Keytar Hero

Dartmouth College | Engs 31 | Final Project 2023

Michael D

Abdibaset B

Edwin O

Erin B

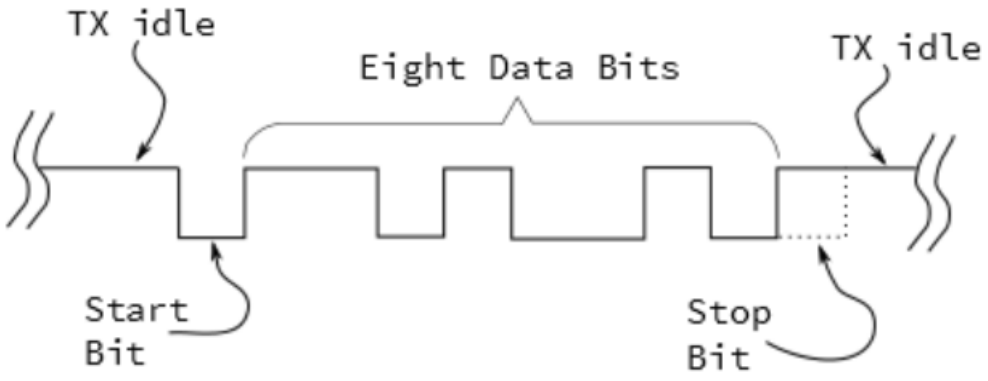
June 5, 2023

Table of Contents

I Background

1.1 MIDI Interface

The MIDI device was connected to our FGPA through a PMod adaptor. The PMod Adaptor steps the voltage down to 3.3 V and is read through Pin 1 [Hansen, 2023].[1] The MIDI uses the Universal Asynchronous Receiver Transmitter (UART) protocol in which 3 bytes of 10 bits are sent serially asynchronous with a baud rate of 31,250 [MIDI Tutorial]. The transmission line is high when in the idle state and drops low for a start bit. Eight bits are transferred before a high stop bit is sent.



UART Communicaiton. MIDI uses UART protocol in which idle states are high, the start bit is low and the stop bit is high. In total, there are 10 bits transferred [Sparkfun, 2023].[2]

In the first byte, if the first bit is a 1, the message is a status bit. The eight status signals are tabulated in the figure below. We use Note On (1001) and Note Off (1000) status signals to determine which note is being pressed. For the note on and off command, the following bytes contain the note identification number and the key velocity [Hansen, 2023]. The key velocity is determined by measuring the delay in two switches positioned on different ends of each note [Sparkfun, 2023]. It is important to note that some devices will skip the Note Off a status bit, sending a velocity of zero to indicate Note Off, a technique termed "Implicit Off" [Sparkfun, 2023]. We use any instance of the Pitch Bend (1110) to drive the player into Overdrive [Sparkfun, 2023]. In original Rock Band and Guitar Hero Games, when the user maintains a

[1] *Prof Hansen's Guide to Midi: Digital Electronics (SP23)*. https://dartmouth.instructure.com/courses/58308/pages/prof-hansens-guide-to-midi. Accessed 17 May 2023.
[2] *MIDI Tutorial - SparkFun Learn*. https://learn.sparkfun.com/tutorials/midi-tutorial/all. Accessed 17 May 2023.

streak of hits, they can hit a pitch wheel, activating overdrive and doubling their points for a short period of time.

MIDI Status Messages					
Message Type	**MS Nybble**	**LS Nybble**	**Number of Data Bytes**	**Data Byte 1**	**Data Byte 2**
Note Off	0x8	Channel	2	Note Number	Velocity
Note On	0x9	Channel	2	Note Number	Velocity
Polyphonic Pressure	0xA	Channel	2	Note Number	Pressure
Control Change	0xB	Channel	2	Controller Number	Value
Program Change	0xC	Channel	1	Program Number	-none-
Channel Pressure	0xD	Channel	1	Pressure	-none-
Pitch Bend	0xE	Channel	2	Bend LSB (7-bits)	Bend MSB (7-bits)
System	0xF	further specification	variable	variable	variable

Bit commands from the MIDI device. This chart defines “nybbles” as 4 bits of the eight-bit signal. In our project, we will use Note On (1001), Note Off (1000), and Pitch Bend (1110) [SparkFun, 2023].

One group member owns an original 1988 Yamaha SHS 200. From the device manual, Key Velocity is set to a constant 64 and the pitch bender has 7-bit resolution [ManualsLib, 2023]. Therefore, in this project, note velocity and pitch bender value will be ignored: our finite state machine will determine if notes or the pitch wheel was pressed but ignore velocity and pitch bend degree.

1988 Yamaha SHS 200. The Yamaha SHS 200 keytar includes a MIDI out.

MIDI Implementation Chart **Version: 1.0**

Function		Transmitted	
		Manual performance	Auto rhythm
Basic	Default	1	16
Channel	Changed	1-16	×
Mode	Default	Mode3	Mode3
	Messages	×	×
	Altered	* * * * * * *	* * * * * * * * * *
Note		23-96 *1	*3
Number	: True voice	* * * * * * *	* * * * * * * * * *
Velocity	Note ON	× 9nH v =64	○ 9FH v=56, 64, 127
	Note OFF	× 9nH v=0	× 9FH v=0
After	Key's	×	×
Touch	Ch's	×	×
Pitch Bender		○ 7 bit resolution	×

1988 Yamaha SHS 200 User's Manual MIDI Components. The Yamaha SHS 200 keytar includes a MIDI out with 16 channels. In manual performance mode, the note velocity is a constant. Additionally, the pitch bend is a 7-bit resolution instead of a maximum 14-bit resolution. Both velocity and pitch bend resolution will be ignored.

With regards to key numbers, Middle C is 60 while C3 is 48 and G3 is 55 [Inspired Acoustics, 2023]. We choose to split the right and left hands on note 59.

MIDI Note Code (Key Number to Note) Lookup Table		
Note	**Key Number**	**Rule**
A	21, 33, 45, 57, 69, 81, 93, 105, 117	(21+n*12)
B	23, 35, 47, 59, 71, 83, 95, 107, 119	(23+n*12)
C	24, 36, 48, 60, 72, 84, 96, 108, 120	(24+n*12)
D	26, 38, 50, 62, 74, 86, 98, 110, 122	(26+n*12)

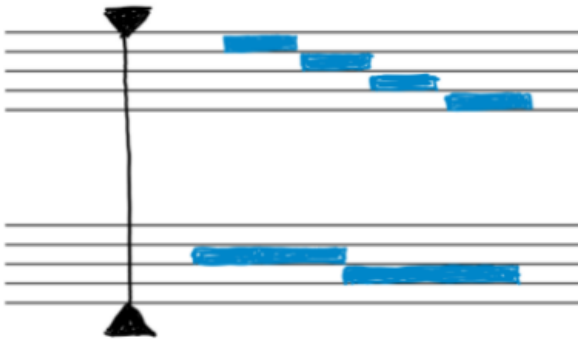
E	28, 40, 52, 64, 76, 88, 100, 112, 124	(28+n*12)
F	29, 41, 53, 65, 77, 89, 101, 113, 125	(29+n*12)
G	31, 43, 55, 67, 79, 91, 103, 115, 127	(31+n*12)

We will represent "ACTIVE" notes with a 5-bit number ($11111_2 = 31_{10}$) to represent 24 possible note locations on the staff (Figure #). On the right-hand (top staff) notes will go from C4 (60) to G5 (79). Notes from key 4 (60) to key 9 (127) will be accepted. For example, if a C is shown, notes 60, 72, 84, 96, 108, or 120 will be accepted. On the left-hand (bottom staff), notes will go from F3 (41) to B3 (59). Notes from key 0 (21) to key 3 (59) will be excepted. For example, is a C is shown, notes 24, 36, or 48 will be accepted. A twenty-four-bit number, NoteOn, will store which note— C through B on the left hand or C through B on the right hand— is pressed. Treble and bass clef mappings are shown in Figure #.

1.2 Visual Design

1.2a Static Staff Design

The figure below contains concept art of the VGA design hand-drawn in Microsoft Journal. Blue bars will sweep across the screen. A static base and treble clef along with target region graphics will be present.



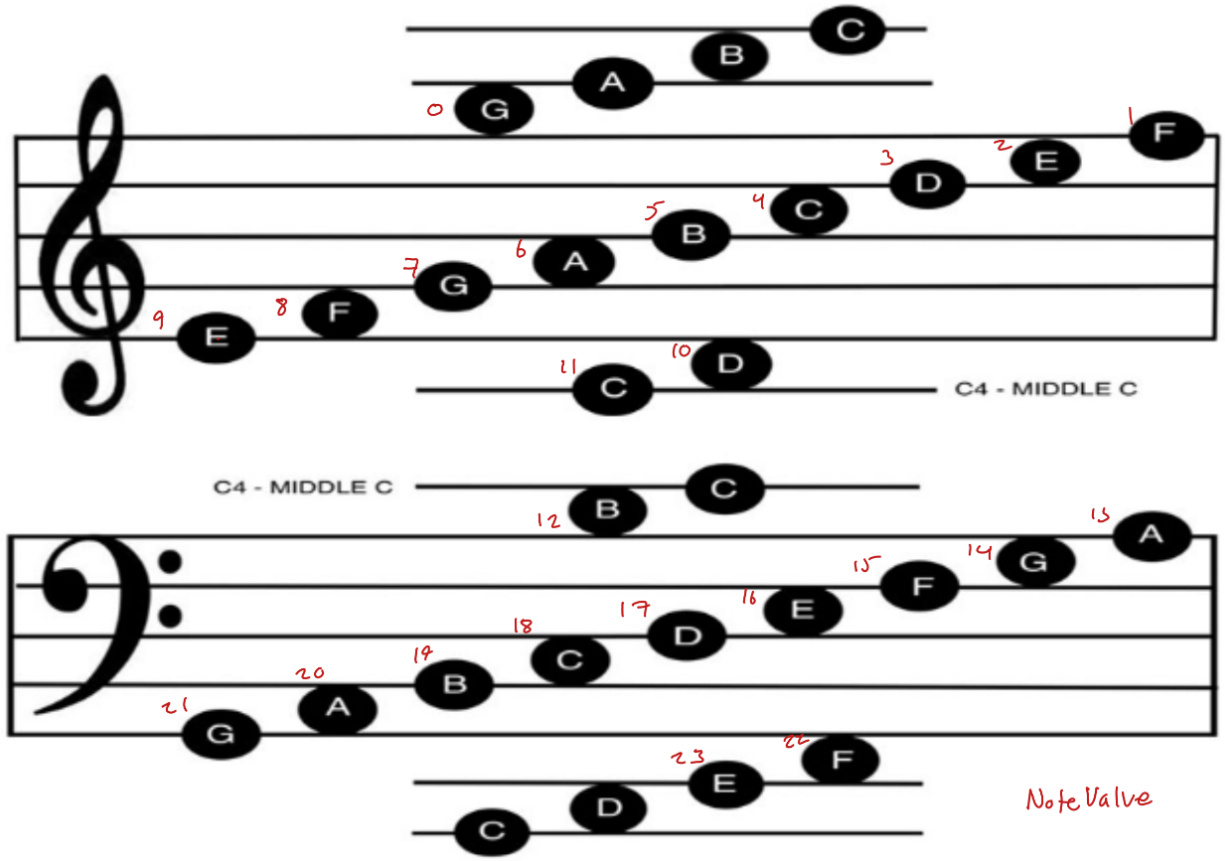
VGA Concept Art. The potential note positions are labeled in red (these labels will NOT be present on the VGA). Blue rectangles, each representing a quarter note will slide across a screen to a target.

The staff pattern was generated in MATLAB using the imshow() function. This pattern was translated into VHDL logical statements. The VGA_Test_Pattern Code was modified to project the staff pattern.

Staff Pattern. The staff pattern logical equations were tested in MATLAB.

1.2b TargetNote to NoteDisplay Representation and Mapping

The staff will contain blue bars corresponding to 24 notes: E through B on the bass clef (left hand) and C through G on the treble clef (right hand). Our NoteOn and TargetNote variables consist of 24-bit vectors representing if a note is on or off. The designed note will be compared to the actual note playing. A match will be made if the correct letter is not being played for each hand. For example, any C note in the right hand (all C notes above 60) will provide a match if either note 12 or 19 is received. This will allow players to choose which octave and pitch to play the song at. Additionally, players can add notes and embellishments as they are only penalized for misses.



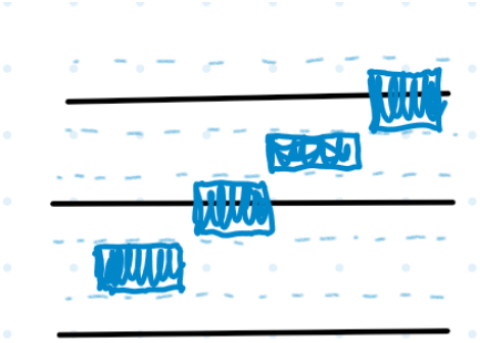
Treble and Base Clef Mappings. Clef position to notes. Red writing represents the indices in our note variables sent between the VGA and the MIDI comparator circuit.

Since one note goes between the lines of the staff and others go on the staff lines (one on the line above, one on the line below) the staff line spacing must be a multiple of four. In order to have seven staff locations (five lines) with spacing, we need 24 locations. If we take the height of the screen (480) and divide by 24, we get that bars are spaced out by twenty and notes will be 10 pixels tall. However, to ease the reading of the register file for the VGA, I load column-wise such that the VGA team can read row-wise; that is each row in the register file is 640 bits and represents a row in the VGA screen. The mapping was performed in Excel.

480-Pixel-to-24-Note Mapping							
TargetNote	Hand	Letter	$[y_1, y_2]$	Start Column	End Column	Start Index	End Index
0	Left	E	[55,65]	55	65	35199	41599

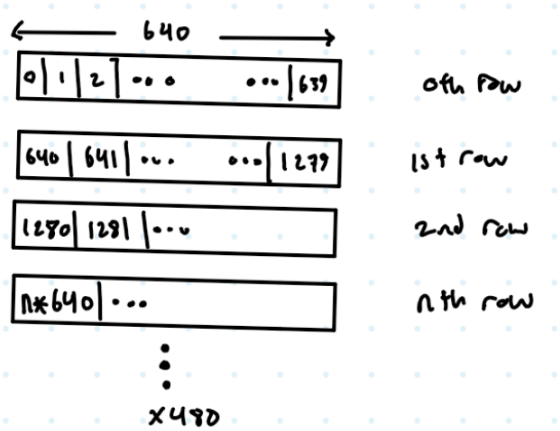
1	Left	F	[65,75]	65	75	41599	47999
2	Left	G	[75,85]	75	85	47999	54399
3	Left	A	[85,95]	85	95	54399	60799
4	Left	B	[95,105]	95	105	60799	67199
5	Left	C	[105,115]	105	115	67199	73599
6	Left	D	[115,125]	115	125	73599	79999
7	Left	E	[125,135]	125	135	79999	86399
8	Left	F	[135,145]	135	145	86399	92799
9	Left	G	[145,155]	145	155	92799	99199
10	Left	A	[155,165]	155	165	99199	105599
11	Left	B	[165,175]	165	175	105599	111999
12	Right	C	[275,285]	275	285	175999	182399
13	Right	D	[285, 295]	285	295	182399	188799
14	Right	E	[295,305]	295	305	188799	195199
15	Right	F	[305, 315]	305	315	195199	201599
16	Right	G	[315, 325]	315	325	201599	207999
17	Right	A	[325,335]	325	335	207999	214399

18	Right	B	[335, 345]	335	345	214399	220799
19	Right	C	[345, 355]	345	355	220799	227199
20	Right	D	[355, 365]	355	365	227199	233599
21	Right	E	[365, 375]	365	375	233599	239999
22	Right	F	[375, 380]	375	385	239999	246399
23	Right	G	[395, 405]	385	395	246399	252799

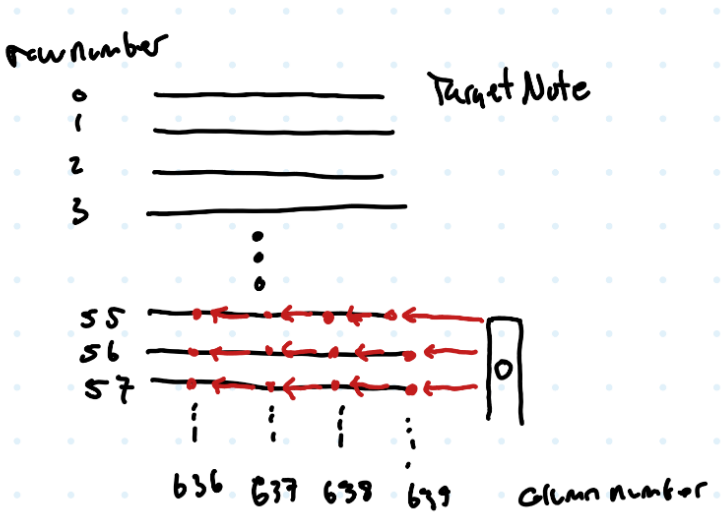


Note Spacing Sketch. Not drawn to scale. Notes are equal heights and either occupy positions on the staff bar or in the middle. Note locations do not overlap.

Note Spacing Simulation. Dary grey bars represent notes on the staff (left) and between staff lines (right).



NoteDisplay Register File. The NoteDisplay register file will consist of 480 rows. Each row will have 640 bits. This will make sweeping rows easy for the VGA display circuit but difficult for the TargetNote loading circuit.



NoteDisplay Register File Loading. The target note will be mapped to rows of the NoteDisplay register file. A shift circuit will increment each index in the NoteDisplay. The first column will become whatever is in TargetNote is loadTarget is high. The last value will be rewritten and lost. Notes will move right to left.

II System Design

2.1 System Design Summary

The COE file has T rows of 24-bit vectors. Each bit represents a note, 12 on the left hand, and 12 on the right hand. A 640 by 480 register file will be binned into 24 locations. For t clock cycles, the Tth row of the COE file will be read into corresponding bins in the first column of the 640 by 480 register file The 640 by 480 register file will shift left such that the second column becomes the first, and the first becomes the mapping of the row of the COE file. This will repeat, moving, 1's and 0's across the 640 by 480 register file which can be displayed on the screen. We will have comparator logic for the target region of the 640 by 480 register file to determine which note should be pressed. This will become the TargetNote variable in the Key Comparator. The Key Comparator circuit will determine the letter of the note the user is pressing and which hand is pressing the note. With regard to left and right hands, the keyboard will be split in middle C. This will allow users to choose which octave they play in. However, two transformations must be performed: the MIDI output transformed to the simplified note letter vector, and also the target note simplified to the note letter. The Key comparator will also determine if the pitch wheel is hit. HIT and OverDrive will quickly be sent back to the VGA circuit such that the display can be updated and give the user feedback based on the color of the note bars.

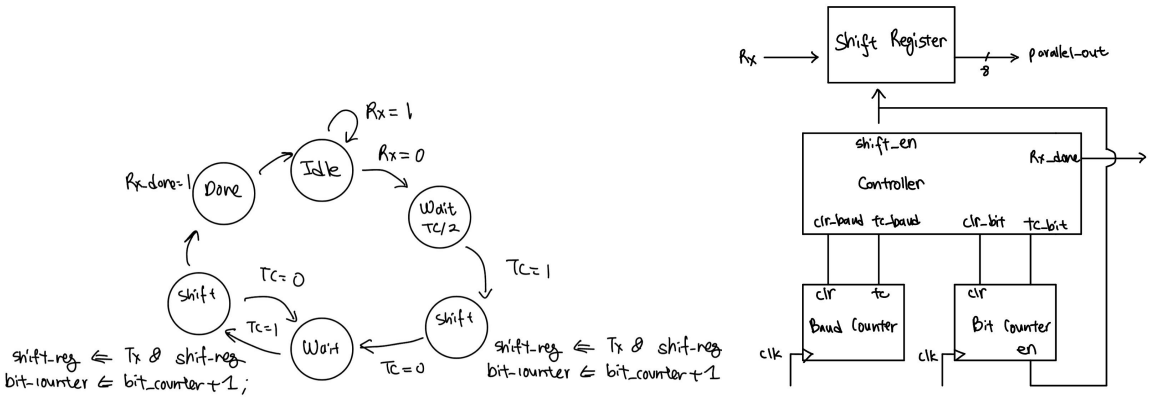
A shell file connects the three main aspects of our system: (1) MIDI In, (2) VGA Display, and (3) the Key Compare Interface between the MIDI and VGA. The shell contains a system clock, the vertical and horizontal sync, and the colors of the VGA display. Additionally, we have

included debugging signals and a constraint file such that we can test our system using the oscilloscope. In the report below, each circuit (represented by a file in our Vivado project), will be discussed. Inputs/outputs, important calculations, and tests to debug the system will be discussed.

2.2 MIDI Reciever

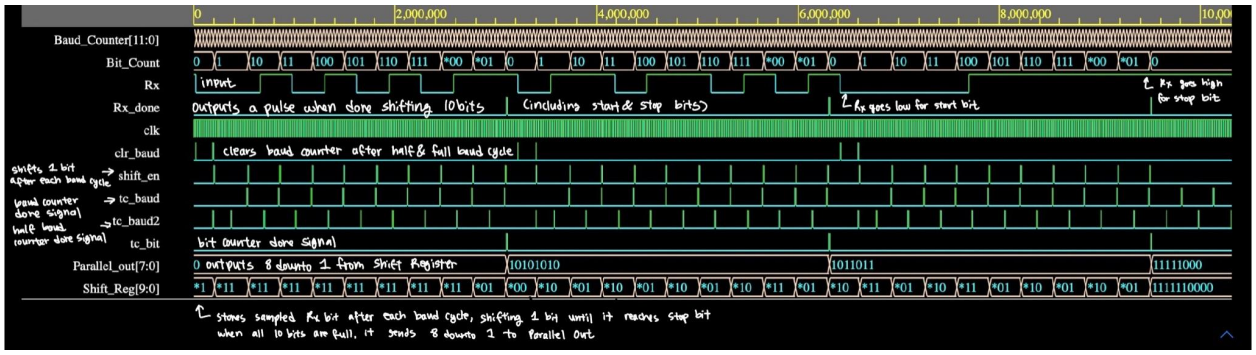
2.2a SCI Receiver

We will use two finite state machines, one to control a SCI shift register receiver (Figure #) and the other to drive the logic of reading bytes from the MIDI (Figure #). The SCI shift register discussed in Lecture 20 will be deployed. The FSM will be in the IDLE state until Rx goes low. In the WaitTC/2 state, the FSM will wait half a clock cycle to center on values and clear baud count. After resetting the baud rate, the state machine will move on to a Shift state, shifting in the start bit into the Shift Register. Then, transferring to a Wait state, the FSM will wait for the a full baud cycle. Then, in the Shift state, the FSM will read in Rx into the shift register, shift one bit, then wait again for a baud cycle. After 10 bits including start and stop bit, Rx_done will be set high. The datapath will use a baud counter to control the shifting at the baud speed and a bit counter to keep track of how many bits have been read. The counters will each have a terminal count and clear signal. The controller will have a shift_en and Rx_done signal. The shift register will output an 8 bit Parallel_Out signal, cutting out the start and stop bit from the Shift Register.

