector, accepting a 1 bit input signal and returning a 1-bit output signal that goes high for exactly one clock pulse on the positive edge of the input. It does this by keeping track of the input value over two clock cycles, and makes the output signal 1 only when the current value of the input is 1 but the previous value was 0, indicating a positive edge. We use this module on our sampler mode button toggle, our read enable signal, and our read clock signal.

Purpose: A lot of our signals need to be pulsed to work effectively, such as signals tied to physical buttons on our FPGA / keyboard and our variable clocks. For operations which are performed as long as a signal is high, using the raw signal inputs for these operations can lead to them being performed many many more times than intended as a result of the signal being high for hundreds or thousands of clock cycles. The purpose of the edging module is to turn these relatively long signals into brief pulses.

impulse.sv

Inputs: [15:0] density, clk, reset,
Outputs: go

Description: This module generates the automatic trigger signals which play grains when the synth is set to granular mode. It is essentially a clock divider with much more flexibility. The clock division count is determined as a function of the density input given by max = ((256-density) * 380000) + 2500000. This function sets the maximum density to have a period of about 50ms, while the minimum density has a period of about 1.5 seconds. A count variable is continually incremented from 0 to the maximum count value. When it reaches this value, the go signal is pulsed for one clock cycle and the count is reset to zero. In the top level, the density input is connected to the dens_mod, the randomized value of the density register.

Purpose: This module is necessary to be able to change the density of grains in real time.

note2clk.sv

Inputs: [7:0] keycode, clk
Outputs: [15:0] count, read_en

Description: This module is simple but powerful, allowing us to play musical notes using the keyboard. All it does is create a case statement which checks the keycode received and outputs a read enable signal and a clock rate which corresponds to a pitch in the chromatic scale. Letter keys Q through B are mapped to two octaves, with note values increasing left to right and top to bottom, like a book. The octave range of the keys are controlled by the *octaves.sv* module, but the total range available is C1 to B5. By default, the D key corresponds to C4, and doesn't alter the pitch of the original sample, in order to have a sort of center point. In order to produce the correct pitches, we needed a function which translates a note frequency to a clock period relative to a 50mhz master clock and when sampled at 44.1khz. The function which does this is f(x) = (50 * 10^6 * 261.63 / 44100) * (1 / x), which scales the sample rate of audio playback by a certain number of semitones relative to middle C when their frequencies are passed as inputs. By putting each note of a two octave range into this function, we were able to find the clock division count required to reproduce them all. Additionally, as long as a note is held down, the module outputs a read enable signal, which is ANDed with the read clock pulse so that the FIFO reads data out at the appropriate rate only when keys are pressed.

Purpose: This module allows us to play our synthesizer live using a keyboard, and is integral to our design. The width of the keyboard makes playing complex melodies surprisingly intuitive.

octaves.sv

Inputs: [15:0] count, [7:0] keycode, clk
Outputs: [15:0] count_mod

Descriptions: This module controls the active octave range of the keyboard using the up and down arrows. It does a couple things to do this. First, this module contains the exact same functionality of the *edging.sv* module, and creates positive edge signals from the up and down arrow keycodes. When these signals are created, a 2-bit variable "octave" which can vary from 0 to 3 is incremented when the up arrow is pressed and decremented when the down arrow is pressed. This octave variable is used to alter note values / clock divider count. By default, octave = 10, which corresponds to no change to the count. When octave = 01, the count is multiplied by 2, which corresponds to lowering the octave by 1. When octave = 00, the count is multiplied by 4, which lowers the octave by 2. Lastly, when octave = 11, the count is divided by 2, which raises the octave by 1. This module provides the last modification of the read count necessary, so the output of this module count_mod is used as the read clock for the FIFOs.

Purpose: This module quadruples the range of the keyboard, and makes it so that one can play samples in any range and further distort sounds by pitching them drastically.

hex_driver.sv

Inputs: clk, Reset, [3:0] in[4],
Outputs: [7:0] hex_seg, [3:0] hex_grid

Description: The hex_driver module was one of the provided files, but modifications had to be made to add certain letters to the interface. Specifically the letters S, r, L, U, n, P, P., d., and "NULL" , and h. This was achieved by figuring out which bits within the module corresponded to each segment on the display.

