

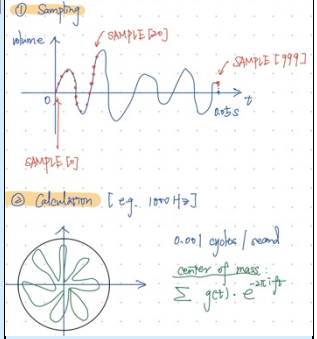

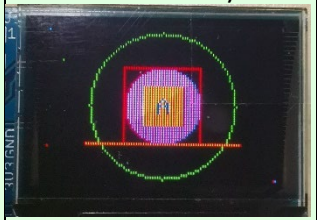





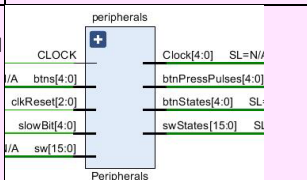


Student A	Student B	Lab Group
Liu Jingming A0204685B	Li Bozhao A0205636H	Thursday AM 04X Group 01

Feature	Marks For	Input Devices	Feature Description	Images / Photos
Real-time audio volume indicator	Student A	SW15	The audio-visual indicator takes in real-time surrounding volumes and displays the mapped volume levels (0~15) on the LEDs and 7 segments. SW15 is 0: mic_in shown on 12 LEDs SW15 is 1: Peak intensity shown on 16 LEDs. The peak intensity drops slowly down instead of dropping instantly even when the volume suddenly goes zero. This is achieved by using a floating cap of maxima.	
Discrete Fast Fourier Transformation	Student A	[11:0] mic_in	This function uses discrete Fast Fourier Transform (DFT) to processes surrounding sound at a period of 0.05 seconds. Its output is the frequency of main sound signal during the time period. Led[16:14] shows the closest frequency of the main signal: 001: 1000Hz 011: 2000Hz 111: 3000Hz (For example, if only led14 and led15 lights up, it means the frequency is 2000Hz.) Working Principle: The Fast Fourier Transform module uses the volume input from the mic ([11:0] mic_in) to calculate the approximate frequency of the main sound source. With a sampling frequency of 20kHz, the module collects 1000 samples in 0.05 seconds. This module continuously processes the sound signal and output the closest frequency continuously in a period of 0.05 seconds. I used discrete Fast Fourier Transform to process the sample values. The formula is: $g(t) \times e^{-2\pi if t}$. For easier calculation, I separated the value into real and imaginary parts. For example, below is the formula used for testing sound signals of 1000Hz. Since we cannot directly calculate trigonometric values in Verilog, I used Look-Up Table method. The sin and cos modules take in the angle (0~90 degrees) and outputs the corresponding sin and cos values. Since we just need to find the closest sound frequency, the one with the highest amplitude is the main sound signal in the 0.05 seconds time period. Hence, we can calculate result = REAL² + IMAG² and the frequency corresponding to the largest result value is the main sound frequency (the largest result value). Formula: [20 samples per cycle] $\begin{cases} REAL(t) = \sum g(t) \times \cos(2\pi 1000t) \\ IMAG(t) = \sum -g(t) \times \sin(2\pi 1000t) \end{cases}$	<p>LEDs showing the current Audio frequency:</p>  <p>Working Principles of FFT:</p> 
Dynamic Volume Bar with Borders, Texts and Themes	Student B	STATE ON – SW14,13, THEME SEL – SW1,0, BOARDER ON – SW2, BOARDER THICK – SW3, BAR ON – SW4 TEXT ON – SW5 FREEZE – SW6	Toggle to VisualizeBar state by turning SW[14:13] to 2'b01. SW[1:0] = 2'b10 is the default color theme. All other configurations of SW1 and SW0 correspond to a custom color theme that changes color for all elements. Turn on/off SW2 to display/hide a boarder . Turn on/off SW3 to thicken/unthicken it. Turn on/off SW4 to display/hide the volume bar , turn on/off SW 5 to display/hide the text "HIGH VOL" (only displayed when the volume bar reached its top colored portion) and "LOW VOL" (always displayed). Turn on/off SW6 to freeze/unfreeze the screen in VisualizeBar state only.	<p>SW[15:0]=16'b0010000001111110 (And when the volume is loud)</p> 
Graphics Processing Unit with Customized Command Set	Student B	Commands that are within the customized 64-bit command set , each command handled by a unique handler module written in pure verilog	Supports 9 different graphing command types : IDL: Hold the GPU busy for 1 clock cycle, takes 1 busy clock cycle PT: Draw a Point onto the display RAM based on coordinates and color, takes 1 busy clock cycle LN: Draw a Line onto the display RAM based on ending coordinates and color. Uses Bresenham Line Generation Algorithm . Takes variable time. CHR: Draw a character in the character map onto the display RAM based on left-top coordinates, size pre-factor and character index. Supports Size pre-factors of 1,2,4 and 8. Takes variable time. RECT: Draw a rectangular boarder onto the display RAM based on left-top and bottom-right coordinates, and color. Takes variable time. CIRC: Draw a circle boarder onto the display RAM based on centre coordinates, radius and color. Uses Bresenham Circle Generation Algorithm . Takes variable time. SPRSCN: Draw a sprite image from the pre-defined sprite blocks onto the display RAM based on coordinates and magnification ratio of 1,2,4 and 8. Takes variable time. FRECT: Fill a solid rectangle onto the display RAM based on left-top and bottom-right coordinates and color. Takes variable time. FCIRC: Fill a solid circle onto the display RAM based on center coordinates, radius and color. Takes variable time. And supports 3 different logic command types : SBNCH: Check if the immediate state IMME of the GPU is a certain value. If satisfied, jump to destination address to look for the next command. If not,	<p>Geometry using PT, LN, CHR, RECT, CIRC, FRECT, FCIRC (just to showcase what the GPU can do, not in the final bitstream):</p>  <p>Game Scene using DBNCH:</p>  <p>(lost game because Steve bumped into a stone wall)</p>