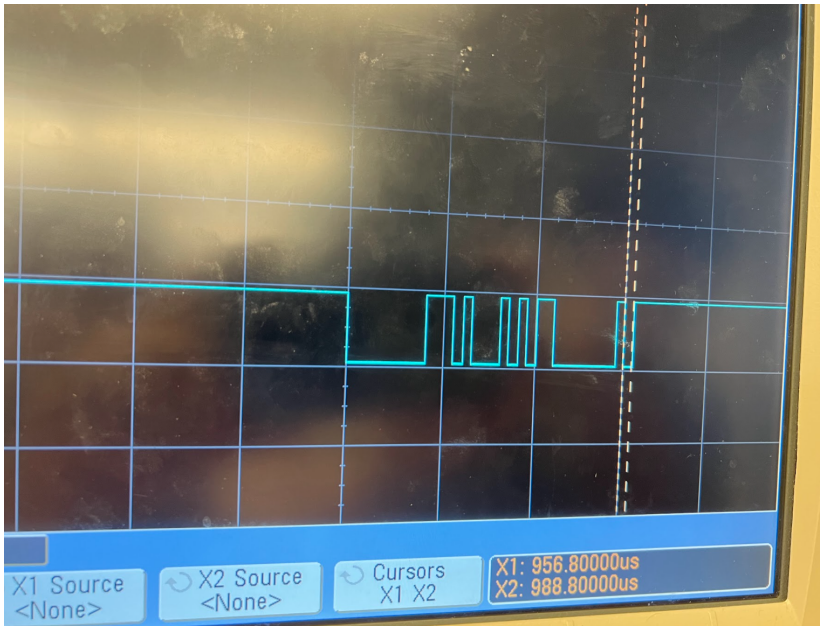


Lecture 20 Shift Register Receiver. A finite state machine, once Rx goes low, will loop through waiting and shifting states before noting that data is ready. A datapath with a baud and bit counter will interface with a controller which will drive the shift register.

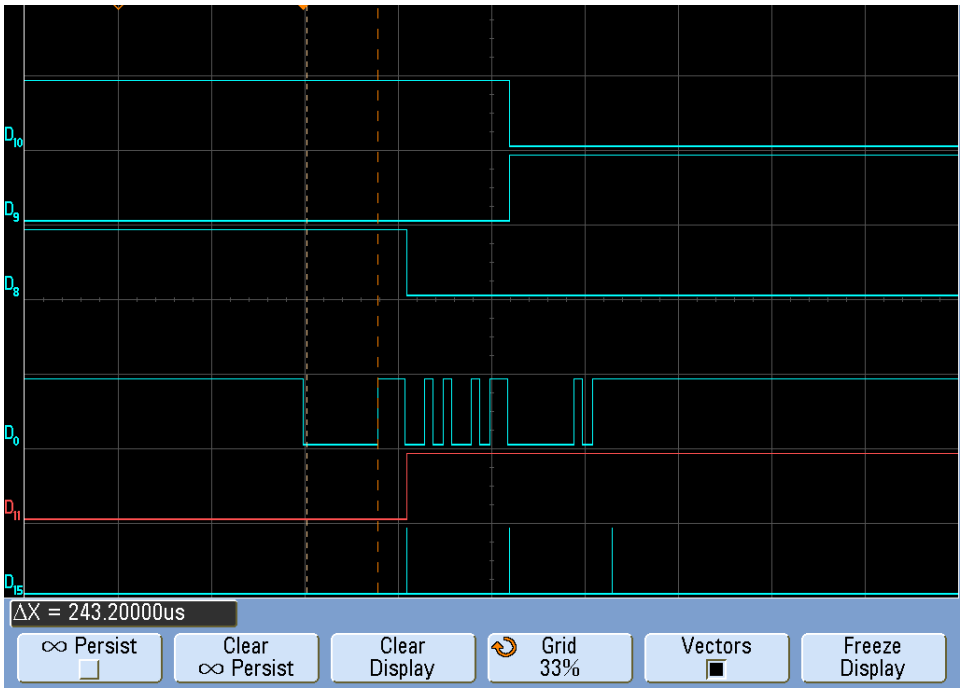
SCI Receiver Signals		
Signal	**Type**	**Notes**
Rx	std_logic	Input data
Baud_Counter	std_logic_vector(11 downto 0)	Counter for baud rate
clr_baud	std_logic	Clears Baud_Counter
tc_baud	std_logic	Full baud count done
tc_baud2	std_logic	Half baud count done
Bit_Count	Integer range 0 to 10	Counter for bit shifting
tc_bit	std_logic	Bit shifting (10) done
Shift_Register	std_logic_vector(9 downto 0)	Stores Rx data
Parallel_Out	std_logic_vector(7 downto 0)	Output Rx data
Rx_done	std_logic	Output done signal



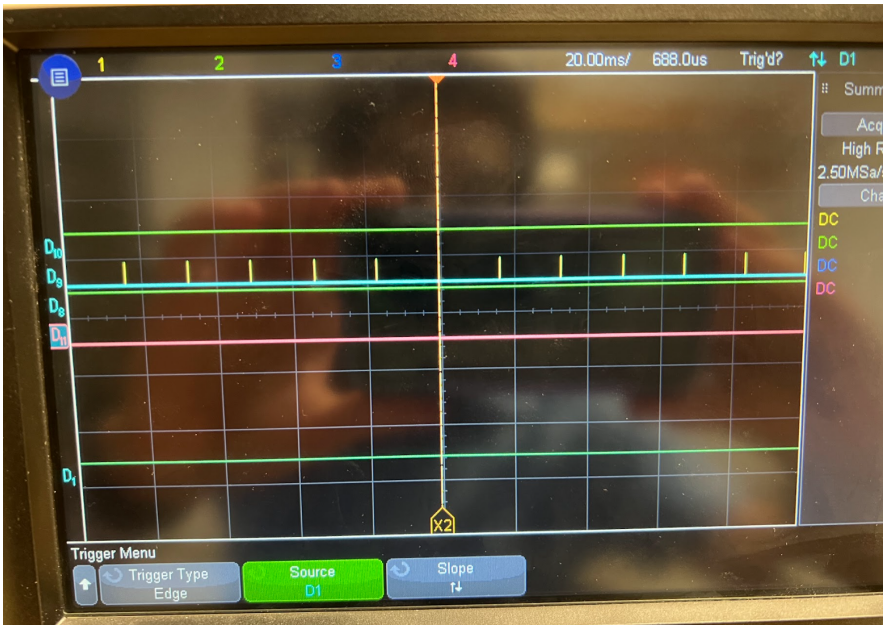
Simulation of SCI Receiver. Takes in 10 bit Rx signals starting with a high start bit and ending with a low stop bit. SCI Receiver outputs Rx_done and Parallel_out signals when it is done shifting a byte. Rx_done is a pulse signaling the end of a byte, and Parallel_out is an 8 bit signal.



Oscilloscope Recording of Rx. The Rx pin on the MIDI Pin Brick was scoped to determine that the input was as documented. Our first keyboard had a bug in which note values were not as tabulated above and online. Our second keyboard sent the correct Rx.



Oscilloscope Recording of Rx and the bits 0, 1, 2, 3 of Parallel Out and Rx_done: This recording shows the 3 byte Rx input signal (D0), 3 Rx_done signals indicating the end of each byte (D15), and the different signals of each bit in the Parallel Out output signal (D8, D9, D10, D11).



Rx Done Scope Testing. Oscilloscope Reading of Rx Done (D9) and debugging variables for the FSM.

MIDI Input Debugging. Rx Done on D10. MIDI's Rx on D11. Above is a screenshot of a bug in which the FSM output logic was incorrectly triggered by the clock.

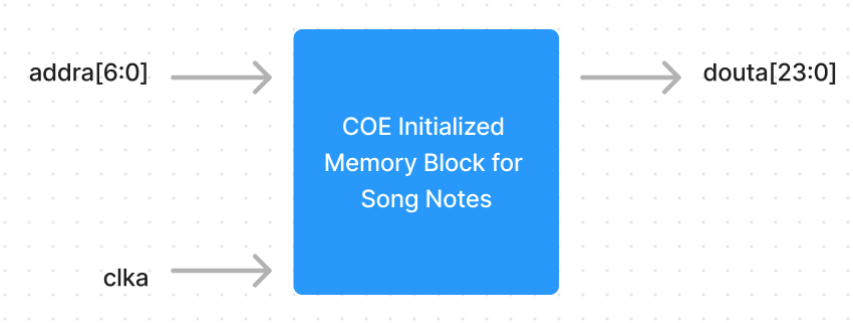
2.3 MIDI to VGA Interface

At the center of our design lies the three interface circuits: (1) BlockMover, (2) KeyCompare, and (3) DetermineHit. A fourth file, which allowed for variable beats per minute, and a fifth, for "overdrive" logic, were written and tested but not implemented.

2.3a COE Initialized Memory Block for Song Target Notes

Songs are written external to our system and imported using a block memory initialized by a COE file. Songs on the Keytar Hero system are represented by Tx24 bit vectors where T/4 is the number of measures in the song (four beats per measure). Songs are written in COE files using a binary radix. Future work could include making a Python or MATLAB helper function to generate songs from user inputs or from MIDI song files.

In Vivado, the COE file is instantiated as a Memory Generator IP Core. The "Single Port ROM" memory type allows other circuits to read the 24-bit vectors from the COE file. This blk_mem_gen_0 has an address (addra), clock, and data output (dout). The core is set to "Always Enabled" and "Write First". We load the initial COE file of interest and buffer the remaining bits with 0's. Like the Boolean Matrix, this memory block has a two-clock period latency.



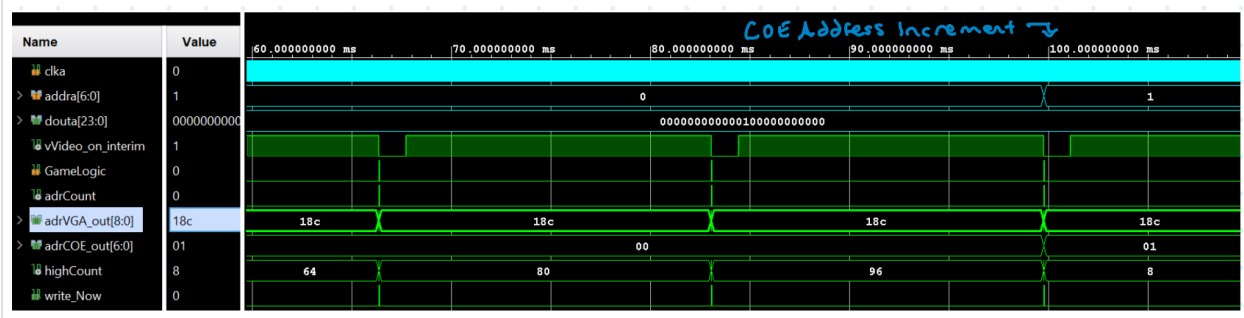
COE Initialized Memory Block Variables		
Variable	**Type**	**Notes**
addra	std_logic_vector(6 downto 0)	an address for the ROM
clk	std_logic	a clock for the ROM
dout	std_logic_vector(23 downto 0)	the data output of the ROM

To test this system, the shell was configured in simulation mode to show the counting of the COE file address variable. The COE address increments every time the notes are shifted 100

pixels. At first, from the simulation, a bug was discovered. Since the pixels are shifted in groups of four or eight at a time, the system was sometimes not reaching the reset value of exactly 96 counts. Rather than update the note and then clear the count in the next clock cycle, an extra signal was implemented such that the COE address count was incremented at the same time that the 100-count counter was reset.

Next, moving to hardware tests, a middle C4 to C5 run and also the opening notes of "Twinkle Twinkle Little Star" were written in the COE file. It was determined that the initial keyboard we were testing on was sending the incorrect notes from the device. We expected this to be a hardware defect: we determined this behavior using the oscilloscope directly on the first keyboard. The SCI_in bit in our top-level shell represents the Rx received by the MIDI. The Rx_done_output (described later in the MIDI receiver section), was also sent to the oscilloscope via a constraint file for testing. For example, the MIDI mode for the C note was note the 60 value as documented online. We tested a second keyboard using the oscilloscope which correctly sends notes as documented online. Lastly, we tested on a Yamaha SHS 200 Keytar which successfully interfaced with our system.

COE Initialized Memory Block Debugging Variables		
Variable	**Type**	**Notes**
SCI_in	std_logic	debugging output to directly see the input from the MIDI in the shell
Rx_done_output	std_logic	output from MIDI receiver representing when 8 bits of MIDI signal had been sent.

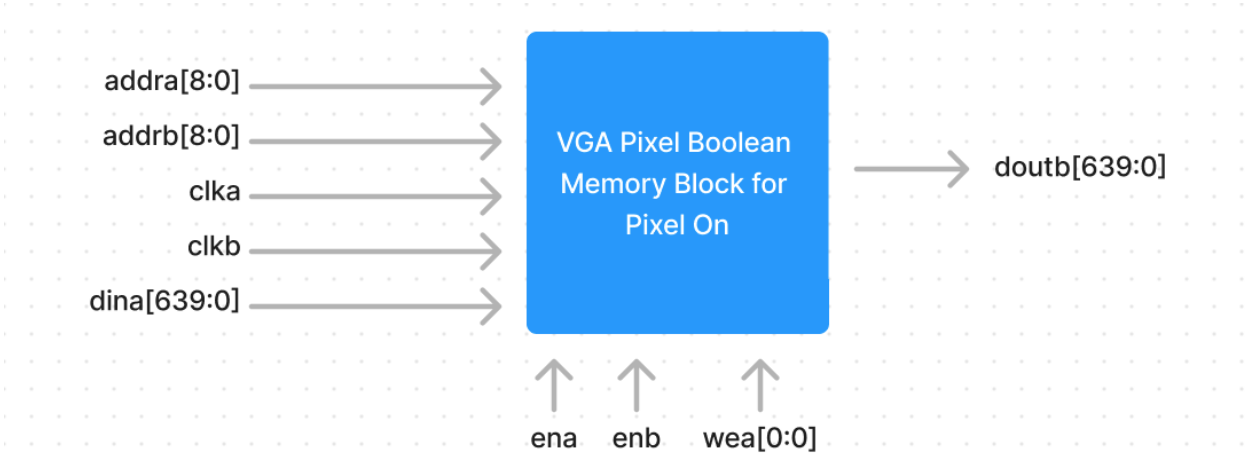


Typed Annotation: COE Address Simulation. When notes are shifted 100 pixels (counted by highCount, see discussion on BlockMover below), the address to the COE file increments, and the next value is output. In this case, the first and second values in

the COE file are both C notes, as represented by the 24 bit vector. Each bit represents one of twelve notes for either the right or left hand.

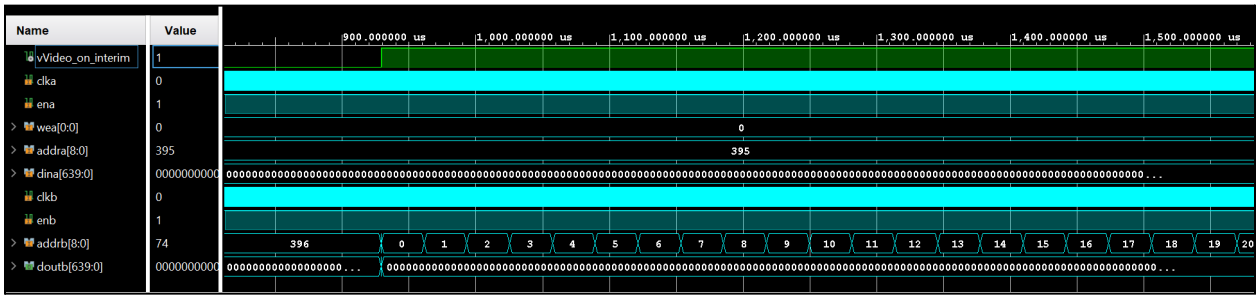
2.3b Memory Block for Pixel Booleans

A boolean matrix holds the VGA screen's pixel on/off values. The MIDI and VGA both connect to this blk_mem_uut block memory IP core. This memory block is a simple dual-port RAM. Port A, the writing port, includes an address (aadra), clock (clka), input line for writing (dina), a write enable (wea), and an enable which was wired high (ena). Port B includes an address (aadrb), clock (clkb), output line for writing (dout), and enable (ena). This block has a width of 640 and a depth of 480, representing the 480 rows and 640 columns of the VGA.

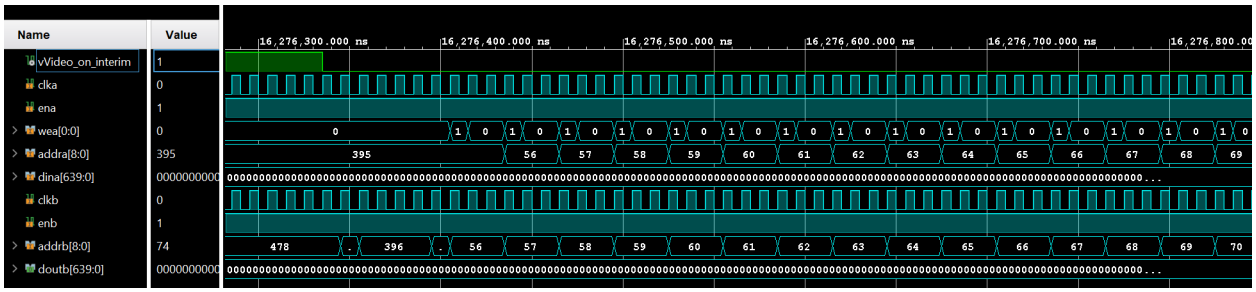


VGA Pixel Boolean Memory Block Variables		
Variable	**Type**	**Notes**
addra	std_logic_vector(8 downto 0)	Address for writing.
addrb	std_logic_vector(8 downto 0)	Address for reading.
clka	std_logic	Clock for writing.
clkb	std_logic	Clock for reading.
dina	std_logic_vector(639 downto 0)	Data to write.
ena	std_logic	Enable Port A.
enb	std_logic	Enable Port B.
wea	std_logic_vector(0 downto 0)	Enable writing.

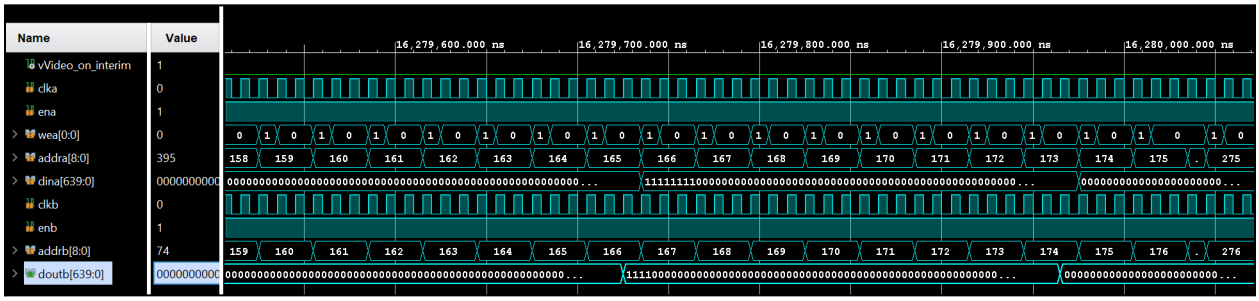
doutb	std_logic_vector(639 downto 0)	Data to read.



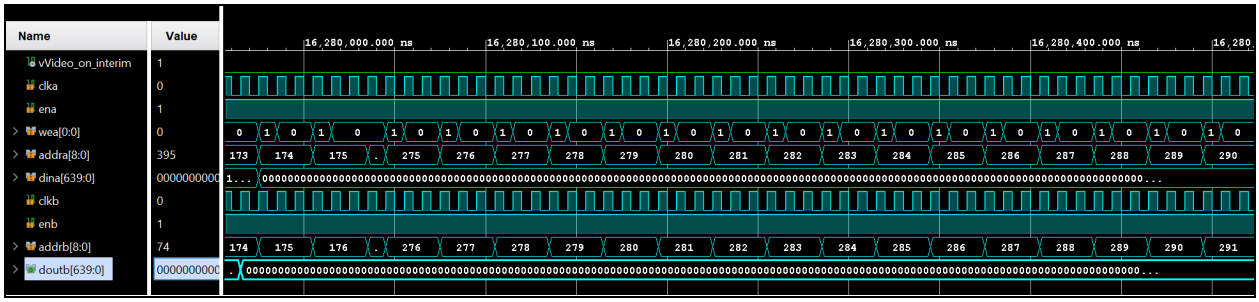
Typed Annotation: Memory Block for Boolean Pixels (VGA Reading). When vVideo_onInterim is high, the VGA circuits read from the Pixel Boolean Matrix to determine which pixels in the given row should be high. Here we see the address of Port B increment as the VGA moves through the rows of the Pixel Boolean Matrix.



Typed Annotation: Memory Block for Boolean Pixels (BlockMover Writing). Here we see that BlockMover controls the write enable of Port A and increments the address of the Pixel Boolean Matrix.



Typed Annotation: Memory Block for Boolean Pixels (BlockMover Shifting). As BlockMover arrives at a note that is high, the current values of the row (four 1's) are read and four more 1's are shifted in. This occurs for all rows encapsulating the height of the note.



Typed Annotation: Memory Block for Boolean Pixels (BlockMover Bounds). BlockMover only reads and writes where notes occur on the screen (from rows 56 to 394). At row 179, the BlockMover jumps the address from the bottom of the top staff to the top of the bottom staff which occurs at 275.

2.3c BlockMover Shifts Notes Across the Pixel Boolean

First, the BlockMover circuit will be considered. A finite state machine has five states: GameLogicWaitSt, StartSLTopst, SLTopst, StartSLlowst, and SLLowst. The block mover waits for a monopulsed signal to tell it that the VGA is between screen prints (when vVideoOn is off during the vertical sync). Only pixels on the top and bottom staffs are shifted as notes can only occur in this region. The FSM goes to the StartSL states to reset counters needed for moving the bottom and top staff respectively. In the SL states, the BlockMover shifts the booleans representing the notes, moving one row of the boolean matrix at a time. Debug states were written in this FSM and tested in simulation in order to ensure that the FSM was functional.