Purpose: To display the letters on the hex display that would indicate which setting the user was currently on.

arpeggiator.sv

Inputs: [31:0] notes, [7:0] arpeggiate, trigger, sampler_mode
Outputs: [7:0] note

Description: This module allows for note arpeggiation. Because we ran out of buttons, we created an 8 bit register accessed by the switches to turn the arpeggiator on and off. If the register value is nonzero, the arpeggiator is on, otherwise it is off. The arpeggiator accepts the 32 bit

GPIO channel containing 4 keycodes as input, and then divides it into separate to check how many keys are being pressed at a time. Because keycodes are always placed in the least significant bits available, knowing the most significant active byte can tell us how many notes are active. Using this information, we create a variable length counter called “tracker” which is incremented on every grain trigger signal. When no or one key is pressed, the tracker remains at zero always, but when two keys are pressed it will count to 1, when three are pressed it will count to 2, and when four are pressed it will count to three before resetting back to zero. Lastly, we assign which keycode byte from the GPIO channel to output in variable “note” based on the value of the tracker. When the tracker is equal to zero the first byte is output, when it's equal to one the second byte is output, and so on. The effect this achieves is that the arpeggiator will cycle through as many notes as are being held down, up to four. If the arpeggiator is off, then only the first byte is output.

Purpose: This module allows us to achieve semi-polyphony, as one can hold down chords and hear the synth automatically cycle through the notes.

clkdiv.sv

Inputs: clk1, reset
Outputs: clk2

Description: This is a simple 2:1 clock divider. On every positive edge of clk1, the output of clk2 is toggled. If reset is pressed, clk2 becomes zero for that clock cycle.

Purpose: This clock divides the system 100Mhz clock down to 50Mhz, which is the fastest speed we use in our design to conform to the specifications of the SD card.

clkdiv3.sv

Inputs: [15:0] read_count, clk, reset
Outputs: read_clk

Description: Technically our second clock divider, but we never changed the name. This module creates the read clock which is used to clock the output of our FIFOs such that samples are played back at the correct rate. Its divider count is controlled by the input read_count, and it is the output of the *octaves.sv* module. When the read counter reaches its max value, the read clock is toggled.

Purpose: This module works together with *note2clk.sv* and *octaves.sv* to play audio back at the correct pitches. The latter two modules create the correct clock divider count, and this module puts that clock into action.

mux2to1.sv

Inputs: a, b, sel
Outputs: y

Description: This is a simple 2:1 mux. When the select is equal to 0, the output y is equal to a, and when select is equal to 1, the output y is equal to b.

Purpose: We used this mux to switch between trigger modes based on the playback mode. The select signal is sampler_mode, a is GO, the output of the impulse train module, and b is read_en_short, the positive edge of the read enable signal sent out by *note2clk.sv*. The effect this achieves is that when sampler mode is on, instead of getting a constant stream of triggers, you just get one every time a key is pressed.

toggle.sv

Inputs: btn, clk, reset
Outputs: signal, reset_out

Description: This module is a slightly modified toggle module. When the input btn is pulsed, the output signal is flipped. In addition, every time the output is toggled, a brief reset pulse is sent out as well in the variable reset_out.

Purpose: This module toggles sampler_mode on and off. The input btn is tied to the positive edge of the sampler_mode button on the FPGA, and the output signal is the sampler_mode signal used by the rest of the system. Additionally, when switching playback modes, the SD FSM must be reset, so we use this reset_out signal to reset our entire system.

MUX.sv

Inputs: [15:0] in, toggle
Outputs: [15:0] out1, [15:0] out2

Description: This acts more like a 1:2 decoder than a mux, so maybe it could've been named something else. The toggle signal is a brief pulse that toggles an internal signal in the module called switch. When switch is 0, out1 is equal to in and out2 is zero, and when switch is 1, out1 is zero and out2 is equal to in.

Purpose: This is kind of the remnant of a feature that never came to fruition. The reason why we have 2 FIFOs instead of one is because we intended for each grain to go into a different FIFO so that grains would overlap one another. Out1 and out2 go to the data in ports of each FIFO. The problem was that for grains to audibly overlap, we needed large FIFO memories, which was not only taxing on memory but also caused the fifo to empty out its contents every time a new key was pressed. This meant that one could change sounds, but the old one would be played until the FIFO was filled with new data. We decided to just shorten the FIFOs to fix this problem and abandon overlapping grains.