Student A	Student B	Lab Group
Liu Jingming A0204685B	Li Bozhao A0205636H	Thursday AM 04X Group 01

			<p>resume. Takes 1 busy clock cycle.</p> <p>DBNCH: Check if the immediate state IMME of the GPU is a certain value. If satisfied, jump to destination address 1 to look for the next command. If not, jump to destination address 2 to look for the next command. Takes 1 busy clock cycle.</p> <p>JMP: Unconditionally jump to destination address to look for the next command. Takes 1 busy clock cycle.</p>	
Command Parser Functions	Student B	Header file that can be used to easily generate GPU commands in readable format.	<p>Functions are:</p> <p>DrawPoint, DrawLine, DrawChar, DrawRect, DrawCirc, DrawSceneSprite, FillRect, FillCirc, QuickDrawSceneSprite, SBNCH, DBNCH, JMP, IdleCmd</p>	<p>Sample Usage:</p> <pre>QuickDrawSceneSprite(7'd8, 6'd4, WHITE, 3 DrawRect(7'd8, 6'd22, 7'd89, 6'd41, WHITE DrawChar(7'd10, 6'd25, 20'd22, AQUA, 1'd1)</pre>
Command Queue Structured Scene Builders	Student B	NIL	<p>Scene Builder modules that generate scenes using sequentially executed commands supported by GPU. These modules can each instantiate a GPU core, each module can be either in clocked mode or in instant access mode. Picture on the right is built by the start screen scene builder in instant access mode. It has 15 useful commands in total, drawing 6 sprites, 1 rectangle, 7 chars and having 1 jump command.</p>	<p>Start Screen Scene Builder effect:</p> 
Single Buffered Display	Student B	<p>Any Input Net Connection in the following format:</p> <p>[6:0]X, [5:0]Y, [15:0]C</p>	<p>A first-level screen buffer of size 6143. All graphics are drawn onto the buffer and not directly to the screen. The screen reads the buffer periodically on its own clock, which is separated from the buffer write clock. In this way, we can easily set pixels without needing to worry about synchronizing with the screen clock, and we can set pixels anywhere at anytime as long as we provide a write signal. When trying to freeze the screen, we only need to cut the write signal to the buffer to prevent overwriting. When first entering maze mode, I did not clear buffer before printing text, so the last frame was still in the background. I think this looked nice and kept it as a feature to showcase my buffer.</p>	
Video Decompression Algorithm for Customized Compressed Text	Student B	SW13 and SW14 must be on at the same time to enter video mode	<p>Video mode (SW13 on, SW14 on) plays a video that is around 3 minutes and 30 seconds long at an amazing 10fps with only the puny storage provided by Basys3 FPGA. How? I used my own compression / decompression algorithm. The video was compressed as text file with each 8 bits representing a command specifying how long to move vertically with the current pixel value in a continuous bitstream. The module decodes that and renders the video in real time onto my DRAM. You can exit the video mode any time and when you come back it will continue playing right from where you left it. The video stream decompresses vertically although the screen refreshes horizontally because the video has more continuous pixels in vertical direction and it can save space. This could not have been possible without having the display buffer.</p>	<p>Bad Apple</p> <p>3.5 min at 10 fps, just about 500kb:</p> 