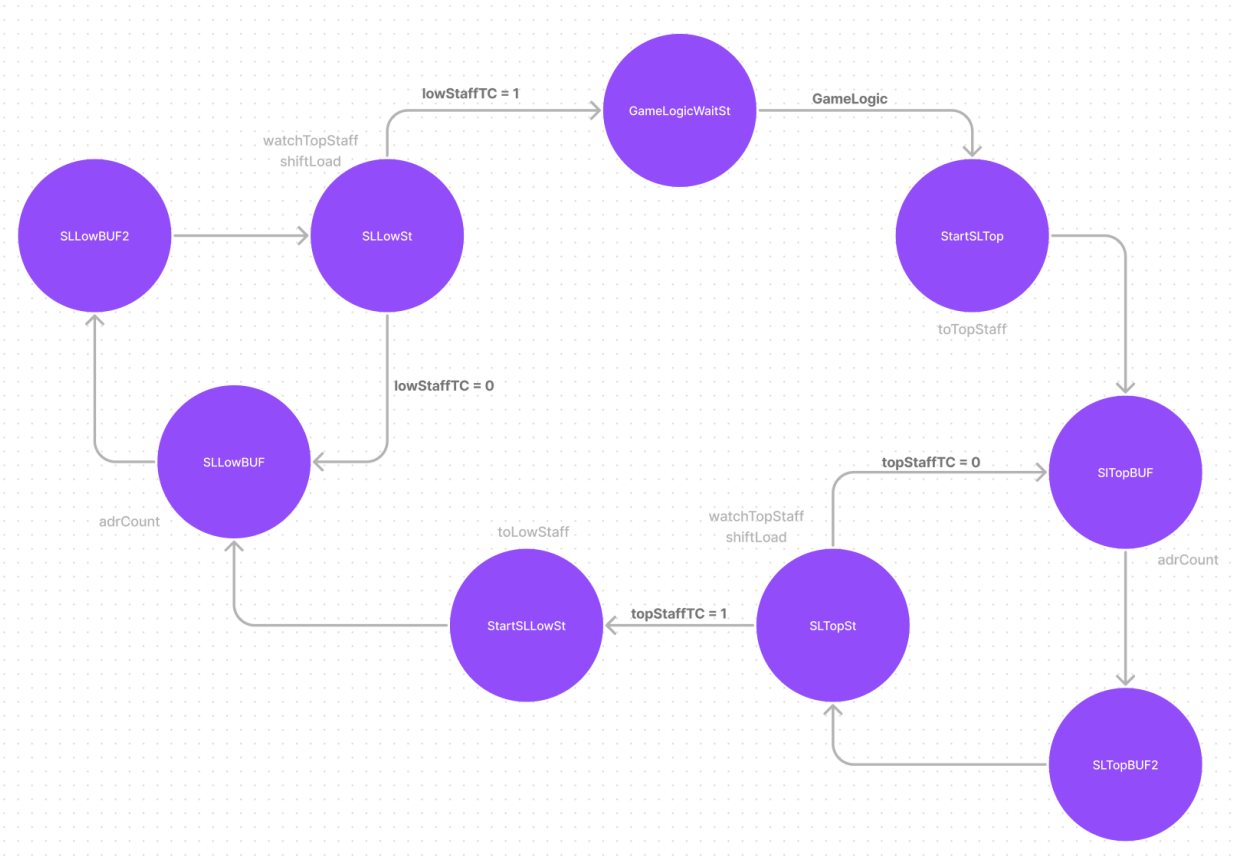
Block Mover Inputs and Outputs

BlockMover Inputs & Outputs		
Variable	**Type**	**Notes**
clk	std_logic	system clock

GameLogic	std_logic	monopulsed signal representing when the screen is off, determined by vVideoOn and helper processes
shiftNPixels_out	std_logic_vector	bits representing how many pixels to shift (1= 4pixels, 2=8 pixels, additional speeds could be implmented and this signal could be used for pause play logic)
RData_in	std_logic_vector(640 downto 0)	represents one row of the Boolean Memory Block
TargetNote_in	std_logic_vector(23 downto 0)	the interface between Block Mover and the COE file initialized memory block with the notes to be played
WData_out	std_logic_vector(640 downto 0)	represents one row of the boolean memory block, for writing
adrVGA_out	std_logic_vector(8 downto 0)	address of the boolean block memory
adrCOE_out	std_logic_vector(8 downto 0)	address of the COE file initialized block memory
endGame	std_logic	currently unused signal for pause/play/end game logic
writeNow	std_logic	the write enable bit for the boolean block memory

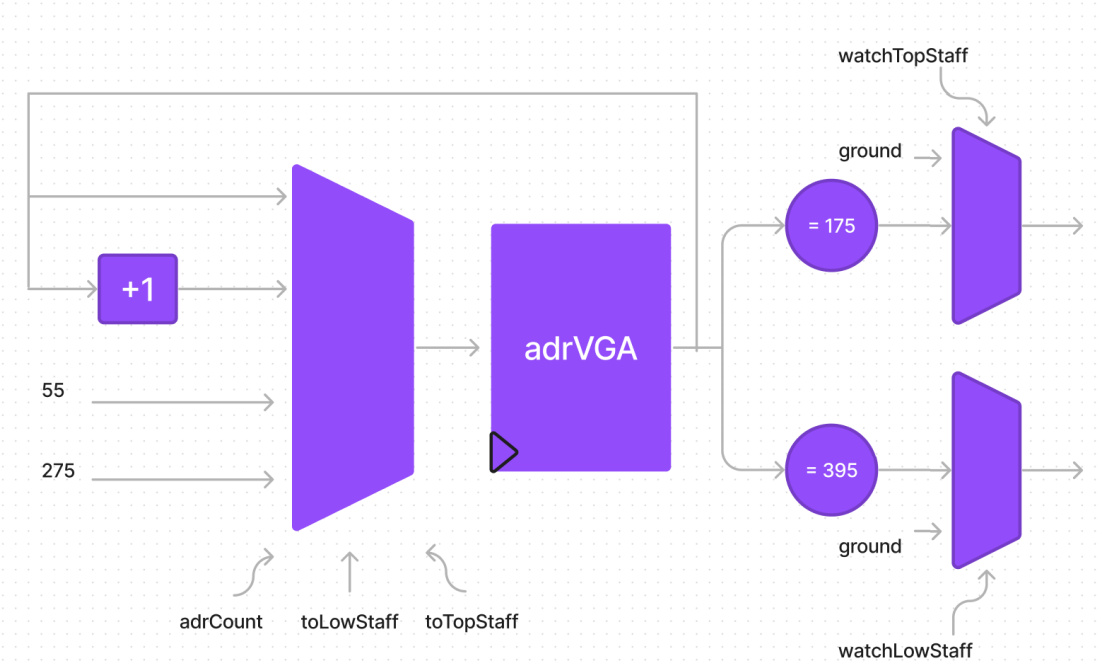
In the GameLogic wait state, the Block Move FSM sets the watchTopStaff and watchLowStaff signals to zero. In the StartSL states, either the toTopStaff or toLowStaff signals are activated. From the simulations, it was determined that a buffer state was needed since the Boolean Memory Block has a 2-clock cycle latency. Therefore, the adrCount signal is high in the SLBUFs states. In summary, the StartSL states set what staff should be written; the SLTopBUF

increments the address; the SLBUF2 allow the address to get to the memory block and take into account the latency.



Block Mover FSM. The Block Mover FSM controls the reading and writing from the Pixel Boolean Memory

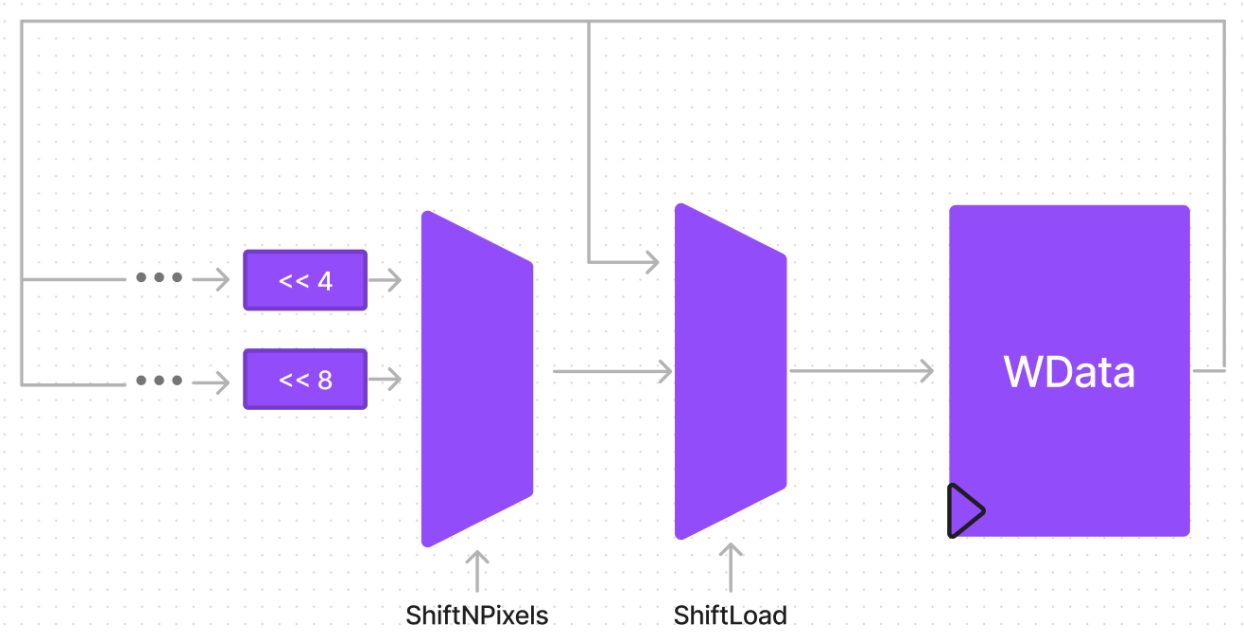
The address counter datapath increments the address being sent to the VGA if adrCount is high. The counter jumps to 55 (the 55th row is the beginning of the top staff) if toTopStaff is high. The counter jumps to 275 (the 275th row is the beginning of the low staff) if toLowStaff is high. The topStaffTC ends with the end of the top staff at the 175 row of the matrix. The lowStaffTC ends with the end of the bottom staff at the 395 row of the matrix.



Block Mover Pixel Boolean Address. The Block Mover Pixel Boolean Address loops through rows 55 to 175 and 275 to 395 (the locations where note shifts can take place).

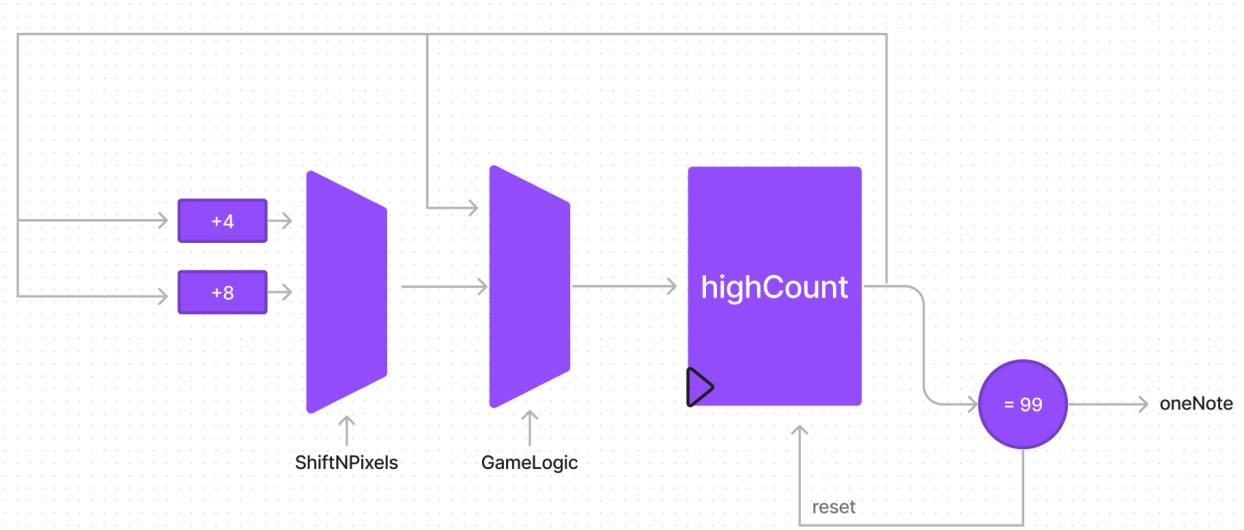
adrVGA Mux Logic			
adrCount	**toLowStaff**	**toTopStaff**	**adrVGA**
-	1	0	275
-	0	1	55
1	0	0	adrVGA+1
1	1	1	adrVGA+1
0	0	0	adrVGA
1	1	1	adrVGA

The shift circuit datapath includes code to create the data to be written into the memory block. This circuit maps the note from the COE file to the rows of the boolean matrix. For example, if the note from the COE file is a right hand G, the COE file targteNote(0) will be high. When the 57th through the 65th row of the boolean matrix is selected, either four or eight 1's will be shifted into the existing boolean values such that the illusion of motion will be created. The superWrite signal shadows the write enabling signal to time the writing of the data into the block memory to take into account the 2-clock latency. ShiftLoad is controlled by the FSM.

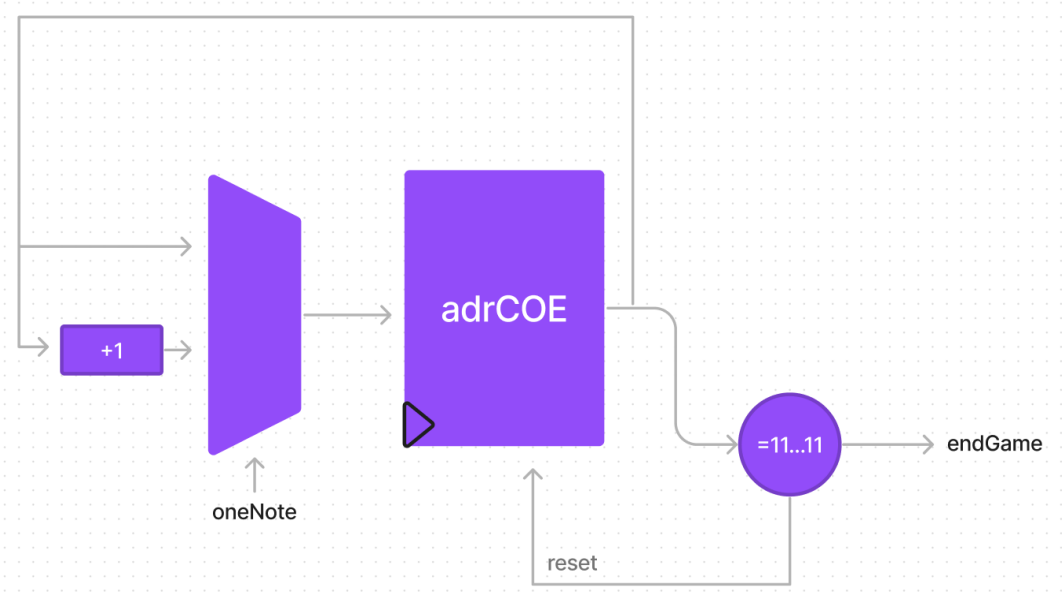


Block Mover Shifter Circuit. The Block Mover shifter circuit can move booleans (representing notes) either four or eight pixels depending on the beats per minute. The ellipses represent the case statement determining which pixels map to which notes.

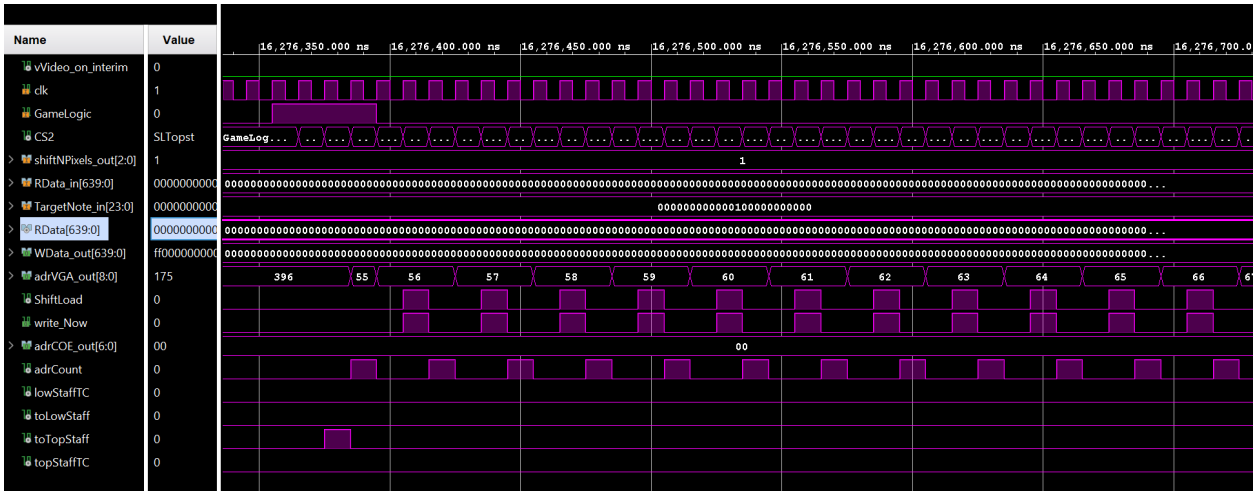
The note width counter determines when 100 pixels shifts has occured. This counter uses the highCount count along with the oneNote terminal count signal to drive the COE file address increment. If oneNote is high, the COE address is incremented. When a COE file input with all 1's is reached, the COE file counter resets. The COE file must be formatted with this signal in mind.



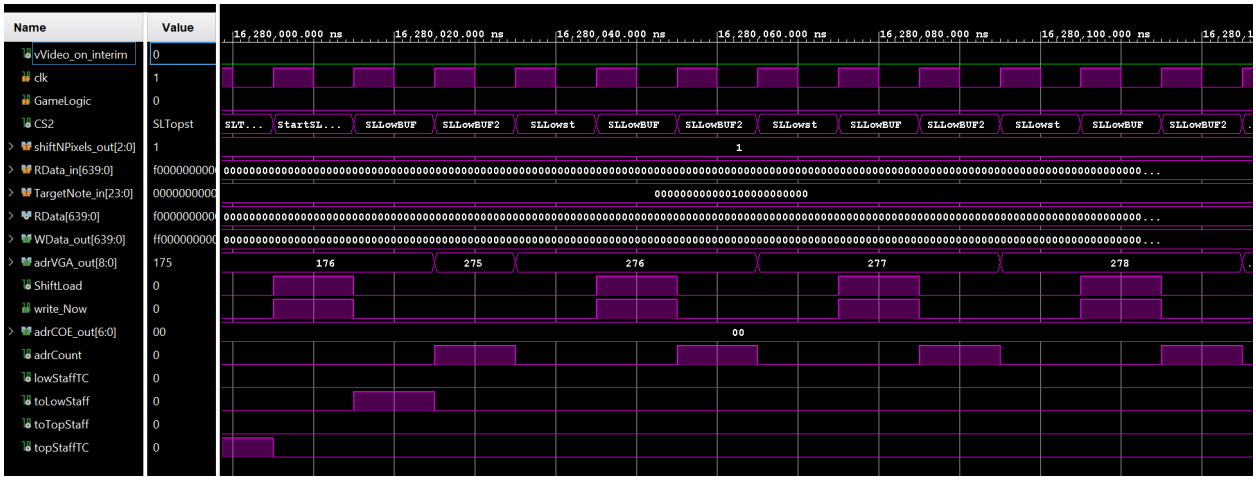
Block Mover Note Counter. The Block Mover note counter parallels the shifter. It counts how many pixels have been shifted and mono-pulses oneNote high whenever 100 pixels have been shifted.



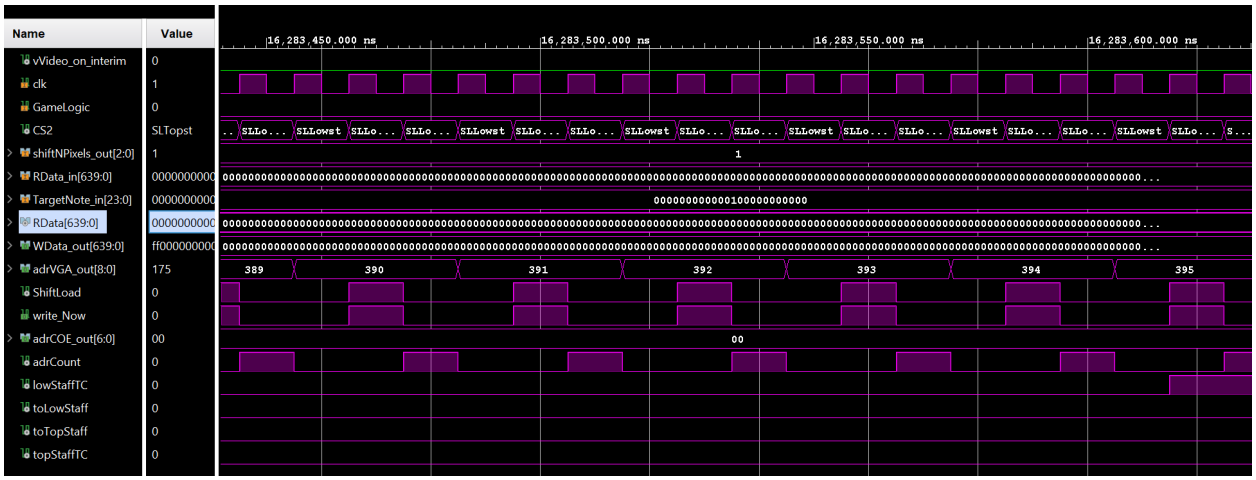
Block Mover Song Looper. The Block Mover song looper moves through the address of the COE file until 1111…1111 is hit at which endGame is monopulsed and the adrCoe is reset. Every time one note (100 pixels) pass, the COE file memory block address increments such that the next note can be taken.



Typed Annotation: BlockMover Starts on the Top Staff. After receiving GameLogic, the mono-pulsed version of the inverted vVideoOn signal, the counter goes to 55 (the top of the top staff, and begins to read and write into the Boolean matrix.

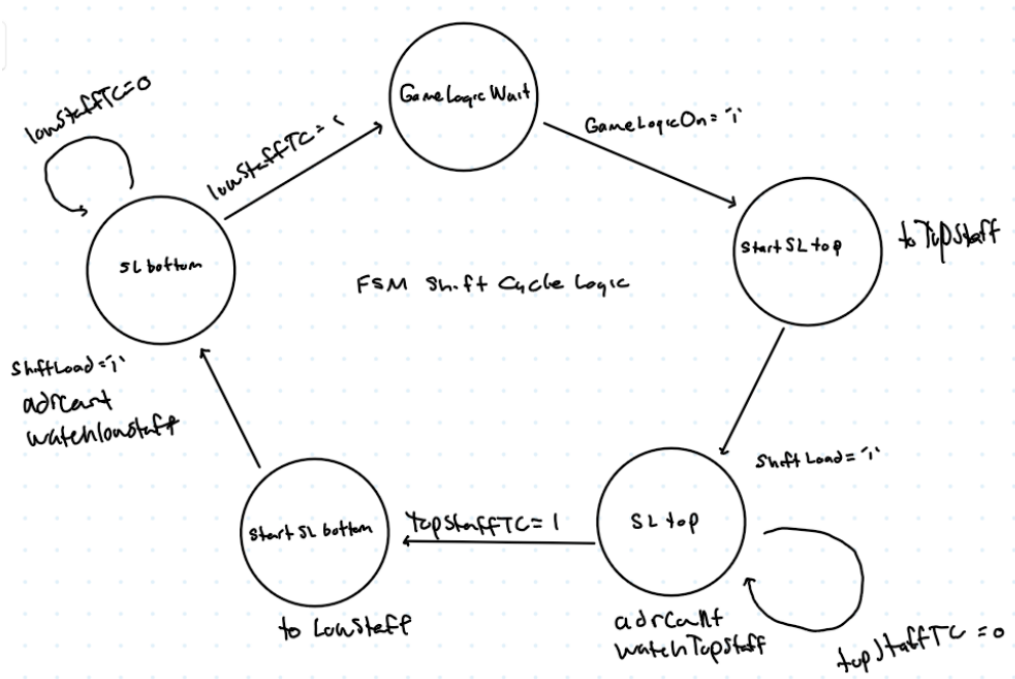


Typed Annotation: BlockMover to Low Staff. After ToTopStaffTC goes high, the BlockMover is finished shifting and writing the top staff values. It jumps to the low staff when the toLowStaff signal is received. This screenshot shows the first note in the COE file (adrCOE = 0) whose value is TargetNote. The ShiftLoad and write_Now determine when the Boolean Memory Block is written. In this case, the bits in each row of the Boolean are zero and not values are shifted out such that the write boolean is zero. In future work, we could make it only write if there was a change. The FSM loops through the states as expected and as outlined above.

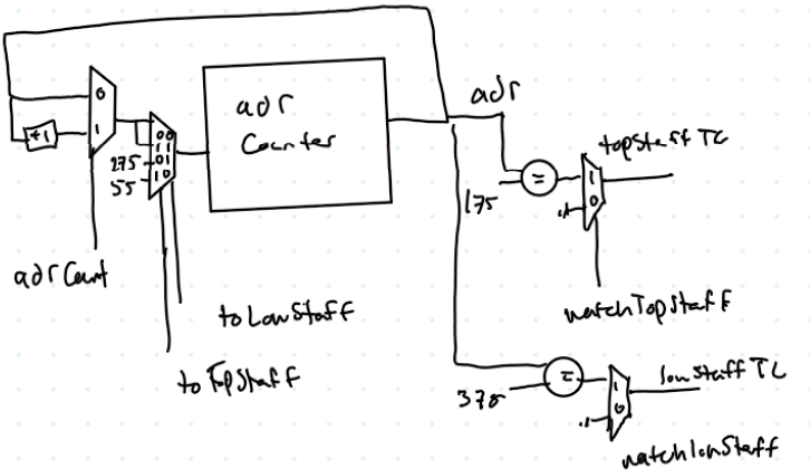


Typed Annotation: BlockMover to End of Low Staff. At the end of the low staff, the lowStaffTC goes high.

Below is a more detailed description of BlockMover from Checkpoints 2 and 3.