PCM2PWM.sv

Inputs: [15:0] data, [4:0] vol, clk
Outputs: PWML, PWMR

Description: This module turns 8-bit PCM data into a stereo analog PWM signal that goes to the headphone drivers. The module first separates the 16-bit stereo sample into the left and right channel bytes. An 8-bit counter continually counts from 0 to 255 on a 50Mhz clock. The value of this counter is compared against the 8-bit mono samples. As long as the counter is less than the sample's value, the PWM signal for that channel is high, but as soon as the counter becomes larger than the sample, the signal is set to 0. What this does it creates a variable duty cycle PWM wave, where smaller values have shorter duty cycles and larger values have longer ones.

Purpose: This module is how we are able to hear our audio data. This module effectively acts as a rudimentary digital to analog converter, and makes our digital representation of samples audible.

inverter.sv

Inputs: a
Outputs: b

Description: b = not a

Purpose: We use this module to turn the full signal of our FIFOs into not full signals that indicate to the SD FSM that there is still room for data to be written. This signal is connected to the ram_op_begun port of the FSM.

<u>IP's and Blocks</u>

clk_wiz_1

Inputs: reset, clk_in1
Outputs: clk_out1, locked

Description: This clock module is used as the main clock for the MicroBlaze to process its data. As such, it is the fastest clock in our design, running at 100MHz. It is connected to the clock input of every IP except mdm_1, which has no clock input. This allows every module to update their data at the same time, ensuring that no timing errors occur.

Purpose: This is the main clock of our design. The MicroBlaze requires a clock because USB devices have a specific polling rate which the CPU needs to keep track of to know when one transaction ends another begins. This module allows the MicroBlaze to do this, and ensures that all data transferred within the system is synchronized such that there are no timing faults in our design.

rst_clk_wiz_1_100M

Inputs: slowest_sync_clk, ext_reset_in, mb_debug_sys_rst, dcm_locked
Outputs mb_reset, [0:0] bus_struct_reset, [0:0] interconnect_aresetn, [0:0] peripheral_aresetn,

Description: An added component to the clock wizard that assists in resetting certain components within the microblaze such as the peripheral and the bus.

Purpose: To reduce the risk of timing issues as well as properly resetting the MicroBlaze when deemed necessary.

Microblaze_0

Inputs : Reset, Clk, Interrupt, [1:0]Interrupt_Ack, Interrupt_Address[31:0], Interrupt, Dbg_Capture, Dbg_Clk, Dbg_Disable, [7:0]Dbg_Reg_En, Debug_Rst, Dbg_Shift, Dbg_TDI ,Dbg_TDO, Dbg_Update

Outputs: [31:0] Data_Addr, D_AS, [3:0] Byte_Enable, DCE, [31:0] Data_Read, Read_Strobe, DReady, DUE, DWait, [31:0] Data_Write, Write_Strobe, [31:0] Instr_Addr, I_AS, ICE, {31:0] Instr, IFetch, IReady, IUE, IWAIT,[31:0] M_AXI_DP_ARADDR, [2:0] M_AXI_DP_ARPROT, M_AXI_DP_ARREADY, M_AXI_DP_ARVALID, [31:0] M_AXI_DP_AWADDR, [2:0] M_AXI_DP_AWPROT, M_AXI_DP_AWREADY, M_AXI_DP_AWVALID, M_AXI_DP_BREADY, [1:0] M_AXI_DP_BRESP, M_AXI_DP_BVALID, [31:0] M_AXI_DP_RDATA, M_AXI_DP_RREADY, [1:0] M_AXI_DP_RRESP, M_AXI_DP_RVALID, [31:0] M_AXI_DO_WDATA, M_AXI_DP_WREADY, [3:0]M_AXI_DP_WSTRB, M_AXI_DP_WVALID

Description : The MicroBlaze is the brain of our design. It is connected to the memory and AXI bus, and controls the reading and writing of data between the two. It generates the

signals which drive data transfer handshakes, such as the “valid” and “ready” signals which precede the reading and writing of data. It is programmed with C code from Vitis which instructs it on how and when to generate these signals, and through it all other functions of our design follow.