Maze Game	Team	<p>SW14 must be on</p> <p>SW 13 must be off</p> <p>PBC (start/reattempt),</p> <p>PBU (moves up),</p> <p>PBD (moves down),</p> <p>PBL (moves left),</p> <p>PBR (moves right)</p>	<p>Toggle SW14 to on position and SW13 to off position to enter the Maze mode. The map of the maze is drawn using the fill rectangle (FRECT) command and the character Steve is drawn using the draw sprite (SPRSCN) command. User can press buttons U/D/L/R to control movement of Steve, each time moving one pixel in the desired direction. The movement is driven on positive edges of the debounced button press signal, thus pressing and holding will not move him continuously. Winning or losing of the game is determined by comparing the position of the top left corner of Steve with the pre-defined movement path. If Steve bumps into any wall, the game will be lost instantly and the "LOSE" scene will be shown. Press central button to retry. If Steve reaches the green box without bumping into a wall, the game is won, and winning scene will be displayed. Press central button to replay. Steve is rectangular and bulky (pay respects to Minecraft!) thus he easily bumps into walls. Must be careful!</p>	
Pre-defined Memory Blocks to reduce hard code	Team	Any input Net connection with the format of: [n:0]INDEX	<p>In order to make future development easier, we needed an interface to provide us with defined codecs so that we can make references to it instead of hard coding the values inside our modules. Thus, we have MemoryBlocks.v, which contains all codecs needed in our project from character maps to trigonometry tables to sprite images. In this way our logic can be decoupled from our data and we can have modules that can be better re-used.</p>	<pre>DisplayRAM, CommandQueue, ReadOnlyBadAppleCompressedData, AudioVisualizationSceneBuilder, StartScreenSceneBuilder, CharBlocks, SceneSpriteBlocks, MazeSceneBuilder, FFI_sin, FFI_cos</pre>
Peripheral Integration	Team	ALL INPUTS	<p>Peripherals Module takes in ALL inputs and pre-process them before passing them onto other modules for use. For example, it receives all switch signals and produce SwitchOnPulses, SwitchOffPulses and SwitchStates. It takes in main clock and divides it into 100MHz, 6.25MHz, 20kHz, self-defined frequency and 10Hz clock channels. It takes in button inputs and produce ButtonPressPulses, ButtonReleasePulses and ButtonStates. All signals that goes out of this module have been debounced using the main 100MHz clock. Only some are connected.</p>	

Feedbacks:

The project helped us to better understand the concepts learned and how to use them. We learned how to use finite state machine to control the states, we also learnt a lot about sequential circuits and multiple channel clock. I think the best part of this project is that it gives us an opportunity to implement what we have learned in class to develop interesting products, and to develop upon them to make interesting circuits. It would be better if improvements weighted more in marks.