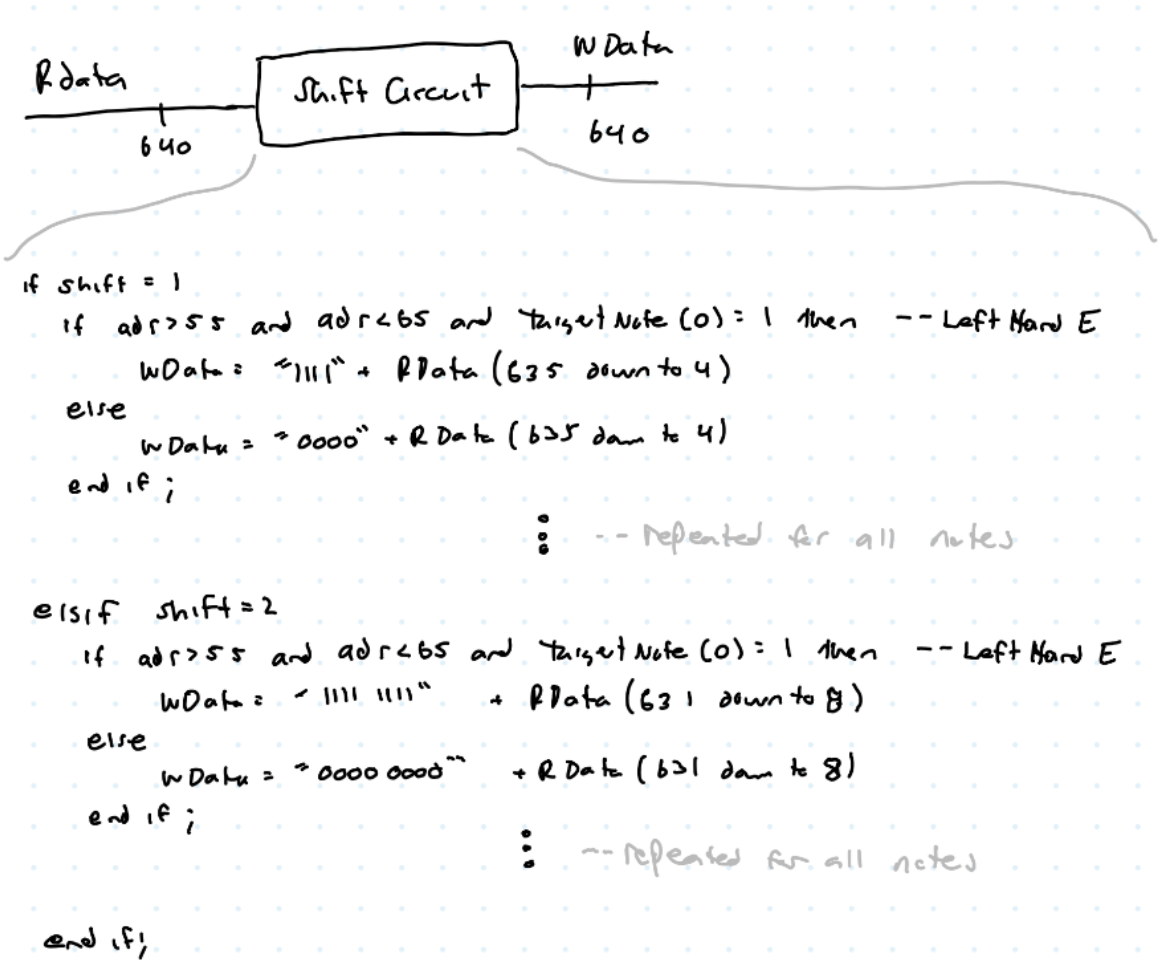
Finite State Machine for NoteDisplay Adress Shifter. Every screen cycle, when game logic can be changed, the addresses of the NoteDisplay memory block in which values will change are called and updated.



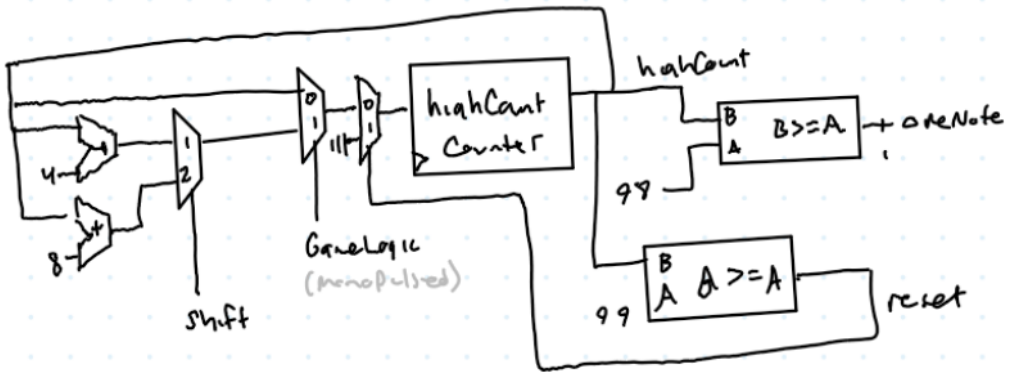
Address Counters. Two address counters count between 5 and 175 and 275 to 395 since only these rows have notes moving.

The figures above outline an FSM that determines when to shift the old notes and loops through the addresses of the memory to determine when to update. We only address rows that contain notes as the other positions are static.

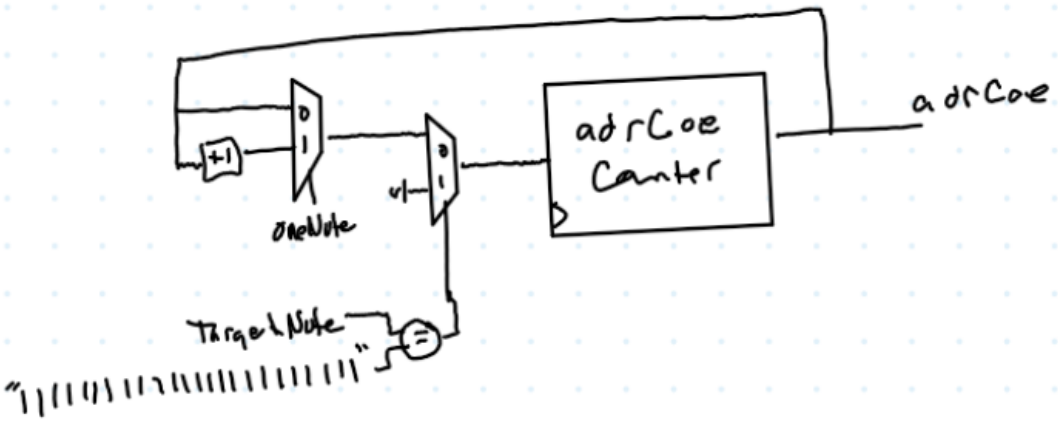
When shiftLoad is high, the read data from a specific row of the memory block will be updated.



Variable Shifter Pseudocode and RTL Diagram. If shiftLoad = 1, the shift circuit will be active, incrementing one row of the display at a time.



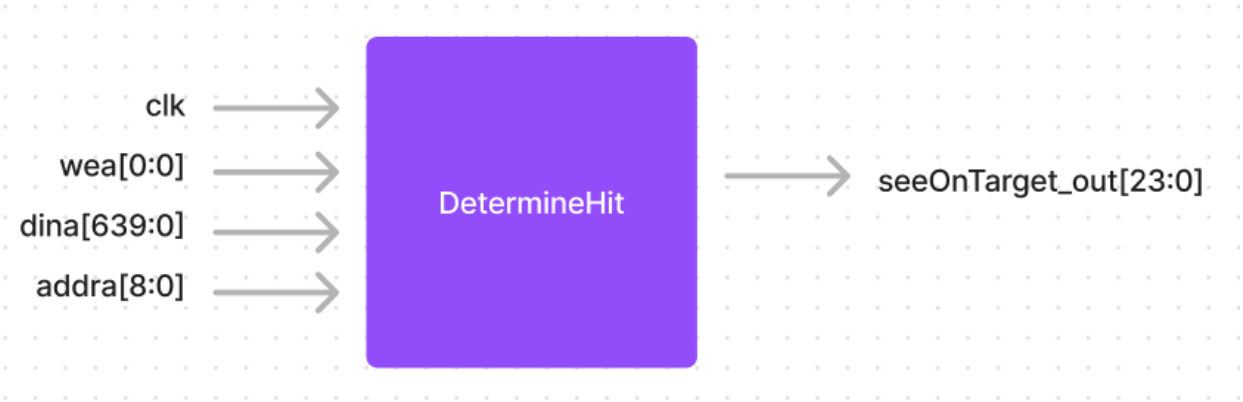
Note High Counter. Measures 100 pixel shifts then sets OneNote high and resets.



COE File Counter. Whenever 100 pixels are shifted, the COE file address is incremented.

2.3e DetermineHit Pulls Notes from the Target on the Screen

While the BlockMover circuit shifts notes into the boolean matrix, the DetermineHit circuit pulls the boolean values from the target to determine which notes are on the target. This circuit has a clock (clk), gets the write enable from the memory block (wea), receives the data being written into the memory block (dina) from BlockMover, receives the address into the memory block (addra) from BlockMover, and outputs a 24-bit vector representing notes on at the target (seeOnTarget_out).

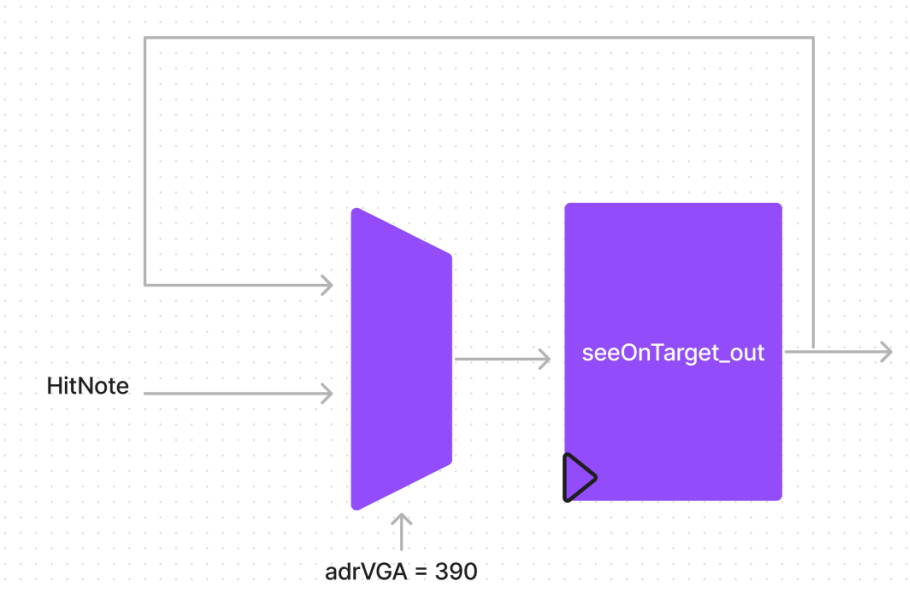


BlockMover Variables		
Variable	**Type**	**Notes**

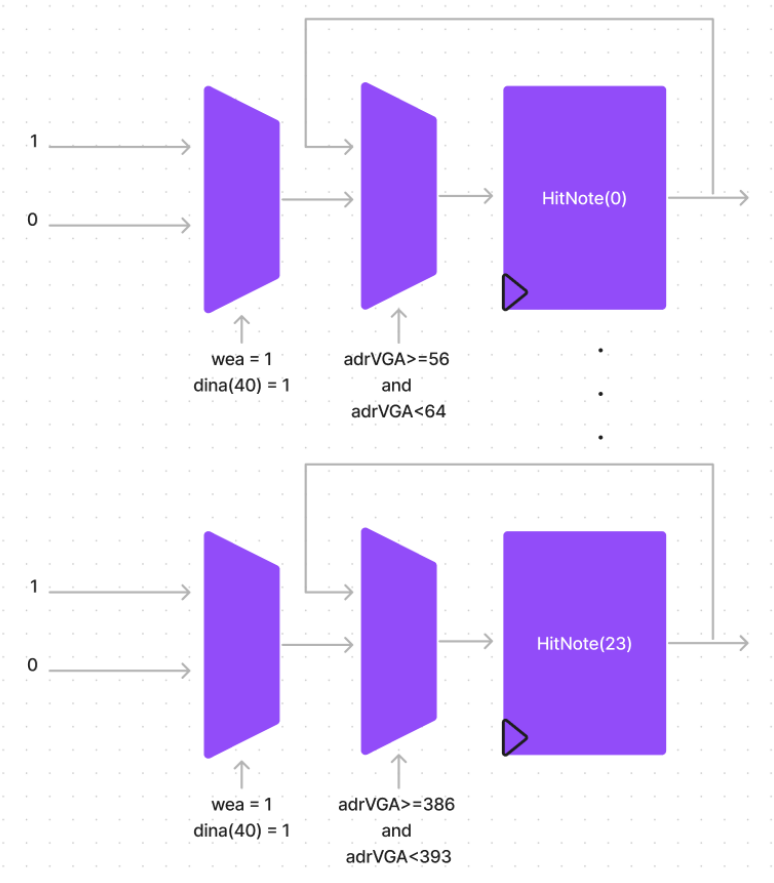
clk	std_logic	System clock
wea	td_logic_vector(0 downto 0)	Write enable bit, used to determine when not to read from the Pixel Boolean Memory
dina	td_logic_vector(639 downto 0)	One row of the Pixel Boolean Memory
addra	td_logic_vector(8 downto 0)	Address of the Pixel Boolean Memory
seeOnTarget_out	td_logic_vector(23 downto 0)	Which of the 24 notes are “On” at the target
adrVGA	integer	Address of the Pixel Boolean Memory
HitNote	std_logic_vector(23 downto 0)	An internal variable for which of the 24 notes are “On” at the target

As the BlockMover writes the shifted rows to the memory block, DetermineHit also receives this data input and pulls the 40th column (the location of the target). Depending on the row selected, the note on is determined. For example, if a right-hand C is on the target, then the 40th column of rows 56 to 64 will be high and HitNote(0) will be set to one. The HitNote is updated each time the write bit goes high on the rising edge of the clock. Debugging on the oscilloscope and simulation was performed to determine the timing of the memory block counter and the Rx coming in from the MIDI. The HitNote is sent out with every screen update, triggered on the VGA row address.

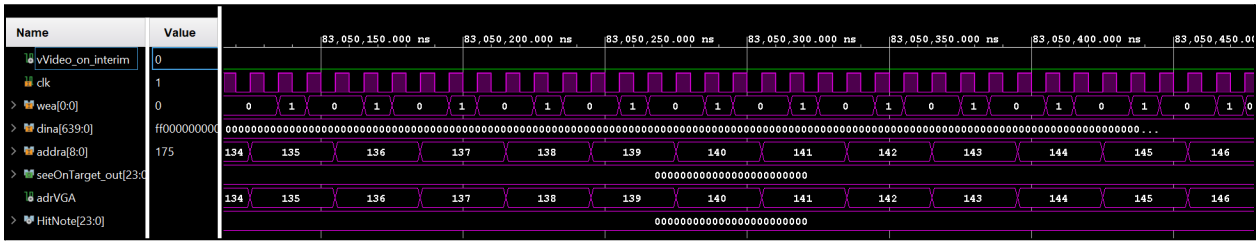
The logic is as follows: if the write enable is high (such that the next clock cycle it will be low), if the 40th bit of the row, corresponding to the target is high, using the 480-pixel-to-24-note map, the correct note is send as an output.



DetermineHit Output Push. At the end of each screen update, the new seeOnTarget_out is pushed from the internal HitNote signal.



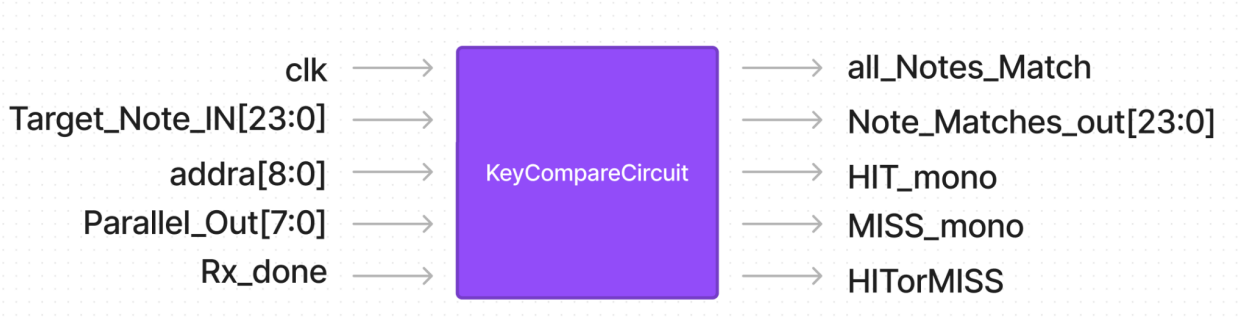
DetermineHit Hit Note Logic Chain. Two of the 24 circuits for setting the HitNote variable. The circuits map the pixel rows to the 24 notes using the 480-pixel-to-24-note mapping.



Typed Annotation: DetermineHit Checks If Anything is at the Target. Each time a row is read and written, DetermineHit Checks to see if the pixel at the target (bit 40), is high. If so, from the 480-pixel-to-24-note map, the noteOnTarget_out is set.

2.3f KeyCompare Compares the Target Note to What the User is Playing

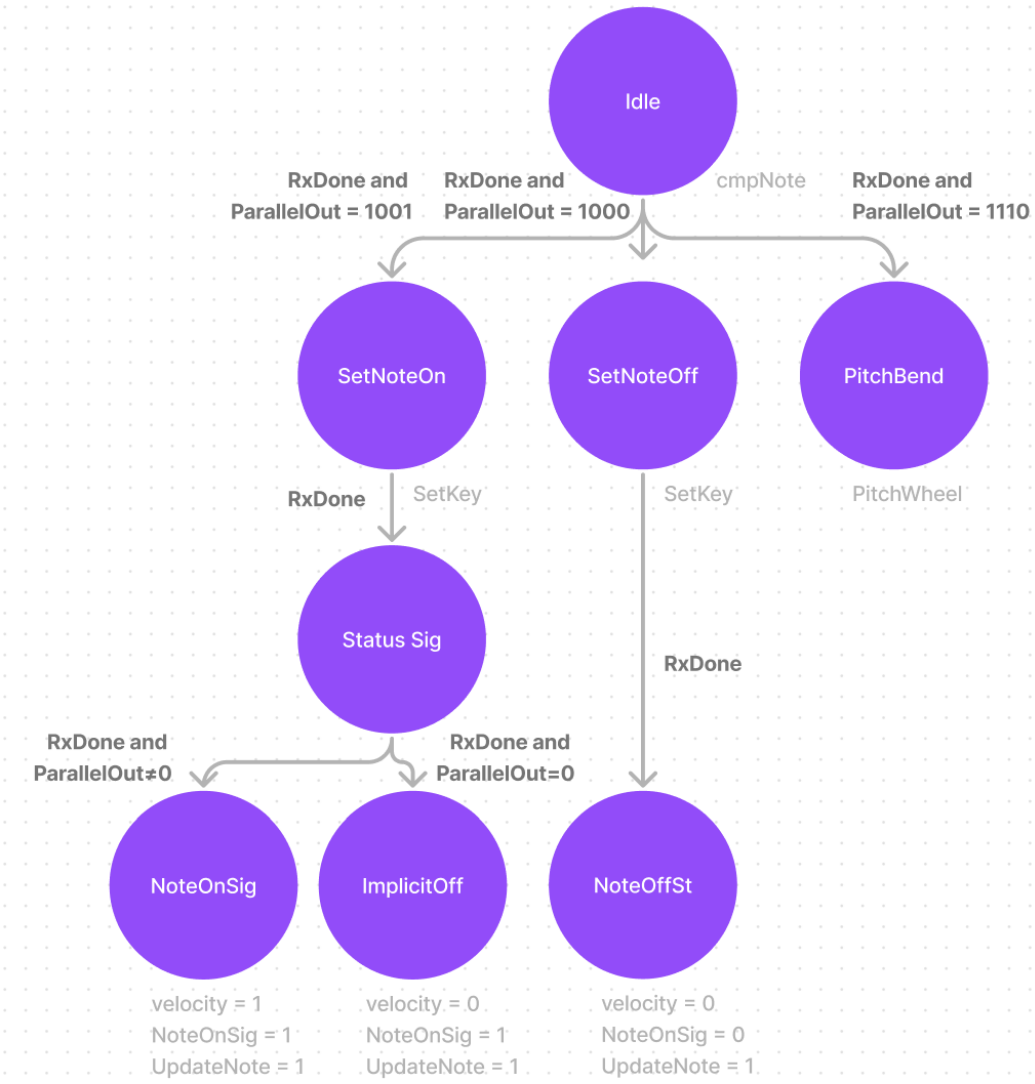
Next, the KeyCompare circuit will be considered. This circuit achieves the following: (1) determines the note being sent from the MIDI receiver (NoteOn), (2) compares to the note on at the target generated by DetermineHit (the TargetNote variable), (3) generates a 24 bit vector of matches (the Note_Matches variable). A match occurs if the target note is high and the user is pressing the note. Users are only penalized if they miss a note, that is, they are not penalized for pressing additional notes. This allows users to play extra embelishments. For example, if a right hand C is on the target, Target_Note(0) will be high. If the user presses the correct C note, NoteOn(0) will go high and Note_Matches(0) will be high along with the other bits of Note_Matches. NoteOn(7) also represents a right hand C so either will set the match variable high. Missing a C in the right hand will cause Note_Matches(0) to go low. When Note_Matches contains all 1's, a master HIT variable will go high. This variable is used by the VGA display to change the color of the note.



To generate NoteOn, KeyCompare takes the ParallelOut of the MIDI receiver. When the receiver Rx_done signal goes high, the first of the three eight bit signals is read. The first bit represents the MIDI status signal. The cmpNotes debug signal (one bit) and DEBUG_STATES (four bit)

outputs were used when testing this circuit's FSM on the oscilloscope. The WHATSTATE_DEBUG, a one hot vector representing the FSM states, was used to debug simulations in Vivado and EDA Playground.

From the idle state, once the status signal was read, the FSM goes into the SetNoteOnSt, SetNoteOffSt, or PitchBend. The PitchBend signal and overdrive system (including a points counter and counter to determine how long the user should be in overdrive for) were written but not implmented. Future work could implment this point system. When in overdrive, notes would turn yellow and points would be doubled. If a note was determined to be on or off, the buffer sates SetNoteOffSt or SetNoteONSt were set. The setkey signal was set high and the FSM waied for the next Rx_done. In the NoteOffSt, updateDate was set high but the velocity and NoteOnSig were low. The NoteOnSt has to wait for an additional Rx_done in order to read the velocity bit. Some keyboards include an implicit off signal. In the implicit off case, a Note On status signal is sent but a velocity of 0 is sent signifying the note to be on even through the status signal was a note on. In StatusSigSt, it is determined if the velocity is zero. If so, the FSM goes to the implicit off state, setting NoteOnSig to high but velocity to low. Otherwise, the FSM goes to the NoteOn state with NoteOnSig and velcity signals both high. Note the velocity signal I use does not capture the value of the velocity.

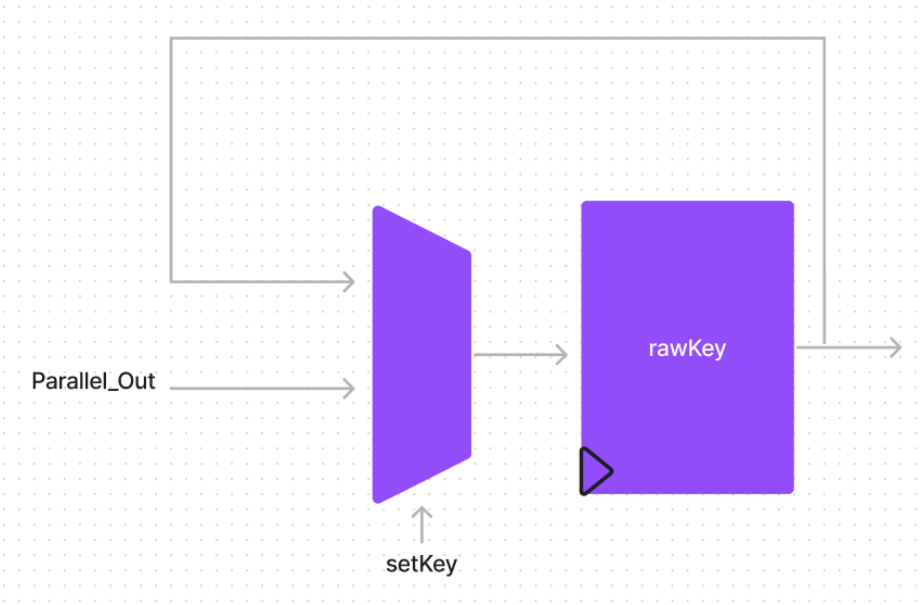


KeyCompareCircuit FSM. The KeyCompareCircuit FSM waits for the RxDone signal. For each MIDI event, it receives three signals: (1) a status signal, (2) the note signal, and (3) the velocity signal. The FSM will only leave idle when one of three status signals are hit: Note On (1001), Note Off (1000), or Pitch Wheel (1110). Pitch Wheel is not used in this version but could be used to implement overdrive.