Purpose: The MicroBlaze serves as the CPU of our design, and thus is vital. It allows for the connection of the USB host, memory, and other Vivado modules so that they can work together to accomplish their tasks. The other modules talk to each other through it, and without it nothing would work.

microblaze_0_axi_intc

Inputs: [8:0] s_axi_araddr, s_axi_arready, s_axi_arvalid, [8:0] s_axi_awaddr, s_axi_awready, s_axi_awvalid, s_axinready, [1:0] s_axi_bresp, s_axi_bvalid, [31:0] s_axi_rdata, s_axi_rready, [1:0] s_axi_rresp, s_axi_rvalid, [31:0]s_axi_wdata, s_axi_wready, s_axi_wstrb[3:0], s_axi_wvalid, s_axi_aclk, s_axi_aresetm, [3:0] intr, processor_clk, processor_rst

Outputs: [1:0]processor_ack, [31:0] interrupt_address, irq

Description: This is the main interrupt controller in our design. It takes the concatenated interrupt signals from other interrupt-generated modules and synchronizes it to the clock before sending it to the MicroBlaze interrupt input. It is clocked by the master clock, and is also connected to the master reset.

Purpose: This completed the transfer of interrupts from modules that produce them to the MicroBlaze. Without this module, interrupts would be unsynchronized, and might not properly register in the MicroBlaze as signals would enter asynchronously.

mdm_1

Inputs:

Outputs: Dbg_Capture_0, Dbg_Clk_0, Dbg_Disable_0, [7:0] Dbg_Reg_En_0, Dbg_Rst_0, Dbg_Shift_0, Dbg_TDI_0, Dbg_TDO_0, Dbg_Update_0, Debug_SYS_Rst

Description: This is the debug module attached to the MicroBlaze. It has only two connections, the DEBUG bundle of signals which goes into the MicroBlaze debug port, and the Debug_SYS_Rst signal, which is the master reset signal of our design, and enters into the reset port of the master clock to become synchronized before being sent to every other module which requires a reset signal.

Purpose: The purpose of this module is to allow for debugging MicroBlaze code. It sends out the reset signal in the case the program needs to restart, and also can detect breakpoints in

MicroBlaze code and send an interrupt through the DEBUG channel. This is essential for checking MicroBlaze code.

microblaze_0_local_memory

Inputs:[31:0] DLMB_abus, DLMB_addrstrobe, [3:0]DLMB_be, DLMB_ce, [31:0] DLMB_readdbus, DLMB_readtstrobe, DLMB_ready, DLMB_rst, DLMB_ue, DLMB_wait, [31:0] DLMB_writedbus, DLMB_writestrobe, [31:0] ILMB_abus, ILMB_addrstrobe, [3:0] ILMB_be, ILMB_ce, [31:0] ILMB_readdbus, ILMB_readstrobe, ILMB_ready, ILMB_rst, ILMB_ue, ILMB_wait, [31:0] ILMB_writedbus, ILMB_writestrobe, LMB_Clk, SYS_Rst

Outputs:

Description: This is the local memory of our MicroBlaze. It is a small, but very readily accessible form of memory directly connected to the processor. It is most useful for storing frequently accessed and critical data. In our case, it stores the C instructions which program the MicroBlaze.

Purpose: The local memory of the MicroBlaze stores the code which controls its behavior, and is essential for its operation.

microblaze_0_axi_periph

Inputs: S00_AXI_awaddr, S00_AXI_awprot, S00_AXI_awvalid, S00_AXI_awready, S00_AXI_wdata, S00_AXI_wstrb, S00_AXI_wvalid, S00_AXI_wready, S00_AXI_bresp, S00_AXI_bvalid, S00_AXI_bready, S00_AXI_araddr, S00_AXI_arprot, S00_AXI_arready, S00_rdata,S00_AXI_rresp, S00_AXI_rvalid, S00_AXI_arvalid, S00_AXI_rready, ACLK, ARESETN, S00_ACLK, S00_ARESETN, M00_ACLK, M00_ARESETN, M01_ACLK, M01_ARESETN, M02_ACLK, M02_ARESETN, M03_ACLK, M03_ARESETN, M04_ACLK, M04_ARESETN, M05_ACLK, M05_ARESETN, M06_ACLK, M06_ARESETN,