Key Comparator FSM Variables			
Variable Name	**Variable Type**	**Default Value**	**Information**
state_type	type	n/a	A variable to define the FSM states: IdleSt, StatusSigSt, SetNoteOnSt,

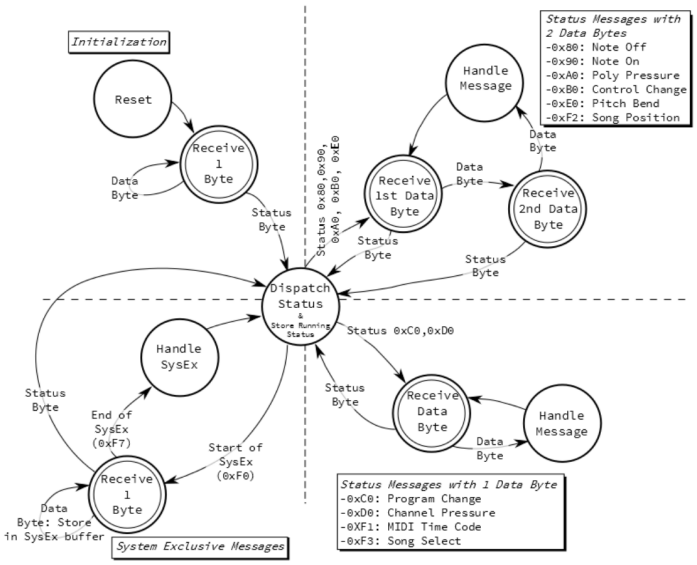
			SetNoteOffSt, PitchBendSt, NoteOnSt, ImpOffSt, and NoteOffSt.
NS	state_type	IdleSt	A variable for the next state.
CS	state_type	IdleSt	A variable for the current state.
setkey	std_logic	0	A signal to tell the circuit to set the key value to the Parallel_Out_Internal .
velocity	std_logic	0	A signal to determine if the velocity is zero or non-zero. Used to determine if implicit off.
NoteOnSig	std_logic	0	A signal to tell if a key set by setkey should be turned on or off.
updateNote	std_logic	0	Tells the circuit when to update the note vector. Must wait for the key to be set with set key and also for the velocity to be determined.
Parallel_Out_Internal	8 bit std_logic _vector	11111111	Latches onto Parlallel_Out such that signals can be determined even if the receiver shift register continues to shift.
Rx_done	std_logic	0	Tells when to read the value from the receiver.

Another process in KeyCompare sets the rawkey variable to ParallelOut when setkey is high in the SetNote states. When updateNote is high but NoteOnSig is low and velocity is low, the NoteOn 24 bit vector's nth bit is set to zero representing the note turned off. Otherwise, the nth bit is turned high representing a hit note.



KeyCompareCircuit RawKey Latch. The rawKey variable is set to Parallel_Out every time the setKey signal is hit.

Below, a more detailed discussion of the KeyCompare FSM from Checkpoint 2 and 3. We base our finite state machine on the FSM cited on the SparkFun Learn MIDI tutorial website [SparkFun, 2023]. Our FSM is a simplification of this state machine, focusing on Note On, Note Off, and Pitch Bend signals.



FSM Inspiration from SparkFun Learn. This finite state machine uses receive byte states and "Handle Message" states similar to our FSM.

An FSM was designed with states IdleSt, StatusSigSt, SetNoteOnSt, SetNoteOffSt, PitchBendSt, NoteOnSt, ImpOffSt, and NoteOffSt. By default pitchWheel, setkey, updateNote, velocity, and NoteOnSig will be zero. The system will wait in IdleSt. When the most significant digit of Parallel_Out and the Rx_done signal goes high, signifying a status signal, the system will move into StatusSigSt. Parallel_Out_Internal will latch to Parallel_Out such that the status signal can be read even if the receiver continues to shift outputs. Before the next signal comes in from the MIDI, while in StatusSigSt, the status signal will be read.

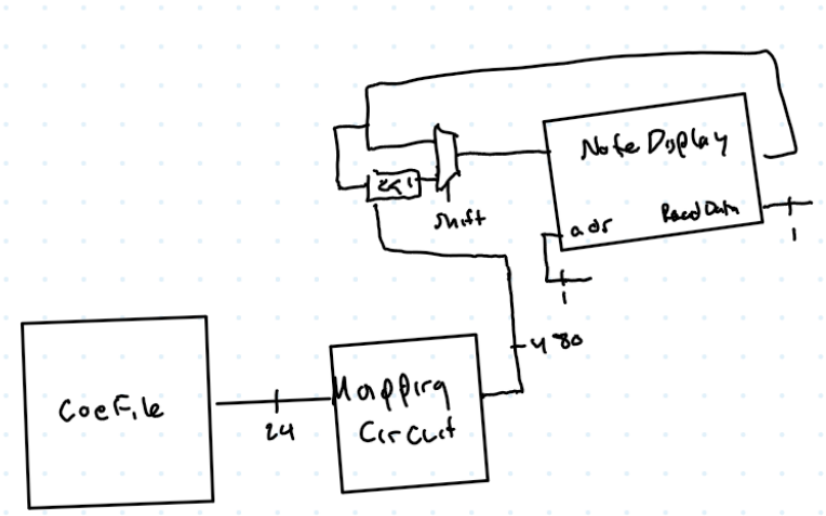
If the four most significant digits of Parallel_Out_Internal are 1001 and Rx_done signal is high then the system will go to the SetNoteOnSt state. If the digits are 1000 and Rx_done signal is high then the system will go to SetNoteOffSt. In both the SetNoteOnSt state and SetNoteOffSt state, setkey goes high.

If the digits are 1110 and Rx_done signal is high then the system will go to PitchBendSt. In the PitchBendSt state, pitchWheel will go high. The PitchBendSt state will then return to the IdleSt.

The SetNoteOffSt state will go to the NoteOffSt state when the Rx_done signal goes high. In the NoteOffSt state, updateNote is high while velocity and NoteOnSig are low. The NoteOffSt state will return to the IdleSt state. The SetNoteOnSt state must determine if the note has been turned on or if an implicit off has been sent. If the Parallel_Out_Internal is zero, the system will go to the ImpOffSt state when the Rx_done signal goes high. In the ImpOffSt state, updateNote is high, velocity is low, and NoteOnSig are high. If Parallel_Out_Internal is not zero, the system will go to NoteOnSt state when the Rx_done signal goes high. In the NoteOnSt state,

updateNote, velocity, and NoteOnSig are high. Both the ImpOffSt state and the NoteOnSt state return to the IdleSt state.

In a process, all the notes will be shifted to the left. Then the new bits will be loaded on the right edge. If the COE file mapping is high (e.g. the A note in the left hand is high), then the corresponding pixel locations on the screen will go high. Note that NoteDisplay is a binary representation if pixels should be on or off for notes. The VGA Display will read from the NoteDisplay to determine if a note is high.



NoteDisplay Shifter High Level RTL Diagram. The 24 bit std_logic_vector from the COE file will be mapped to columns of the NoteDisplay. Upon sihfting, the values will be left shifted such that the rightmost column goes high if the COE file mapping is high.

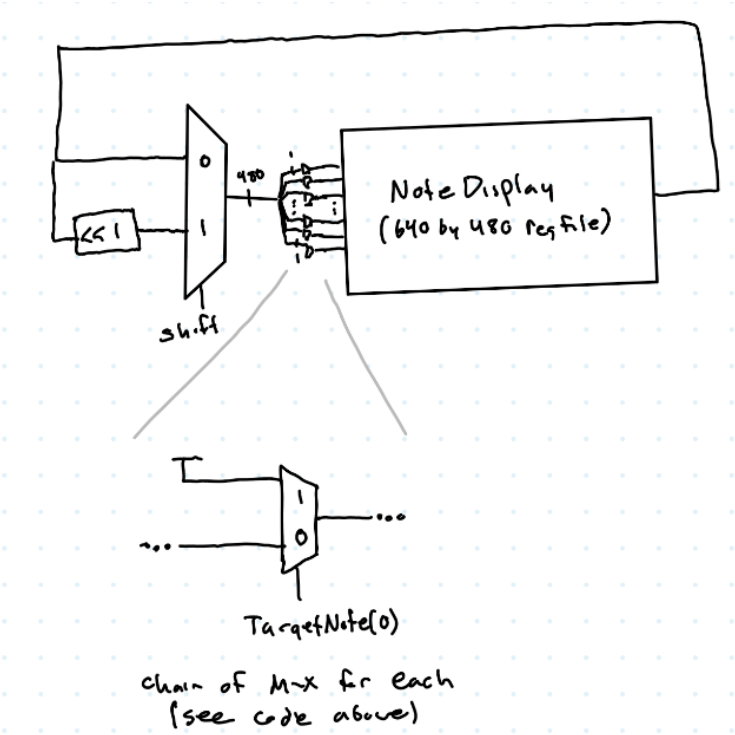
The following VHDL will be used to initialize and later shift the NoteDisplay:

```
type reg_file_type is
        array(0 to 3) of std_logic_vector(7 downto 0);
signal shift_reg: reg_file_type:= (others => (others => '0'));
        ...
shift_reg <= (others => ((22 downto 0)+"0"));
```

To load the rightmost bit into NoteDisplay, each bit in TargetNote will be mapped to the shift register:

```
if TargetNote(0) = '1' then
        Shift_reg <= ((55 to 65) => ((23 downto 1) + '1'));
elsif ...
```

The pixels in the rightmost column will be set note if the corresponding TargetNote is high. See the table above for the mapping between TargetNote and NoteDisplay. The shift mux will go high when the VarClock goes high.



NoteDisplay Shifter Block Diagram. The NoteDisplay shifter will shift the bits and then for each of the rightmost bits (rightmost column) decide if the pixels should go high.

The memory block holding booleans representing if pixels should be on or off can only be updated as the VGA resets its vertical scan. This occurs in 1.3 ms every 16.7 ms such that the screen updates 60 times per minute. Therefore, if we shift four pixels every update, we can only have a maximum of 240 increments such that 60 beats per minute is realized. If we shift 8 pixels every update, we have 480 increments such that 120 beats per minute is realized. For speeds between 60 and 120 beats per minute, a combination of 4-pixel and 8-pixel shifts can be alternated.

Speed - 60 = fourPixelShifts
60 - fourPixelShifts = eightPixelShifts

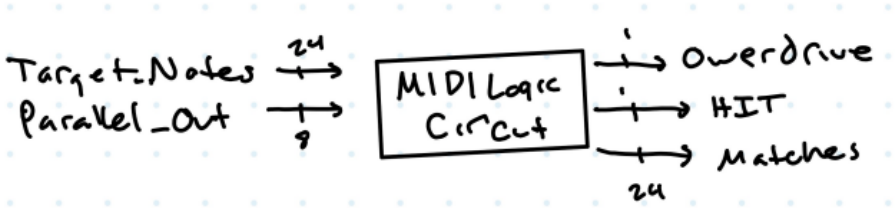
For example, for 65 beats per minute:

65 - 60 = 5
60 - 5 = 55

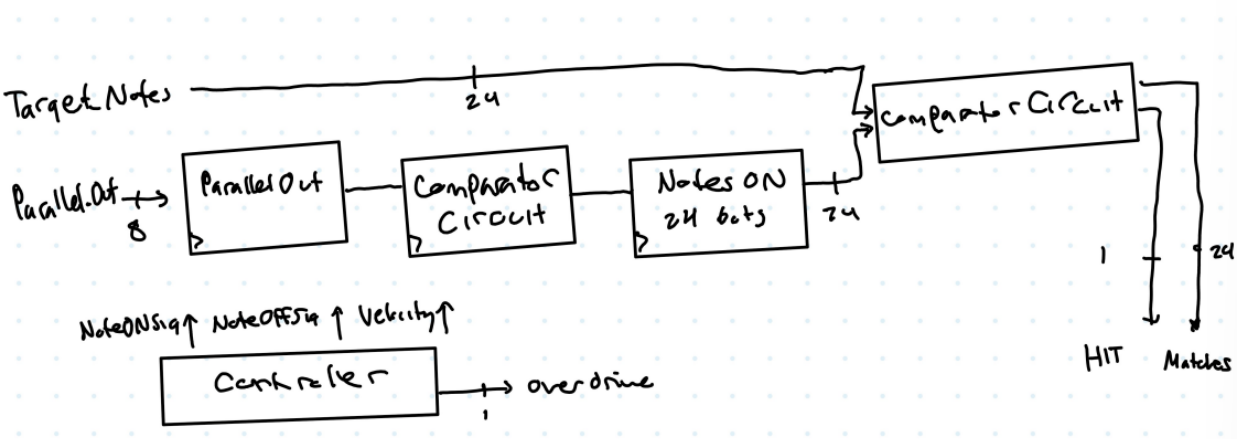
If we shift these values, alternating shifts of 4 and 8:

5(8) + 55(4) = 260 pixels/sec

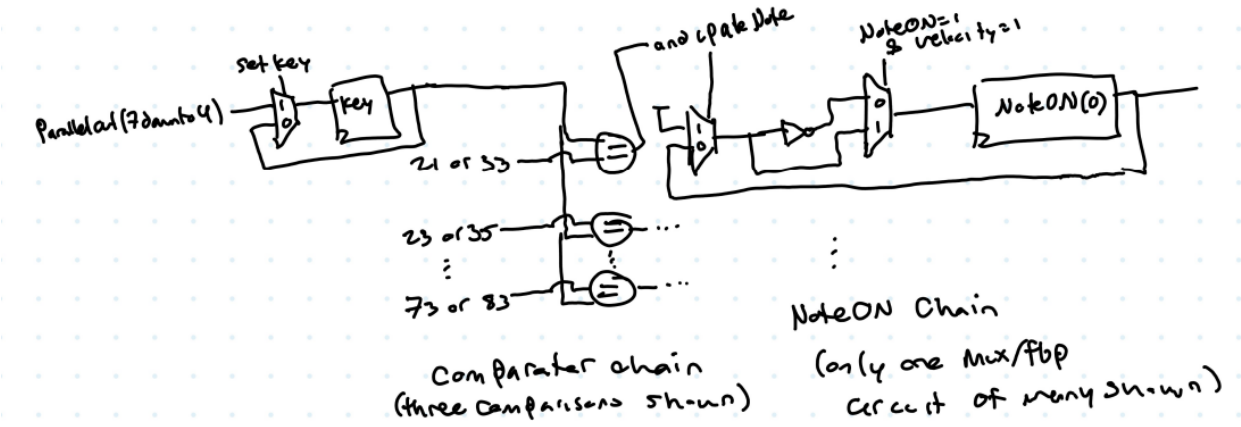
260 $\frac{pixels}{sec}$ $\frac{60\ sec}{min}$ $\frac{1\ note}{60\ pixels}$ $\frac{1\ beat}{4\ note}$ = 65 beats



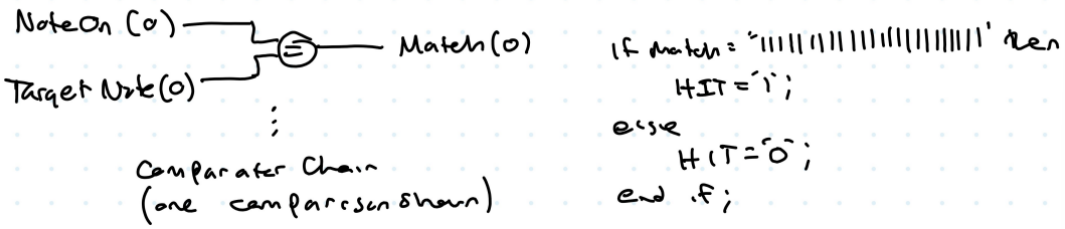
High-Level Input/Output Diagram. The original high level RTL block diagram for Key Compare. The target note on the VGA target and parallel output from the MIDI receiver are compared and output matches (a vector of hits), HIT (true if all of matches' hits are true), and overdrive.



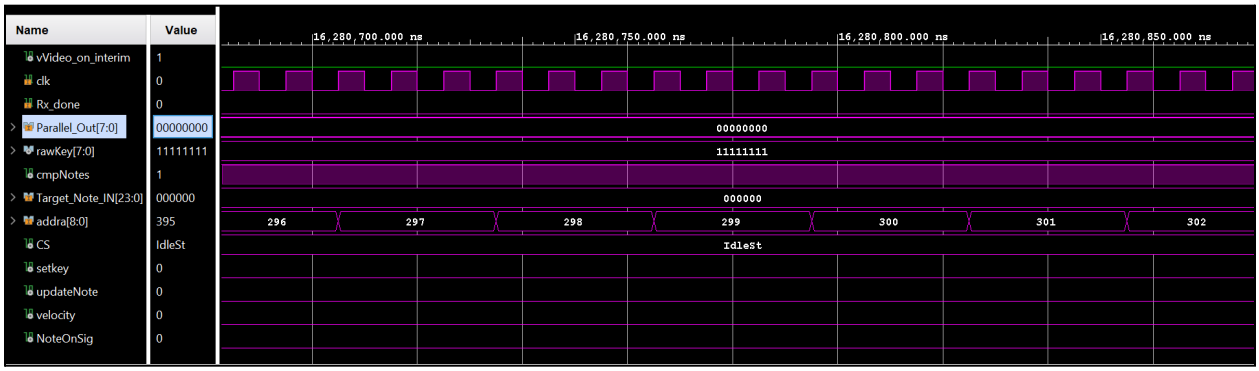
Key Comparator RTL Diagram. The 8 bit parallel out must be mapped to a 24 vector representing notes to be compared to the 24 bit target note.



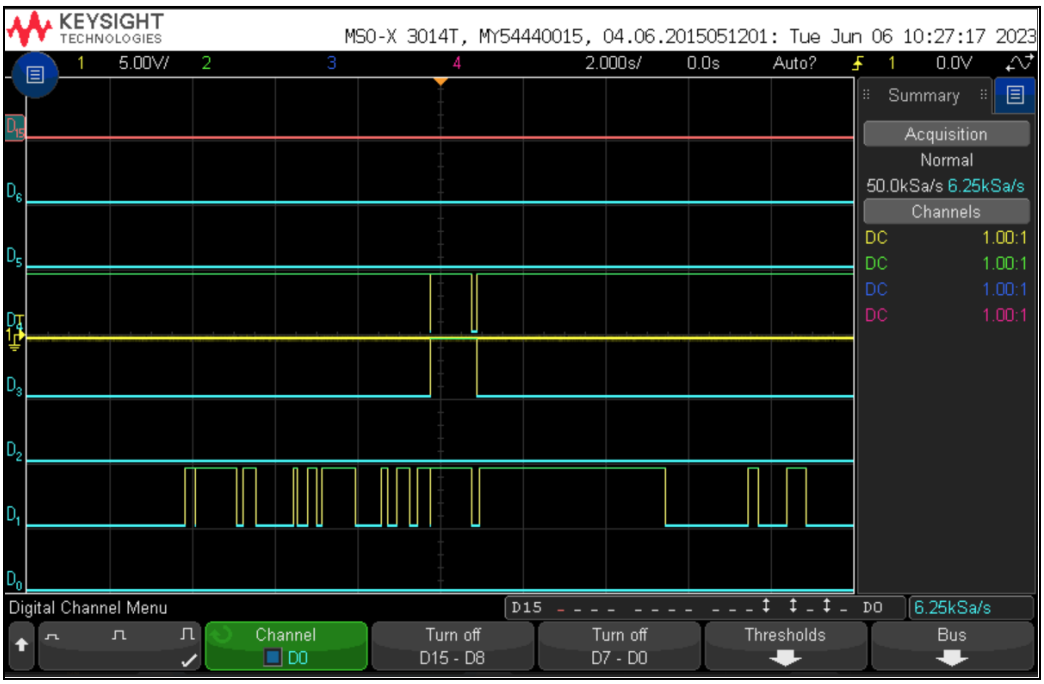
Comparator First Comparison Circuit. Parallel out's bits initialze the key when the FSM sends the set key signal. Depending on the note code from the MIDI (see table above) a specific NoteOn will be updated when the UpdateNote signal is sent. The update only occurs when NoteOnSig and Velocity are both true. Above, I show 1 of 24 update circuits in the chain.



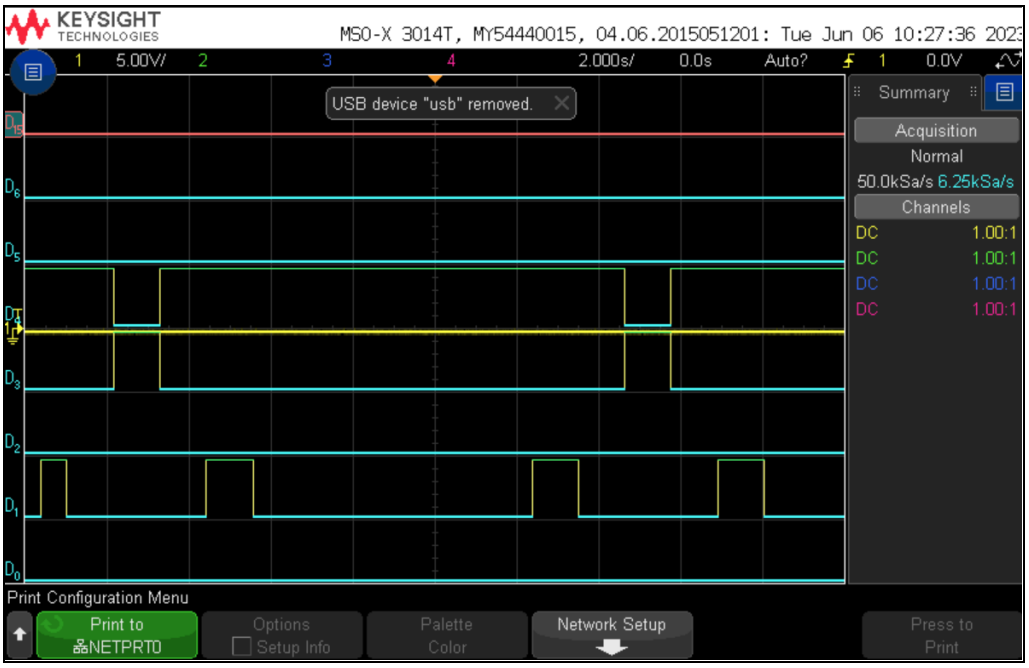
Key Comparator Second Comparison Circuit. Once the note from the MIDI is converted into a 24 bit vector, it is compared to the target note 24 bit vector to set the Match for that note. The match is only false if target note is on but the MIDI note is off. I remind the reader that we allow the reader to choose octaves although the top staff is for notes above 60 (middle C) while the bottom is for notes below.



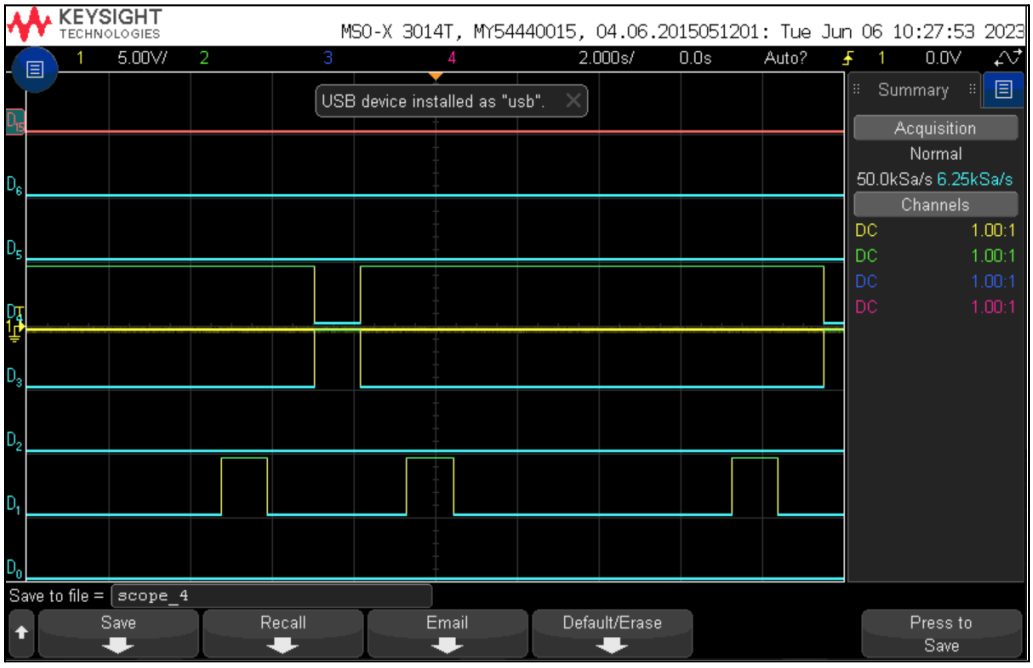
Key Comparator No Incoming Note Simulation. When there is no incoming note, KeyCompare remains in idle state.



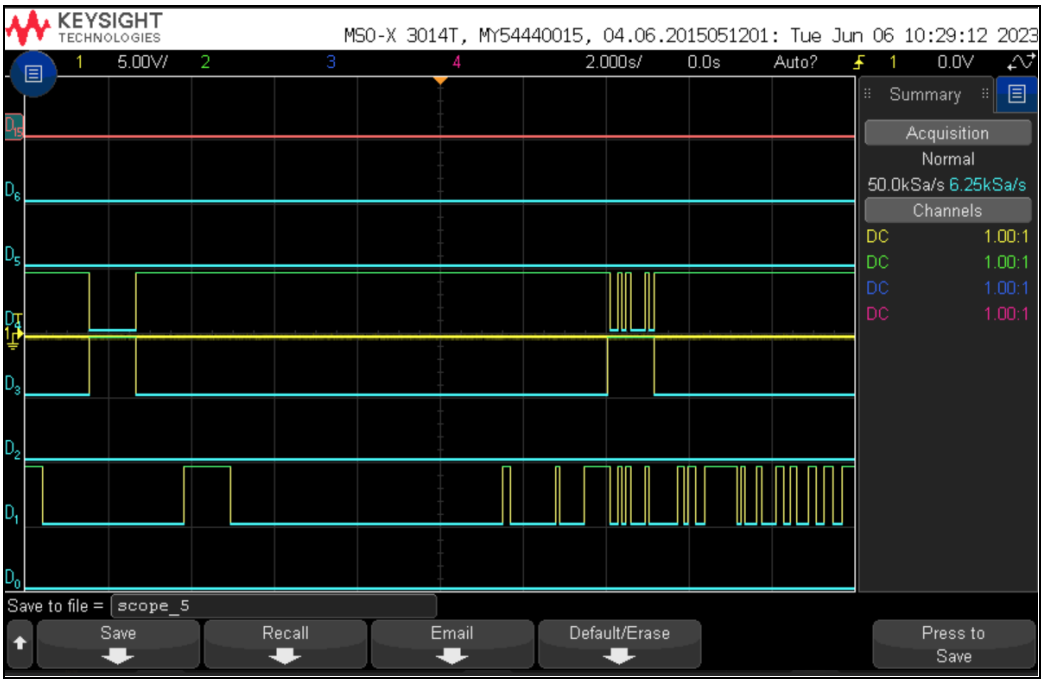
Key Comparator Oscilloscope Recordings (i). As it was difficult to simulate the KeyCompare Circuit, we turn to the scope. Note_Matches(23) is on D5, hit is on D3, and SCI_In is D1. The D1 corresponds to the user input from the Keytar that's shifted in 8 bits. The user input is compared against the notes in our memory. If the notes on the target match the notes on the notes in our memory, hit signal goes HIGH to show the score. In this screenshot, notes come in from the keyboard and one note is successfully hit.



Key Comparator Oscilloscope Recordings (ii). This recording shows the bug in which when the note is off the target but not off the screen, hit briefly goes high. This was expected from the code but an oversight in design. In future versions, we hope to use the Matches vector rather than hit and implement a color change per each note such that the note turns red and stays red until it leaves the screen.



Key Comparator Oscilloscope Recordings (iii). Same finding as recording ii.



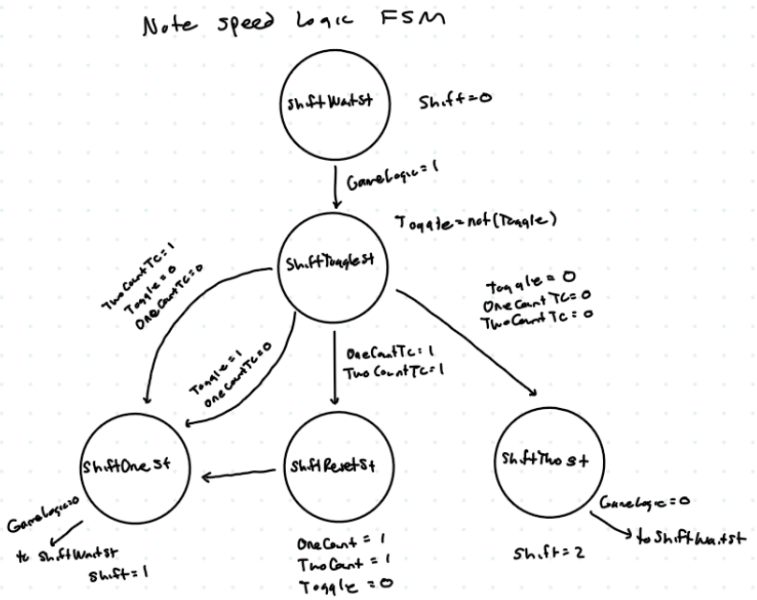
Key Comparator Oscilloscope Recordings (iv). A note is pressed as another slides over the target. However, the wrong note is pressed so matches (D5) does not go hight.

2.3h BeatsPerMinCircuit & ScoringSystem Written But Not Implemented

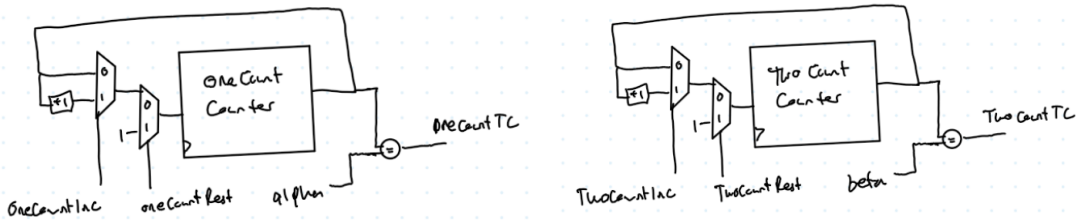
The following files were written and tested but not implemented into the final product due to project deadlines.

The BeatsPerMinCircuit took a user-defined beats per minute input and determined at every screen update whether to shift 4 or 8 pixels in order to obtain the exact beats per minute. For example, 60 beats per minute would be all 4 pixels shifts. 120 beats per minute would be all 8-pixel shifts. For speeds in between, the circuit would alternate between 4 and 8-pixel shifts, adding additional 4-pixel shifts to achieve the desired beats per minute.

Each note will represent a quarter note and 4-4 timing will be assumed. Our game will have a speed variable. Future work could allow users to tune speeds using a button input. We will set the speed to a constant 100 beats per minute but design a system that would work for variable speeds from 60 to 120 beats per minute. The beat per minute will affect the load and shift times for the NoteDisplay Circuit.



Finite State Machine for Shift Value Such that Beats Per Minute is as Desired. A finite state machine decides how fast to shift pixels based on the desired timing of the song.



Shift Value Counters. Count to the number of 4 and 8 pixel shifts necessary to achieve the beats per minute speed.

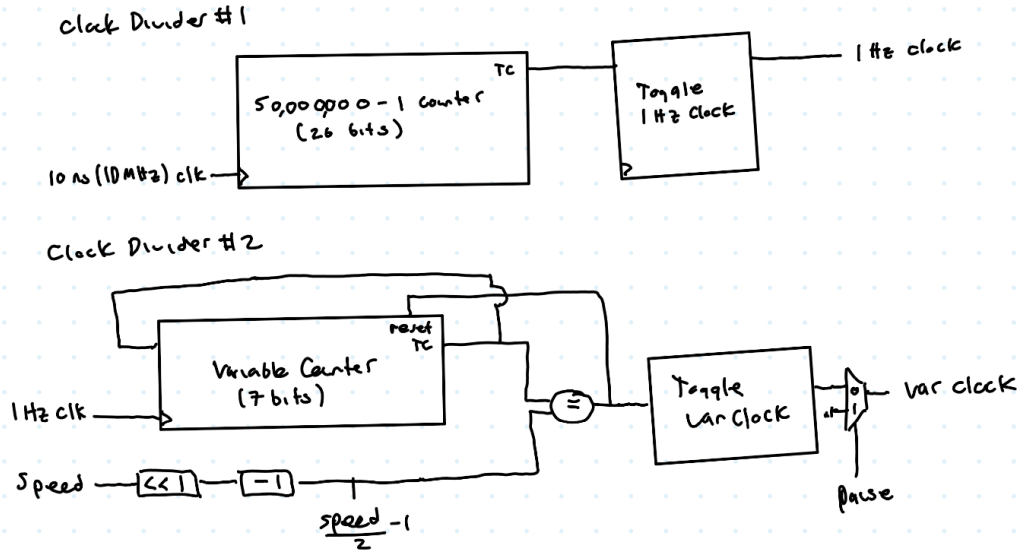
Notes will come from the right side of the screen and move to the left. As discussed above, note height will be 10 pixels. The width of the pixels must be divisible by 60 as pixels per second must be an integer.

$$\frac{X\ pixels}{1\ note} * \frac{4\ notes}{beat} * \frac{Y\ beats}{min} * \frac{1\ min}{60\ sec} = \text{integer number } \frac{pixels}{sec}$$

$$\frac{600\ pixel\ screen}{X\ pixel} = \text{integer number notes on the screen}$$

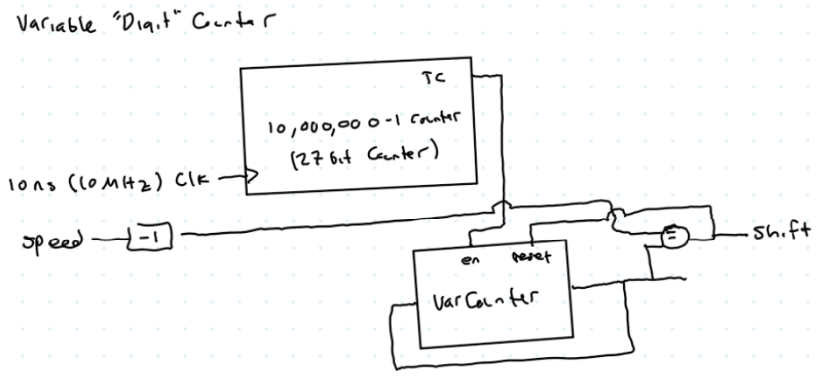
If the notes are X=60 pixels wide, both of these requirements are fulfilled and also the (pixels*min)/(beat*sec) ratio will be one, simplifying math below. For example, to achieve Y=100 beats per minute, the address would have to be incremented to achieve 400 increments

per second. Incrementing can only occur when the VGA is not reading the memory block. The shift register is moving at 25 Hz.

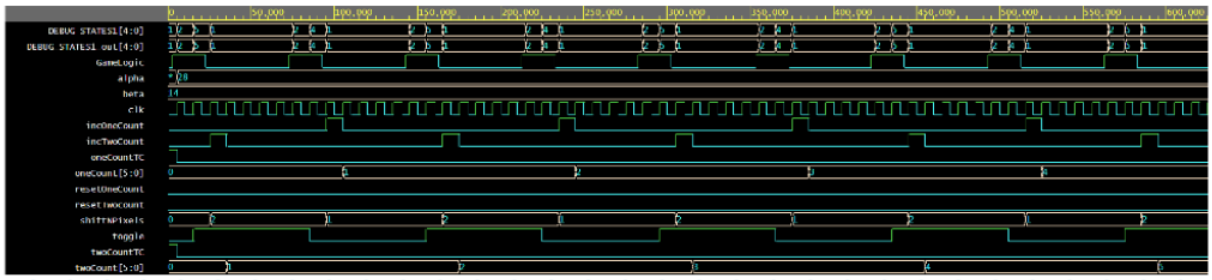


NoteDisplay Shifter Clock Speed. The NoteDisplay will be shifted every 60 to 120 Hz depending on Speed which represents beats per minute. The first divider will create a 1 Hz clock. The 1 Hz clock can easily be divided further into the 60 to 120 Hz clock. Speed will be shifted and decremented to obtain (speed/2)-1. Both counters will output to a toggle flop such that the duty cycle is 50%. The variable clock output will be gated by a mux, allowing for Pause.

Alternatively, the shifting could be performed with a counter instead of a clock:



NoteDisplay Shifter Counter. Instead of a divided clock, the counters cound count to M-1, setting shift high whenever the varCounter goes high.

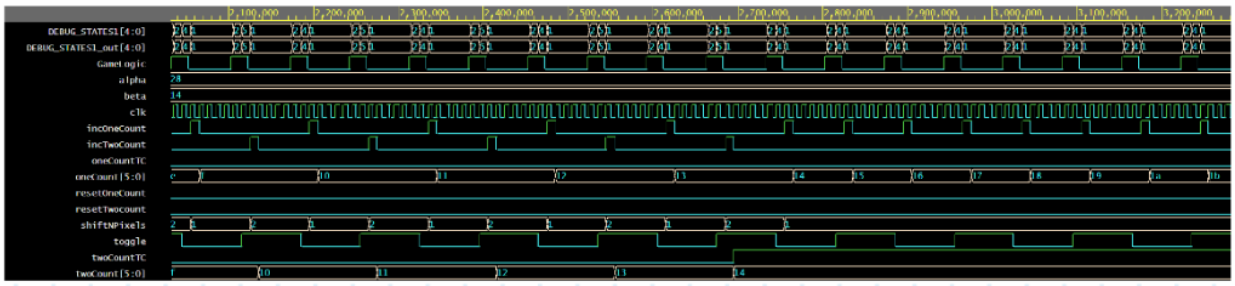


Whenever GameOn is high, the FSM goes to the toggle state and changes toggle toggle will allow notes to be incremented by 4 then 8 then 4 then 8 pixels at a time. This is seen as the oneCount and two count increment in states 5 and 4 respectively.

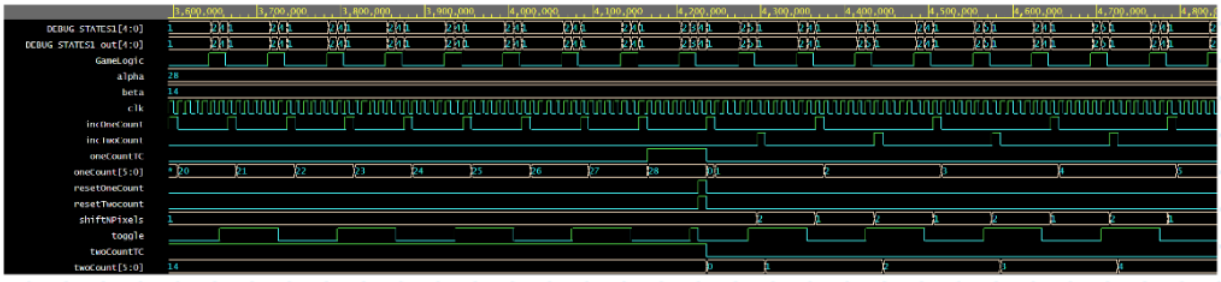
Alpha and beta are calculated by the NotesPerMinute, here shown as 80. There will be 40 4 pixel shifts and 20 8 pixel shifts for a total of 60 shifts and $80 \times 4 = 320$ pixels.

ShiftNPixels is an output telling the shift logic to shift either 4 or 8 pixels.

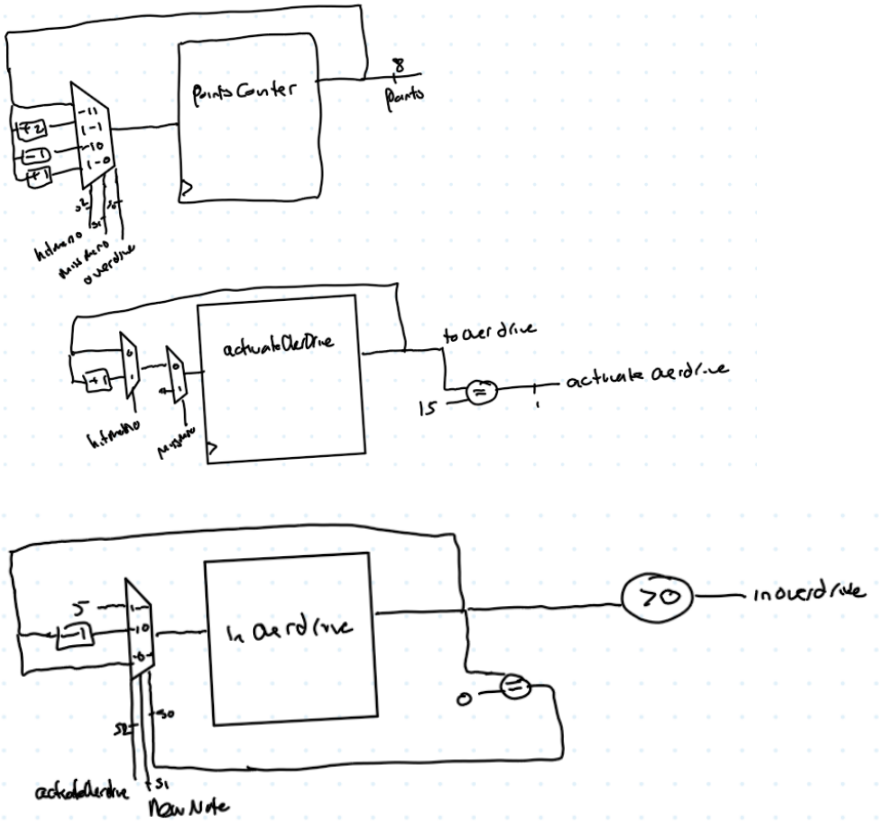
In the real circuit, GameOn will be much less frequent.



Once all 20 8 pixel shifts are finished, the counter continues shifting the 4 pixel shifts.



The ScoringSystem circuit received HIT and also MISS inputs from KeyCompare (mono-pulsed signals for hits and misses). Each hit incremented the score while misses decremented the score until zero was hit. In overdrive, hits incremented by two while users were not penalized for misses. A counter determined how long the user was in overdrive.

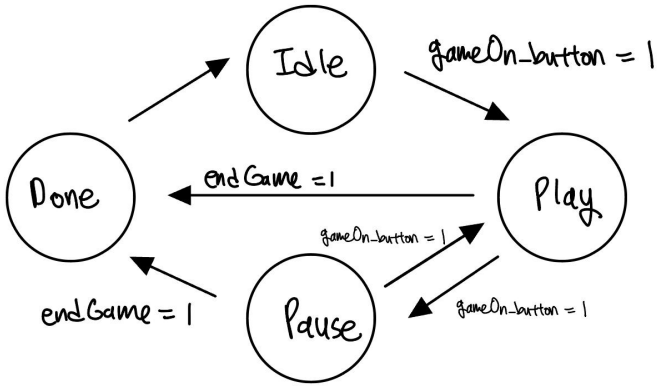


Scoring System MUX			
Hit Mono	**Miss Mono**	**Overdrive**	**Output**
-	1	1	hold points
1	-	1	+2 points
-	1	0	-1 points
1	-	0	+1 points

2.3i GameOn Logic

GameOn Logic implements a FSM that determines the state of the game (Play, Pause, Done), adjusts the speed, and displays the score. When the user presses the gameOn_button once, the

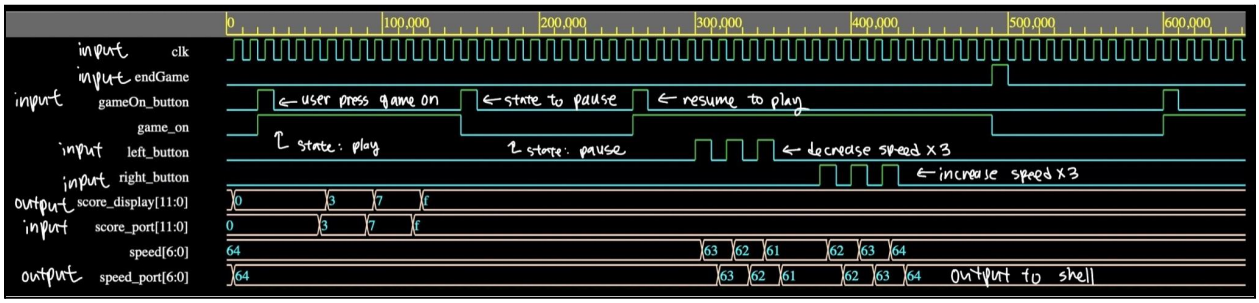
game is set to Play mode. Whenever the player presses the gameOn_button, the state toggles between Play and Pause, letting the player to pause and resume the game. The left and right button decrease and increase the speed of the game, decrementing and incrementing 1bpm at a time. Initially, the game is set at 100bpm, and the user can adjust the speed within the range of 60bpm and 120bpm. The score_port receives the current score from DetermineHit and outputs it to score_display, where it will be connected to the Seven Seg display.



GameOn Logic. A finite state machine, when gameOn_button is pressed, Idle state will transition to Play, toggling between Pause and Play with the press of the gameOn_button. When endGame signal is received, the game will move to the Done state, and return to the Idle state, waiting for the next press of the gameOn_button.

GameOn Logic Signals		
Signal	**Type**	**Notes**
clk	std_logic	Clock for the game
gameOn_button	std_logic_vector	Input play/pause button
game_on	std_logic	Play/(Idle, Pause, Stop) state
endGame	std_logic	Input to end game
left_button	std_logic	Reduce speed by 1bpm
right_button	std_logic	Increase speed by 1bpm
score_port	std_logic_vector(11 downto 0)	Input score from game
score_display	std_logic_vector(11 downto 0)	Output score to display

speed	unsigned(6 downto 0)	Speed variable
speed_port	std_logic_vector(6 downto 0)	Output speed to game

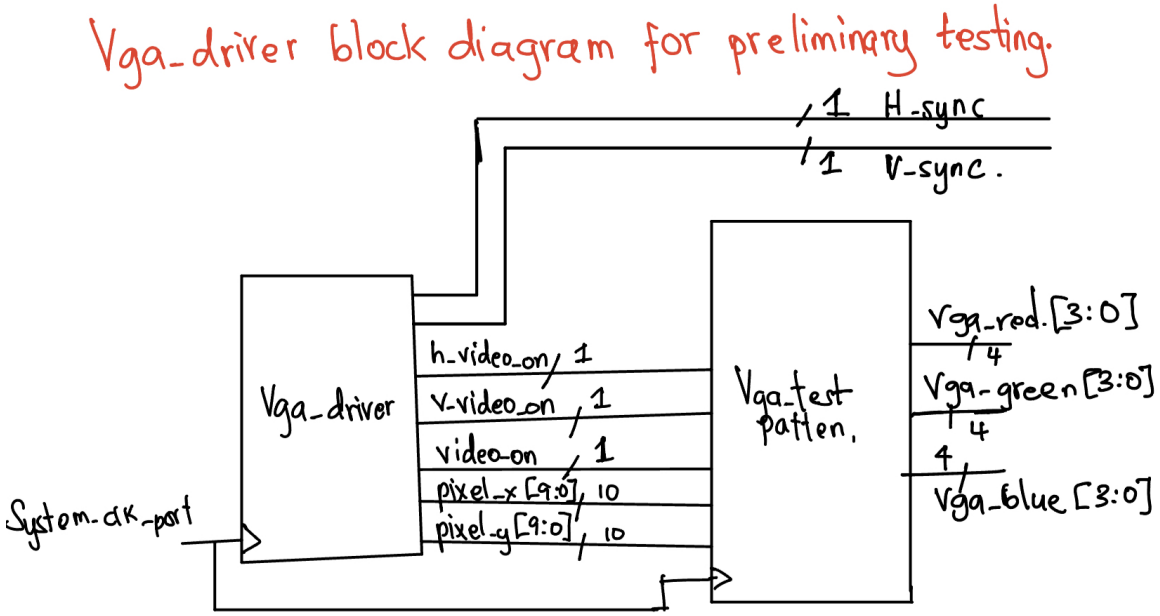


Simulation for GameOn. gameOn_button toggles between Play and Pause states, left_button decrements speed by 1bpm, right_button increments speed by 1bpm. endGame sets the game state to Done. score_display is connected to SevenSeg display.