Outputs: M00_AXI_awaddr, M00_AXI_awprot, M00_AXI_awvalid, M00_AXI_awready, M00_AXI_wdata, M00_AXI_wstrb, M00_AXI_wvalid, M00_AXI_wready, M00_AXI_bresp, M00_AXI_bvalid, M00_AXI_bready, M00_AXI_araddr, M00_AXI_arprot,M00_AXI_arvalid, M00_AXI_arready, M00_rdata,M00_AXI_rresp, M00_AXI_rvalid, M00_AXI_rready (Note: Signals for M01-M06 are identical to M00)

Description: This module implements the MicroBlaze's AXI bus peripheral, and is how the MicroBlaze interfaces with the bus. Input to the interface are all the signals, mainly read and write signals, that the MicroBlaze sends to its slave devices. All of these signals are placed onto the AXI bus to be received by the devices the MicroBlaze controls. At the same time, it transfers outputs sent from those devices back to the MicroBlaze, and routes them to the appropriate module in the block design.

Purpose: The AXI bus is the glue which holds our design together, and the roads which allow for data transfer. The bus is necessary to route signals to their appropriate destination and to allow modules within the block design to communicate with those outside it.

axi_uartlite_0

Inputs: [3:0] s_axi_araddr, s_axi_arready, s_axi_arvalid, [3:0] s_axi_awaddr, s_axi_awready, s_axi_awvalid, s_axi_bready,[1:0] s_axi_bresp, s_axi_bvalid, [31:0] s_axi_rdata, s_axi_rready, [1:0] s_axi_rresp, s_axi_rvalid, [31:0] s_axi_wdata, s_axi_wready, [3:0]s_axi_wstrb, s_axi_wvalid, s_axiaclk, s_axi_aresetn,

Outputs: rx, tx, interrupt

Description: The UART (Universal Asynchronous Receiver-Transmitter) is a device which allows multiple signals to be sent in serial format. It transforms its parallel input data into a serial stream, and converts the serial streams it receives back into parallel for the recipient devices to use. The data is transmitted asynchronously; rather than using a clock, the two devices have a shared baud rate, which determines the rate at which bits are transferred between devices. This allows data to remain synchronized even without a clock.

Purpose: In our design, the UART is used to generate print statements and send them to the serial console in Vitis, which allows us to add checks to our code and debug it within the serial console.

Xlconcat_0

Inputs : ln0[0:0],ln1[0:0],ln2[0:0],ln3[0:0]

Outputs: dout[3:0]

Description : Concatenates 4 one bit input signals into a singular 4 bit output signal.

Purpose: This is used specifically to be connected to the interrupt signal of the microblaze_0_axi_intc block. It takes interrupt signals from the UART, AXI Peripheral, Usb Timer, and USB Interrupt blocks.

timer_usb_AXI

Inputs: [4:0] s_axi_araddr, s_axi_arready, s_axi_arvalid, [4:0] s_axi_awaddr, s_axi_awready, s_axi_awvalid, s_axi_bready, [1:0] s_axi_bresp, s_axi_bvalid, [31:0] s_axi_rdata, s_axi_rready, [1:0] s_axi_rresp, s_axi_rvalid, s_axi_wready, [31:0] s_axi_wdata, s_axi_wready, [3:0] s_axi_wstrb, s_axi_wvalid, capturetrig0, capturetrig1, freeze, s_axi_aclk, s_axi_aresetn

Outputs: generateout0, generateout1, pwn0, interrupt

Description: This module is used to monitor the USB host device and check when a new transaction is beginning. Based on the contents of the 6th channel of the AXI bus, it generates an interrupt which is sent to the interrupt concatenator to temporarily halt the system.