2.4 VGA Display System

2.4a VGA Driver

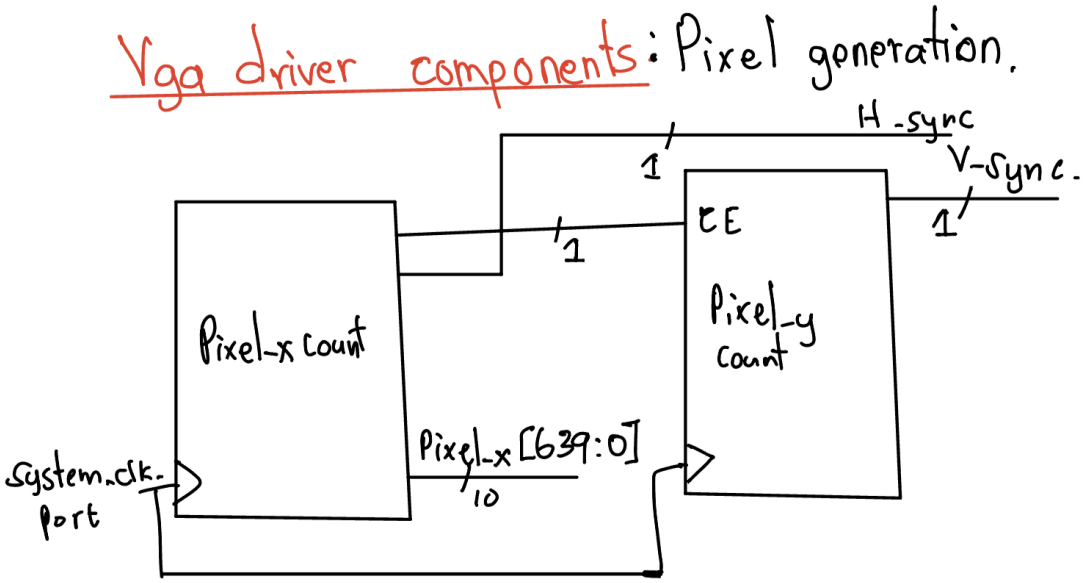
The vga driver generates the pixels on the screen such that every point can be represented by (x, y) coordinates on the screen. Also, it is used to determine when the screen is on and provide other helpful signals for the vga system. To implement this, the vga driver file counts to 800 for the pixel_x and 520 for pixel _ y. However, the actual coordinates for the pixel x and pixel y should be between 640 and 480 respectively. The difference to their respective maximum count values is used to account for retracing and determining when the horizontal and vertical videos are on and off.

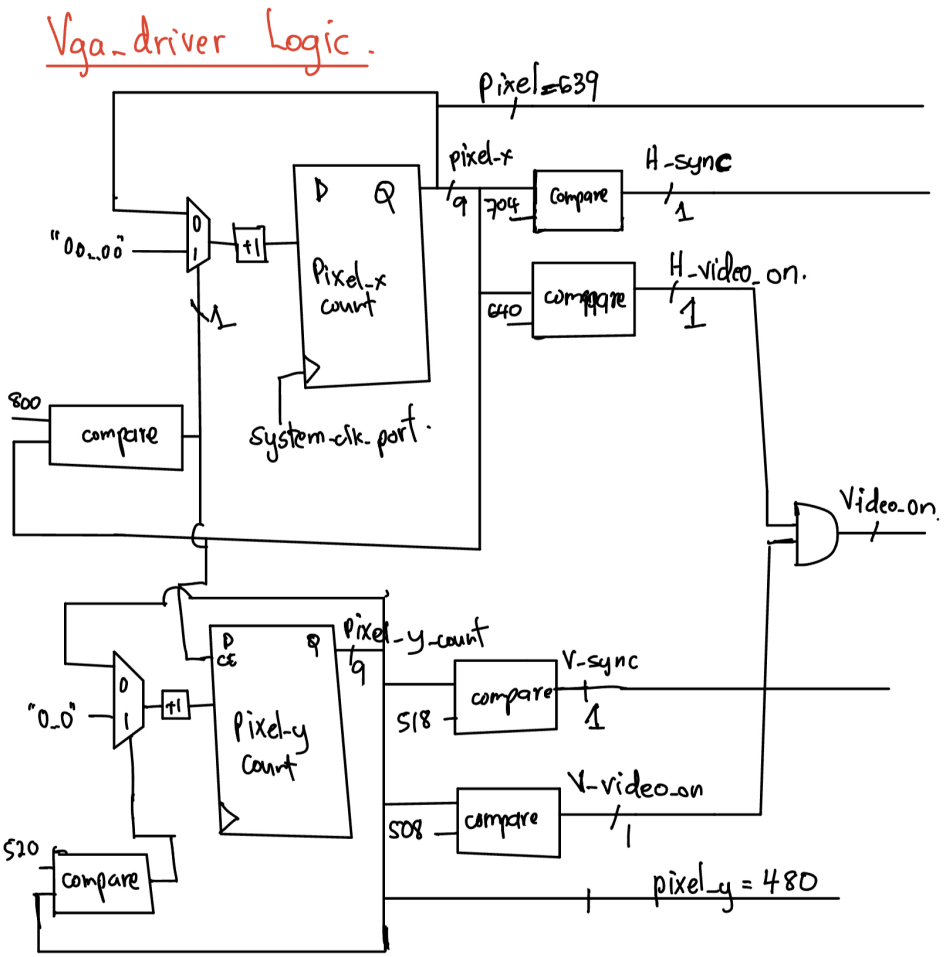


2.4b VGA Driver Entity Ports

a. System clock(in) : 25MHz clock
b. H_sync(out) : this is a horizontal synchronization signal that is HIGH for 704 counts and LOW for 96 counts of pixel x. When LOW, a new line is started on the screen.
c. V_sync(out) : this is a vertical synchronization signal that is HIGH for 518 counts and LOW for 2 counts of pixel y. When LOW, a new frame is displayed on the screen.
d. H_video_on(out) : 1-bit signal that is HIGH for 640 counts pixel x that's between 48 and 688, and LOW between 689 to 800.
e. V_video_on(out) : 1-bit signal that is HIGH for 480 counts of pixel y that's between 29 and 508, and LOW between 508 and 520.
f. Video_on(out) - this is a 1-bit signal that is high for when both H_video_on and V_video_on are on.
g. pixel_x : 10-bit vector signal that ranges from 0 to 639
h. Pixel_y: 10-bit vector signal that ranges from 0 to 479

2.4c Pixel Generation Process and Testing Procedures



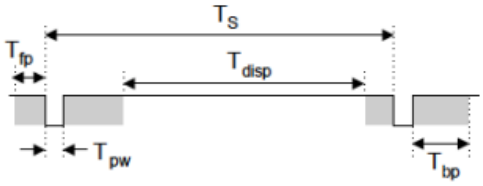


The vga driver file has two registers, pixel_xCount and pixel_yCount. These registers together with several logic operations provide the outputs. The pixel_xCount counts from 0 to 800 and it is incremented on every rising edge of the clock. There are 4 critical values for the timing of the vga system that's accounted for by the pixel_xCount that's the leftborder, horizontal display, right border and horizontal retrace. From our design, the leftborder ranges from 0 to 48, horizontal display ranges from 48 to 688, rightboder border ranges from 688 to 704 and horizontal retrace ranges from 704 to 800. If the pixel_xCount is between the leftborder and horizontal display, H_video_on is set HIGH, else it is LOW. The H_sync is set HIGH except for the horizontal retrace portion, and the horizontal display - 48 is mapped to the pixel_x out vector.

Similarly, the pixel_yCount accounts for 4 critical values for the timing for the vga system that's top border, vertical display, bottom border and vertical retrace. The pixel_yCount is controlled by the pixel_xCount. The pixel_yCount is incremented by 1 when pixel_xCount counts to 800. From our design, the topborder ranges from 0 to 29, vertical display ranges from 29 to 508, bottom border ranges from 508 to 518 and vertical retrace ranges from 704 to 800. If the pixel_yCount is between the top and horizontal display, H_video_on is set HIGH, else it is LOW. The B_sync is set HIGH except for the vertical retrace portion and the horizontal display - 29 is mapped to the pixel_x out vector.

The video_on output is set to the output of an AND-gates with the inputs H_video_on and V_video_on.

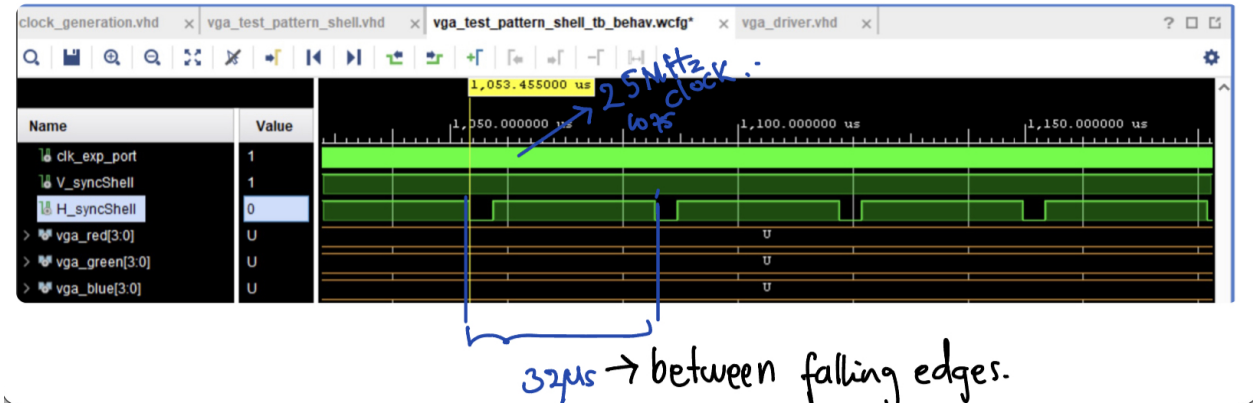
To test our vga driver file before implementing our project we simulated the vga driver, and tested it with the test pattern that was provided. Below are annotated screenshots of how we have tested that the timings for all the signals were accurate.

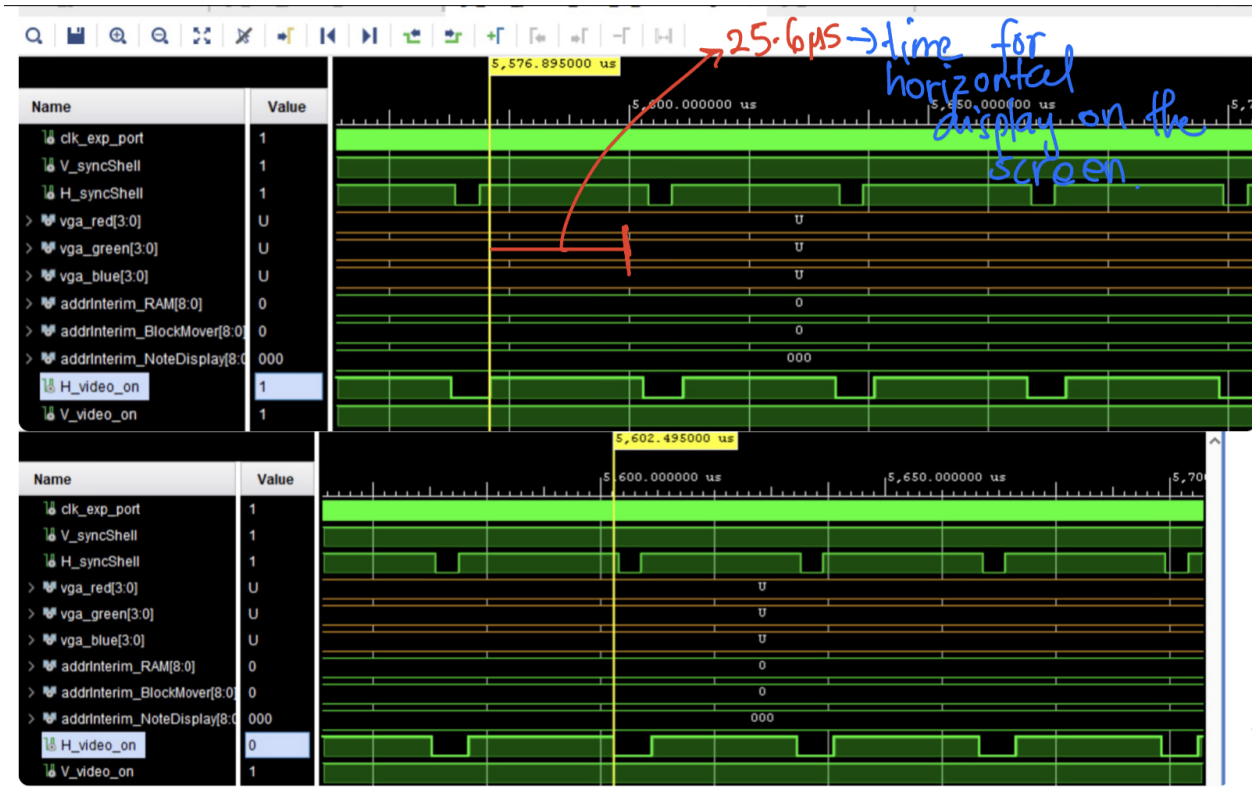


Symbol	Parameter	Vertical Sync			Horiz. Sync	
		Time	Clocks	Lines	Time	Clks
T_S	Sync pulse	16.7ms	416,800	521	32 us	800
T_{disp}	Display time	15.36ms	384,000	480	25.6 us	640
T_{pw}	Pulse width	64 us	1,600	2	3.84 us	96
T_{fp}	Front porch	320 us	8,000	10	640 ns	16
T_{bp}	Back porch	928 us	23,200	29	1.92 us	48

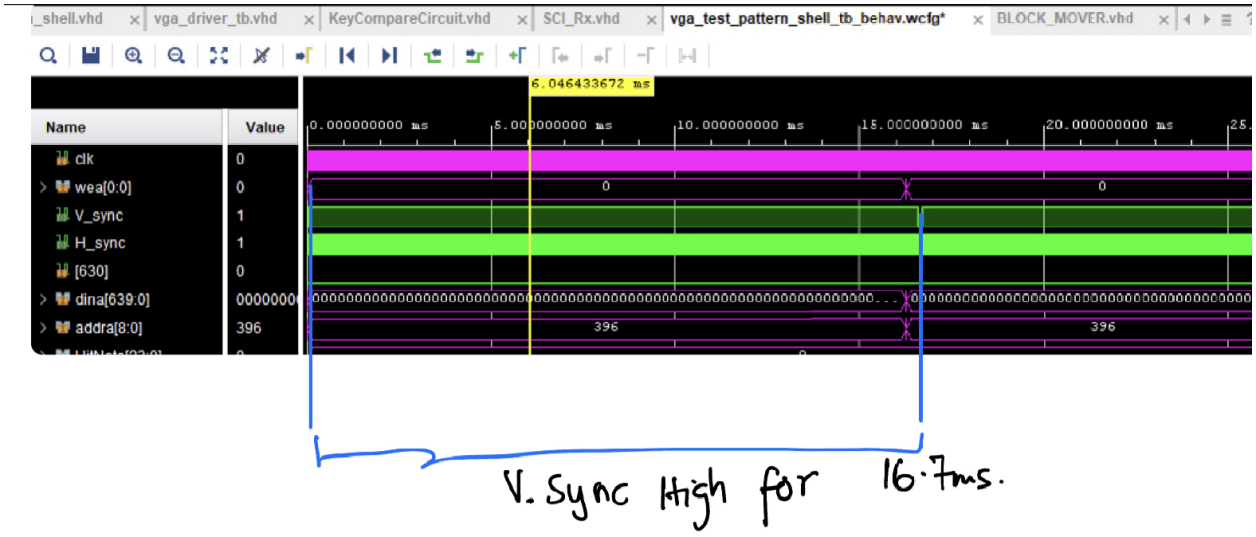
Figure 19: VGA system timings for 640x480 display

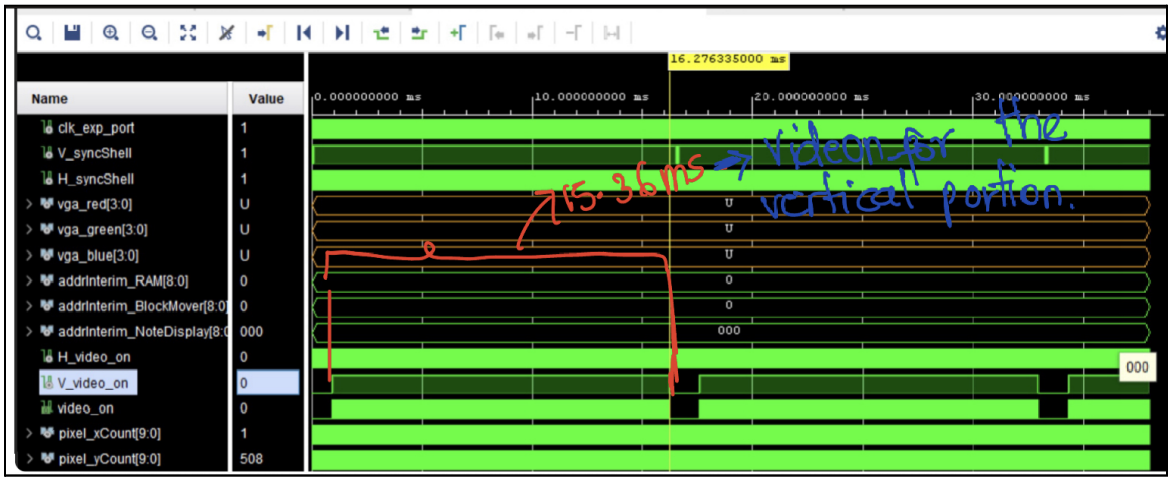
2.4d Horizontal Sync Timing





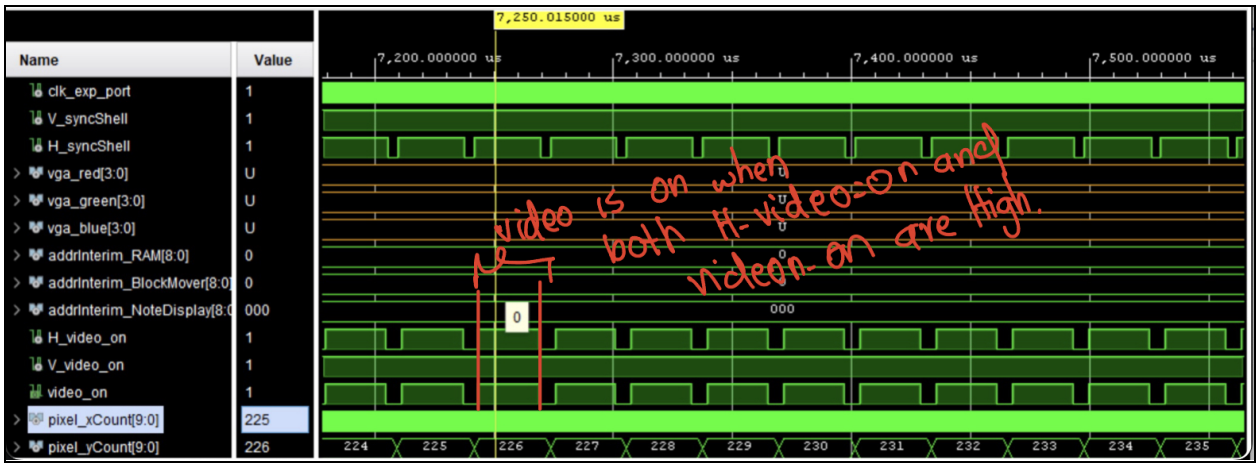
2.4e Vertical Sync Timing



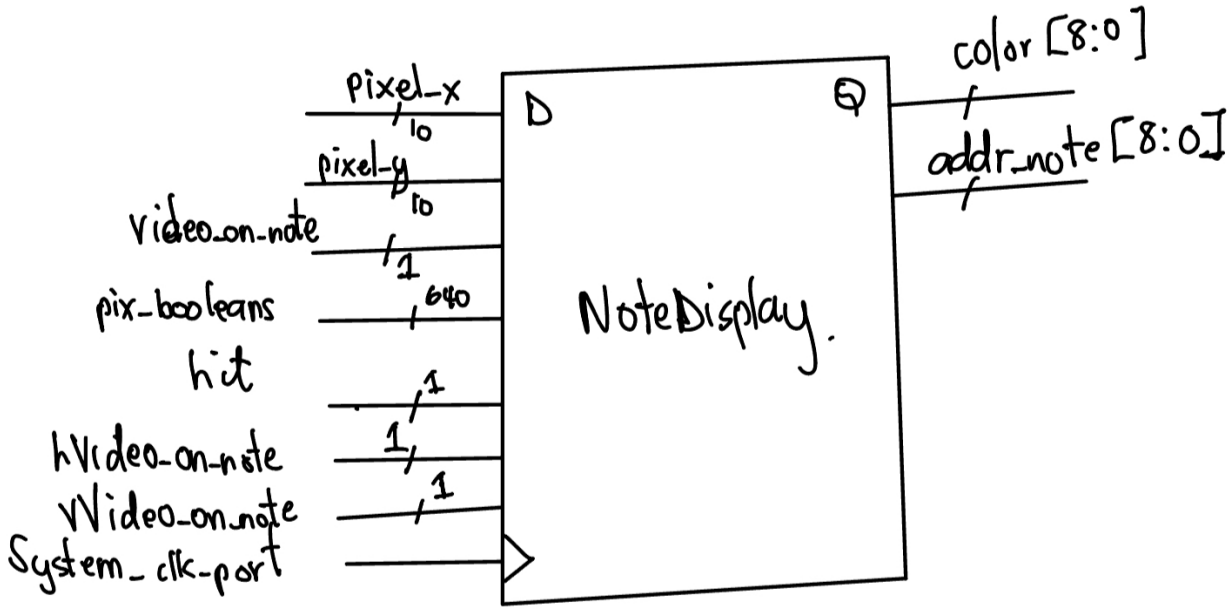


2.4f Video On Logic

As shown from the simulations above, the found timing of the vga signals were as expected. To implement it on the screen. We wired the signals to the FPGA that was interfacing with the screen. To generate different colors, we referenced different coordinates on the screen to set their colors using constant 8-bit vectors. Since the vga expects RGB values, we divided the 8-bit vectors such that the first 3-bits correspond to RED, the next 3-bits correspond to GREEN and remaining 2-bits correspond to BLUE. The RED, GREEN, BLUE are 4-bits values with leading zeros that are wired to the FPGA board.