Purpose: The purpose of this module is to allow the USB device to create an interrupt so that data can be properly read into the MicroBlaze.

microblaze_0_axi_intc

Inputs: [8:0] s_axi_araddr, s_axi_arready, s_axi_arvalid, [8:0] s_axi_awaddr, s_axi_awready, s_axi_awvalid, s_axinready, [1:0] s_axi_bresp, s_axi_bvalid, [31:0] s_axi_rdata, s_axi_rready, [1:0] s_axi_rresp, s_axi_rvalid, [31:0]s_axi_wdata, s_axi_wready, s_axi_wstrb[3:0], s_axi_wvalid, s_axi_aclk, s_axi_aresetm, [3:0] intr, processor_clk, processor_rst

Outputs: [1:0]processor_ack, [31:0] interrupt_address, irq

Description: This is the main interrupt controller in our design. It takes the concatenated interrupt signals from other interrupt-generated modules and synchronizes it to the clock before sending it to the MicroBlaze interrupt input. It is clocked by the master clock, and is also connected to the master reset.

Purpose: This completed the transfer of interrupts from modules that produce them to the MicroBlaze. Without this module, interrupts would be unsynchronized, and might not properly register in the MicroBlaze as signals would enter asynchronously.

spi_usb

Inputs: [6:0] s_axi_araddr, s_axi_arready, s_axi_arvalid, [6:0]s_axi_awaddr, s_axi_awready, s_axi_awvalid, s_axi_bready,[1:0] s_axi_bresp, s_axi_bvalid, [31:0] s_axi_rdata, s_axi_rready, [1:0] s_axi_rresp, s_axi_rvalid, [31:0] s_axi_wdata, s_axi_wready, [3:0] s_axi_wstrb, s_axi_wvalid, ext_spi_clk, s_axi_aclk, s_axi_aresetn

Outputs: io0_i, io0_o, io0_t, io1_i, io1_o, io1_t, sck_i, sck_0, sck_t,[0:0] ss_i, [0:0]ss_o, ss_t, ip2intc_irpt.

Description: This module facilitates SPI communication between the MicroBlaze and the USB host. Its inputs are the master clock and reset, as well as the AXI bus in order to receive information from the USB host. It outputs MOSI (master-out slave-in) data, a slave clock, and a 1-bit slave select (since we only have one USB device we can select).

Purpose: The purpose of this device is to set up serial data transfer from the USB device to the block design, and is essentially how the USB device “talks” to the MicroBlaze.

gpio_usb_int

Inputs: [8:0] s_axi_araddr, s_axi_arready, s_axi_arvalid, [8:0] s_axi_awaddr, s_axi_awready, s_axi_awvalid, s_axi_bready, [1:0] s_axi_bresp, s_axi_bvalid, [31:0] s_axi_rdata, s_axi_rready, [1:0] s_axi_rresp, s_axi_rvalid, s_axi_wready, [31:0] s_axi_wdata, s_axi_wready, [3:0] s_axi_wstrb, s_axi_wvalid,s_axiaclk, s_axi_aresetn,

Outputs: [0:0]gpio_io_i, ip2intc_irpt

Description: The interrupt signal generated by the USB device, transmitted to the interrupt concatenator.

Purpose: Allows the USB to pause data flow.

gpio_usb_rst

Inputs: [8:0] s_axi_araddr, s_axi_arready, s_axi_arvalid, [8:0] s_axi_awaddr, s_axi_awready, s_axi_awvalid, s_axi_bready, [1:0] s_axi_bresp, s_axi_bvalid, [31:0] s_axi_rdata, s_axi_rready, [1:0] s_axi_rresp, s_axi_rvalid, s_axi_wready, [31:0] s_axi_wdata, s_axi_wready, [3:0] s_axi_wstrb, s_axi_wvalid,s_axiaclk, s_axi_aresetn,

Outputs: [0:0] gpio_io_o

Description: A reset for the usb gpio.

Purpose: It is used to initialize the usb connected device, as well as correct possible errors that may occur.

gpio_usb_keycode

Inputs: Inputs: [8:0] s_axi_araddr, s_axi_arready, s_axi_arvalid, [8:0] s_axi_awaddr, s_axi_awready, s_axi_awvalid, s_axi_bready, [1:0] s_axi_bresp, s_axi_bvalid, [31:0] s_axi_rdata, s_axi_rready, [1:0] s_axi_rresp, s_axi_rvalid, s_axi_wready, [31:0] s_axi_wdata, s_axi_wready, [3:0] s_axi_wstrb, s_axi_wvalid,s_axiaclk, s_axi_aresetn,

Outputs: [31:0] gpio_io_o, [31:0] gpio2_io_o

Description: This block is created to read the USB keycodes that are being produced via user input. Individual keycodes are stored as 8 bit hexadecimal values. In the case that multiple keys are pressed, the gpio_io_o will be more than 8 bits long, and thus can extend to 32 bits.

Purpose: Used to read and write the keyboard inputs to be implemented as the keyboard sound for the granular synth. Each key corresponds to a certain pitch.

axi_gpio_0

Inputs: Inputs: [8:0] s_axi_araddr, s_axi_arready, s_axi_arvalid, [8:0] s_axi_awaddr, s_axi_awready, s_axi_awvalid, s_axi_bready, [1:0] s_axi_bresp, s_axi_bvalid, [31:0] s_axi_rdata, s_axi_rready, [1:0] s_axi_rresp, s_axi_rvalid, s_axi_wready, [31:0] s_axi_wdata, s_axi_wready, [3:0] s_axi_wstrb, s_axi_wvalid,s_axiaclk, s_axi_aresetn,

Outputs: [31:0] gpio_io_o

Description: Used for the input of certain registers

Purpose: To create an input for each of the appropriate registers (specifically R1,R2,R3, and R4) so that they can be randomized through the MicroBlaze. R2 and R4 represent randomization values and R1 and R3 represent position and density.

axi_gpio_1

Inputs: Inputs: [8:0] s_axi_araddr, s_axi_arready, s_axi_arvalid, [8:0] s_axi_awaddr, s_axi_awready, s_axi_awvalid, s_axi_bready, [1:0] s_axi_bresp, s_axi_bvalid, [31:0] s_axi_rdata, s_axi_rready, [1:0] s_axi_rresp, s_axi_rvalid, s_axi_wready, [31:0] s_axi_wdata, s_axi_wready, [3:0] s_axi_wstrb, s_axi_wvalid,s_axiaclk, s_axi_aresetn,

Outputs: [31:0] gpio_io_o

Description: Used for the output of registers that will be put in the change the way the SD card reads audio.

Purpose: To output the modified position, density, length, and flip probability to be read in the SD card state machine.

axi_gpio_2

Inputs: Inputs: [8:0] s_axi_araddr, s_axi_arready, s_axi_arvalid, [8:0] s_axi_awaddr, s_axi_awready, s_axi_awvalid, s_axi_bready, [1:0] s_axi_bresp, s_axi_bvalid, [31:0] s_axi_rdata, s_axi_rready, [1:0] s_axi_rresp, s_axi_rvalid, s_axi_wready, [31:0] s_axi_wdata, s_axi_wready, [3:0] s_axi_wstrb, s_axi_wvalid,s_axiaclk, s_axi_aresetn,

Outputs: [23:0] gpio_io_o

Description: Used for the input of registers.

Purpose: To create an input for each of the appropriate registers (specifically R5,R6, and R8) so that they can be randomized through the MicroBlaze. R5 represents length and R6 length randomization and R8 is flip.

Design Resources and Statistics

LUT	5100
DSP	7
Memory (BRAM)	24
Flip-Flop	5109
Frequency	114.34 MHz
Static Power	.076W
Dynamic Power	.188W
Total Power	.264W

Surprisingly, the total power utilized by this project was not as high as other labs, as well as the frequency being lower than some labs. To no surprise however, this project required a large amount of LUT, DSP, BRAM, and FFs compared to the other labs that were done.

Conclusion:

Overall, although this project required a lot of time and effort, the end product was successful and it made the work worthwhile. We were able to successfully implement all of the settings and even create the additional setting of an arpeggiator and volume control, as well as including a sample mode. Working on this project taught us how important communication was when distributing the workload and discussing which parts were completed so that further implementation could be added to the project, as well as teaching us how to use debugging tools such as the ila to monitor signals and diagnose problems if they came up.

If we were to continue adding functionality to this project, one thing that we could do is add an HDMI display component to the FPGA, so that maybe the settings could be configured through the display. Another thing we may consider is also adding volume control via the potentiometer, so that a fade effect can be added to the grains.