2.4g Note Display Logic



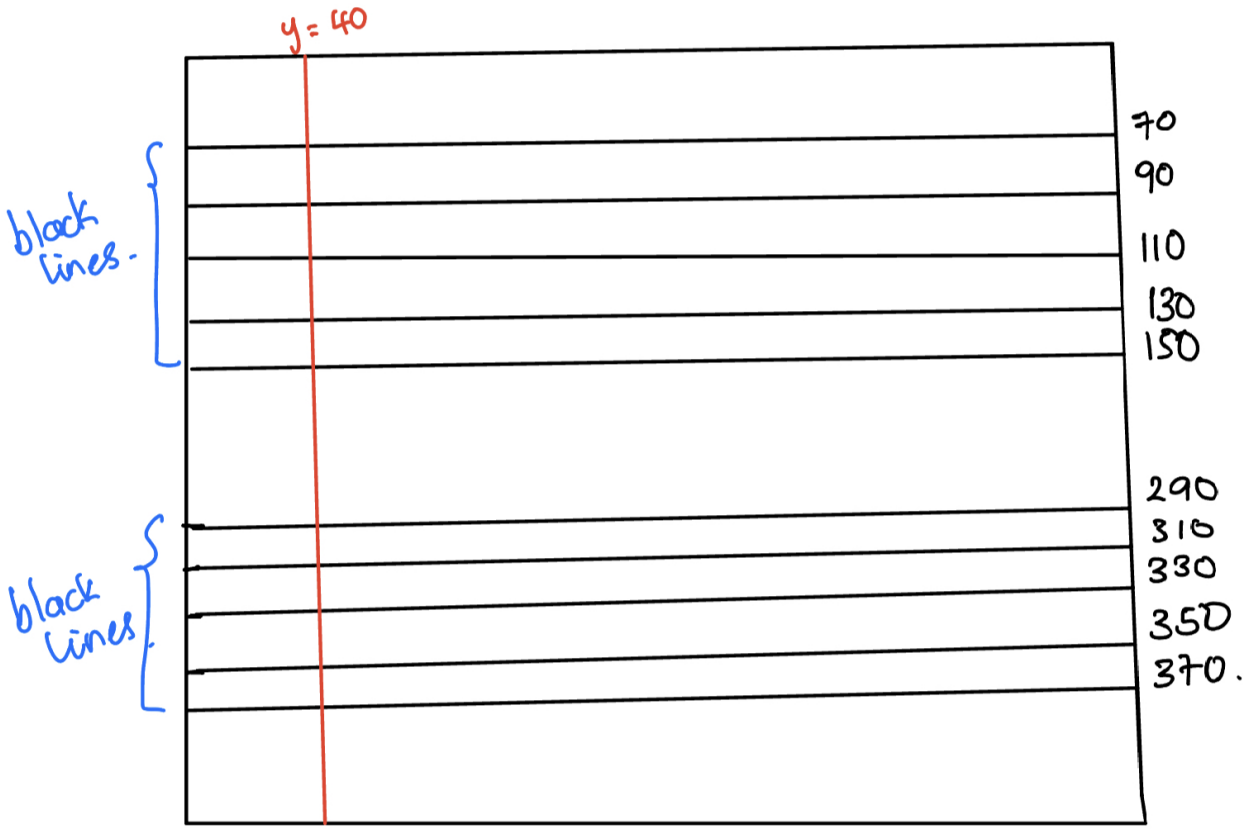
Entity ports description

a. Clk(in) - 25MHz clock
b. Hit - 1-bit signal that's HIGH when the user has pressed the correct note.
c. video_on_note - 1-bit signal from the vga driver.
d. hVideo_on_note - 1-bit signal corresponding to the output from the vga driver.
e. vVideo_on_note - 1-bit signal corresponding to the output from the vga driver.
f. pixelX and pixelY - 10-bit vectors that correspond to the coordinates on the screen.
g. Color - 8-bit vector for the color of every spot on the screen.
h. addr_note - 8-bit vector that corresponds to the pixelY representing every row on the screen. This address is sent to a block of memory that provides the pixel booleans.

i. pix_boolean - 640-bit vectors that come from a block memory. This 640 bits corresponds to every pixel on the current row only that it is a representation of 1's and 0's.

The NoteDisplay has two aspects, a static image that shows the rows on which the notes run across on the screen and a dynamic image whose color is determined by the combinations of the hit and pix_boolean values.

To get the static images, we referenced the pixel_y and it is equal to the values shown on the diagram below, their color was set to black. To get the target line, pixel_y is set 40, and reset is set to pixel_y. This is all done when the video_on_note signal is HIGH and on the rising_edge of the clock.



The dynamic images were determined by the pix_boolean and the hit signals. We referenced each value the pix_boolean using the pixelX signals, and if it pixel_x values is above 40, hit signal is HIGH/LOW and the pix_boolean is HIGH, we set the color to blue to signify that there is a note at the spot. If the pixel_boolean is HIGH, pixel_x values is 40 and the hit signal is LOW, we set the color to red because it is a mess else if the the pix_boolean is HIGH, the value of pixel is below 40 and the hit signal is HIGH, the color is set to green it is a hit.

The screen is updated after every 800 clock cycles, hence we requested new pix_boolean values from the pix_booleans memory block when hVideo_on_note is LOW, and vVideo_on_note is HIGH. This ensures that there is no continuous reading from the pix_booleans_mem block as we update the current row on the screen. To fetch the pix_booleans input from the pix_booleans_mem block we send addr_note which corresponds to the current row on the screen and pix_boolean_mem address.

2.5 A Constraint File Connects the Interface to the MIDI and VGA

A constraint file was used to connect the MIDI input and VGA display to the shell. Throughout the testing procedures the debug outputs were sent to the JA and JB pins for oscilloscope probing.

Constraint File Connections		
Package Pin	**Variable**	**Notes**
W5	clk_exp_port	System clock.
A14	SCI_in	MIDI Rx signal.
G19	vga_red[0]	Four VGA red bits.
H19	vga_red[1]	
J19	vga_red[2]	
N19	vga_red[3]	
N18	vga_blue[0]	Four VGA blue bits.
L18	vga_blue[1]	
K18	vga_blue[2]	
J18	vga_blue[3]	
J17	vga_green[0]	Four VGA green bits
H17	vga_green[1]	
G17	vga_green[2]	
D17	vga_green[3]	
P19	H_syncShell	VGA horizontal sync signal.
R19	V_syncShell	VGA vertical sync signal

2.6 Clock generation Process

Entity port descriptions:

1. clk_ex_port(in) : std_logic bit that is connected to FPGA to supply the 100MHz.
2. system_clk_port(out) : std_logc bit of the stepped down clock.

For some of our processes we needed a 25MHz clock such as the vga_driver and the NotDisplay. To step-down the FPGA 100MHz's, we toggled the system_clk_port for every two rising_edges of the 100MHz clock that's for every 2 terminal counts.

2.7 Top Shell

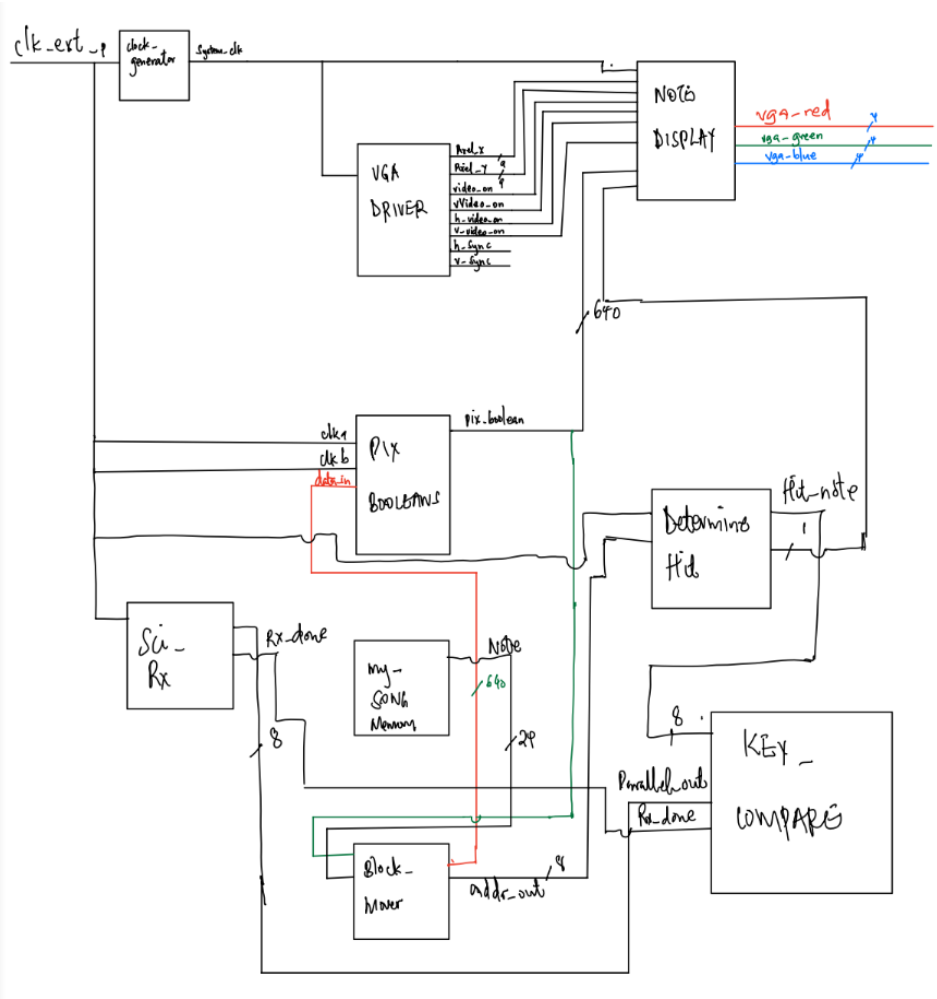
With all the descriptions provided for the components mentioned above, the top shell serves as the central element that brings everything together.

The objective of the game is to play song notes and display them. The first step involves finding a way to load the song so that we have notes to display. This is achieved through the blk_mem_gen_0 memory block. This memory block operates on an external clock and requires a 7-bit address to produce a 24-bit output representing the next note to be displayed. Similarly, the pix_booleans undergo a similar process. These booleans indicate whether there is a note to display on a specific pixel (1) or not (0).

The VGA team needs to read from this memory to determine whether to display the background or a note for each pixel. This reading is performed once for every row of pixels. To enable this, the VGA team requires a driver and a note display file. The driver ensures that the display moves to the next pixel at a frequency of 25 MHz. Then, the VGA team sends the appropriate color combination to this driver along with the hsync and vsync signals, resulting in the image being displayed on the screen. The note display file ensures that the correct color combination is sent for each pixel. Detecting incorrect notes requires knowledge of the MIDI note that has been pressed. The MIDI team sets up a communication system between the MIDI and the FPGA using an SPI receiver. A lot of SPI receiver details have already been discussed in class.

Once communication with the MIDI is established and the logic for displaying the note color is implemented, an additional logic is required to modify the memory responsible for storing the note locations. Since the displayed notes are dynamic, the VGA needs to be aware of the note positions stored in the memory block. The pix_booleans memory block is edited based on the content of the note storage memory block. For example, if note A is being displayed, the game logic identifies the corresponding entry in the note storage memory and inserts 1s into the pix_booleans memory block accordingly. After insertion, the booleans are shifted at the appropriate frequency to ensure they are visible on the screen. This shifting process is facilitated by the block mover, and the signals required for each component have already been described above. The determined hit is present to let the VGA team know if the note is hit correctly or not.

The diagram below strikes the most important points to help explain this further:



2.8 Bugs and Next Steps

The following next steps could be implemented:

1. Fix the bug in which notes flash green if the note on the target is off the target but on the screen. USe the matches vector to change the color for individual notes rather than changing all to green or all to red.
2. Implement the game start/pause.
3. Implement the scoring and overdrive system.
4. Implement more songs using the MATLAB script (see Appendix).

III References

Prof Hansen's Guide to Midi: Digital Electronics (SP23). https://dartmouth.instructure.com/courses/58308/pages/prof-hansens-guide-to-midi. Accessed 17 May 2023.

MIDI Tutorial - SparkFun Learn. https://learn.sparkfun.com/tutorials/midi-tutorial/all. Accessed 17 May 2023.

Yamaha SHS-200 Owner's Manual (Page 38 of 41) | ManualsLib.

https://www.manualslib.com/manual/196428/Yamaha-Shs-200.html?page=38#manual.

Accessed 17 May 2023.

MIDI Note Numbers and Center Frequencies | Inspired Acoustics.

https://www.inspiredacoustics.com/en/MIDI_note_numbers_and_center_frequencies.

Accessed 17 May 2023.

IV Appendix

1.1 Original Static Staff MATLAB Design

This is the MATLAB code which we based our original staff design:

```
close all; clear all;

% first set the screen to white
VGA_Display = ones(480, 640);  % [255, 255, 255]

% shade the hit region
VGA_Display(:, 40:110) = 0.8; % [204, 204, 204]

% bars for the left hand
VGA_Display(80, :)  = 0; % [0, 0, 0]
VGA_Display(100, :) = 0; % [0, 0, 0]
VGA_Display(120, :) = 0; % [0, 0, 0]
VGA_Display(140, :) = 0; % [0, 0, 0]
VGA_Display(160, :) = 0; % [0, 0, 0]

% bars for the right hand
VGA_Display(300, :) = 0; % [0, 0, 0]
VGA_Display(320, :) = 0; % [0, 0, 0]
VGA_Display(340, :) = 0; % [0, 0, 0]
VGA_Display(360, :) = 0; % [0, 0, 0]
VGA_Display(380, :) = 0; % [0, 0, 0]

% rectangular targets
VGA_Display(1:10, 40:110)    = 0;  % [0, 0, 0]
```

```
VGA_Display(220:235, 40:110) = 0;  % [0, 0, 0]
VGA_Display(470:480, 40:110) = 0;  % [0, 0, 0]

% save and show figure
figure(); imshow(VGA_Display);
saveas(gcf, 'StaffPattern.jpeg')
```

1.2 MATLAB COE Generator

This code generates the Coe files from .mid files:

```
close all; clear all;
readme = fopen('CmajorScale.mid');
[readOut, byteCount] = fread(readme);
fclose(readme);
% Concatenate ticksPerQNote from 2 bytes
ticksPerQNote = polyval(readOut(13:14),256);
% Initialize values
chunkIndex = 14;     % Header chunk is always 14 bytes
ts = 0;              % Timestamp - Starts at zero
BPM = 120;
msgArray = [];
myNoteArray = [];
myVelocityArray = [];
myTimeStamps = [];
% Parse track chunks in outer loop
while chunkIndex < byteCount

   % Read header of track chunk, find chunk length
   % Add 8 to chunk length to account for track chunk header length
   chunkLength = polyval(readOut(chunkIndex+(5:8)),256)+8;

   ptr = 8+chunkIndex;             % Determine start for MIDI event parsing
   statusByte = -1;                % Initialize statusByte. Used for running status
support

   % Parse MIDI track events in inner loop
   while ptr < chunkIndex+chunkLength
       % Read delta-time
       [deltaTime,deltaLen] = findVariableLength(ptr,readOut);
       % Push pointer to beginning of MIDI message
       ptr = ptr+deltaLen;

       % Read MIDI message
       [statusByte,messageLen,message] = interpretMessage(statusByte,ptr,readOut);
       % Extract relevant data - Create midimsg object
       [ts,msg] = createMessage(message,ts,deltaTime,ticksPerQNote,BPM);

       % Add midimsg to msgArray
       msgArray = [msgArray;msg];
       % grab the note
       if ~isempty(msg) & msg.Type ~= 7
```

```
            myNoteArray = [myNoteArray; msg.Note];
            myVelocityArray = [myVelocityArray;msg.Velocity];
            myTimeStamps = [myTimeStamps;msg.Timestamp];
        end
        % Push pointer to next MIDI message
        ptr = ptr+messageLen;
    end

    % Push chunkIndex to next track chunk
    chunkIndex = chunkIndex+chunkLength;
end
disp(msgArray)
endTime   = ceil(max(myTimeStamps));        % get the max time in the file
timeSteps = floor(max(myTimeStamps)/0.25);  % get the number of time points
timeArray = 0:0.25:max(myTimeStamps);       % generate a time array
% initialize COE file output with the first note
NoteOn = zeros(1, 24);
CoeOut = [NoteOn];
i = 1;
for t = timeArray                   % for each timestep
    while i<= length(myTimeStamps) & myTimeStamps(i) <= t      % while the timestep
for the note is less than the time step of the counter
        if myVelocityArray(i) == 0
            % note off case
            switch myNoteArray(i)
                case {24, 36, 48}
                            NoteOn(18+1) = 0;
                    case { 26, 38, 50}
                            NoteOn(17+1) = 0;
                    case {28, 40, 52}
                        NoteOn(23+1) = 0;
                        NoteOn(16+1) = 0;
                    case{29, 41, 53}
                                NoteOn(22+1) = 0;
                        NoteOn(15+1) = 0;
                    case{31, 43, 55}
                                NoteOn(21+1) = 0;
                        NoteOn(14+1) = 0;
                    case{21, 33, 45, 57}
                                NoteOn(20+1) = 0;
                        NoteOn(13+1) = 0;
                    case{23, 35, 47, 59}
                                NoteOn(19+1) = 0;
                        NoteOn(12+1) = 0;
                    case{60, 72, 84, 96, 108, 120}
                            NoteOn(4+1) = 0;
                    case{62, 74, 86, 98, 110, 122}
                            NoteOn(10+1) = 0;
                        NoteOn(3+1) = 0;
                    case{64, 76, 88, 100, 112, 124}
                                NoteOn(2+1) = 0;
                        NoteOn(9+1) = 0;
                    case{65, 77, 89, 101, 113, 125}
                                NoteOn(1+1) = 0;
                        NoteOn(8+1) = 0;
```

```
                case{67, 79, 91, 103, 115, 127}
                            NoteOn(0+1) = 0;
                    NoteOn(7+1) = 0;
                case{69, 81, 93, 105, 117}
                            NoteOn(6+1) = 0;
                case{71, 83, 95, 107, 119}
                            NoteOn(5+1) = 0;
            otherwise
        end
    else
        % note on case
         switch myNoteArray(i)
            case {24, 36, 48}
                        NoteOn(18+1) = 1;
                case { 26, 38, 50}
                       NoteOn(17+1) = 1;
                case {28, 40, 52}
                    NoteOn(23+1) = 1;
                    NoteOn(16+1) = 1;
                case{29, 41, 53}
                            NoteOn(22+1) = 1;
                    NoteOn(15+1) = 1;
                case{31, 43, 55}
                            NoteOn(21+1) = 1;
                    NoteOn(14+1) = 1;
                case{21, 33, 45, 57}
                            NoteOn(20+1) = 1;
                    NoteOn(13+1) = 1;
                case{23, 35, 47, 59}
                            NoteOn(19+1) = 1;
                    NoteOn(12+1) = 1;
                case{60, 72, 84, 96, 108, 120}
                       NoteOn(4+1) = 1;
                case{62, 74, 86, 98, 110, 122}
                       NoteOn(10+1) = 1;
                    NoteOn(3+1) = 1;
                case{64, 76, 88, 100, 112, 124}
                            NoteOn(2+1) = 1;
                    NoteOn(9+1) = 1;
                case{65, 77, 89, 101, 113, 125}
                            NoteOn(1+1) = 1;
                    NoteOn(8+1) = 1;
                case{67, 79, 91, 103, 115, 127}
                            NoteOn(0+1) = 1;
                    NoteOn(7+1) = 1;
                case{69, 81, 93, 105, 117}
                            NoteOn(6+1) = 1;
                case{71, 83, 95, 107, 119}
                            NoteOn(5+1) = 1;
            otherwise
         end
    end
    i = i+1;
end
CoeOut = [CoeOut;NoteOn];
```

```
End
% repeat X times
CoeOut = repelem(CoeOut, 1,3);
% Convert the vector elements to strings
strVector = num2str(CoeOut.');
% Remove leading and trailing whitespaces
strVector = strtrim(strVector);
% Join the elements with a comma separator
resultString = strjoin(cellstr(strVector), ', ');
disp(resultString);
% Open the file in write mode
fileID = fopen('output.txt', 'w');
% Convert the vector elements to strings
strVector = num2str(CoeOut.');
% Remove leading and trailing whitespaces
strVector = strtrim(strVector);
% Split the elements into a cell array
cellVector = cellstr(strVector);
fprintf(fileID, '%s\n', "memory_initialization_radix=2;");
fprintf(fileID, '%s\n', "memory_initialization_vector=");
% Print each element on a new line with a comma separator
for i = 1:numel(cellVector)
   fprintf(fileID, '%s,\n', cellVector{i});
end
fprintf(fileID, '%s\n', ";");
% Close the file
fclose(fileID);
function [valueOut,byteLength] = findVariableLength(lengthIndex,readOut)
byteStream = zeros(4,1);
for i = 1:4
   valCheck = readOut(lengthIndex+i);
   byteStream(i) = bitand(valCheck,127);   % Mask MSB for value
   if ~bitand(valCheck,uint32(128))        % If MSB is 0, no need to append further
       break
   end
end
valueOut = polyval(byteStream(1:i),128);    % Base is 128 because 7 bits are used for
value
byteLength = i;
end
function [statusOut,lenOut,message] = interpretMessage(statusIn,eventIn,readOut)
% Check if running status
introValue = readOut(eventIn+1);
if isStatusByte(introValue)
   statusOut = introValue;         % New status
   running = false;
else
   statusOut = statusIn;           % Running status—Keep old status
   running = true;
end
switch statusOut
   case 255     % Meta-event (FF)—IGNORE
       [eventLength, lengthLen] = findVariableLength(eventIn+2, ...
           readOut);   % Meta-events have an extra byte for type of meta-event
       lenOut = 2+lengthLen+eventLength;
```

```
        message = -1;
    case 240     % Sysex message (F0)—IGNORE
        [eventLength, lengthLen] = findVariableLength(eventIn+1, ...
            readOut);
        lenOut = 1+lengthLen+eventLength;
        message = -1;

    case 247     % Sysex message (F7)—IGNORE
        [eventLength, lengthLen] = findVariableLength(eventIn+1, ...
            readOut);
        lenOut = 1+lengthLen+eventLength;
        message = -1;
    otherwise    % MIDI message—READ
        eventLength = msgnbytes(statusOut);
        if running
            % Running msgs don't retransmit status—Drop a bit
            lenOut = eventLength-1;
            message = uint8([statusOut;readOut(eventIn+(1:lenOut))]);

        else
            lenOut = eventLength;
            message = uint8(readOut(eventIn+(1:lenOut)));
        end
end
end
% ----
function n = msgnbytes(statusByte)
if statusByte <= 191        % hex2dec('BF')
   n = 3;
elseif statusByte <= 223    % hex2dec('DF')
   n = 2;
elseif statusByte <= 239    % hex2dec('EF')
   n = 3;
elseif statusByte == 240    % hex2dec('F0')
   n = 1;
elseif statusByte == 241    % hex2dec('F1')
   n = 2;
elseif statusByte == 242    % hex2dec('F2')
   n = 3;
elseif statusByte <= 243    % hex2dec('F3')
   n = 2;
else
   n = 1;
end
end
% ----
function yes = isStatusByte(b)
yes = b > 127;
end
function [tsOut,msgOut] =
createMessage(messageIn,tsIn,deltaTimeIn,ticksPerQNoteIn,bpmIn)
if messageIn < 0     % Ignore Sysex message/meta-event data
   tsOut = tsIn;
   msgOut = midimsg(0);
   return
```

```
end
% Create RawBytes field
messageLength = length(messageIn);
zeroAppend = zeros(8-messageLength,1);
bytesIn = transpose([messageIn;zeroAppend]);
% deltaTimeIn and ticksPerQNoteIn are both uints
% Recast both values as doubles
d = double(deltaTimeIn);
t = double(ticksPerQNoteIn);
% Create Timestamp field and tsOut
msPerQNote = 6e7/bpmIn;
timeAdd = d*(msPerQNote/t)/1e6;
tsOut = tsIn+timeAdd;
% Create midimsg object
midiStruct = struct('RawBytes',bytesIn,'Timestamp',tsOut);
msgOut = midimsg.fromStruct(midiStruct);
end
```